Lotus. software

IBM

Lotus Domino Designer 6

LotusScript Language Guide

Disclaimer

THIS DOCUMENTATION IS PROVIDED FOR REFERENCE PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS DOCUMENTATION, THIS DOCUMENTATION IS PROVIDED "AS IS" WITHOUT ANY WARRANTY WHATSOEVER AND TO THE MAXIMUM EXTENT PERMITTED, IBM DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SAME. IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, DIRECT, INDIRECT, CONSEQUENTIAL OR INCIDENTAL DAMAGES, ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS DOCUMENTATION OR ANY OTHER DOCUMENTATION. NOTWITHSTANDING ANYTHING TO THE CONTRARY, NOTHING CONTAINED IN THIS DOCUMENTATION OR ANY OTHER DOCUMENTATION IS INTENDED TO, NOR SHALL HAVE THE EFFECT OF, CREATING ANY WARRANTIES OR REPRESENTATIONS FROM IBM (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF THE APPLICABLE LICENSE AGREEMENT GOVERNING THE USE OF THIS SOFTWARE.

Copyright

Under the copyright laws, neither the documentation nor the software may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form, in whole or in part, without the prior written consent of IBM, except in the manner described in the documentation or the applicable licensing agreement governing the use of the software.

© Copyright IBM Corporation 1985, 2002

All rights reserved.

Lotus Software
IBM Software Group
One Rogers Street
Cambridge, MA 02142

US Government Users Restricted Rights — Use, duplication or disclosure restricted by GS ADP Schedule Contract with IBM Corp.

List of Trademarks

1-2-3, cc:Mail, Domino, Domino Designer, Freelance Graphics, iNotes, Lotus, Lotus Discovery Server, Lotus Enterprise Integrator, Lotus Mobile Notes, Lotus Notes, Lotus Organizer, LotusScript, Notes, QuickPlace, Sametime, SmartSuite, and Word Pro are trademarks or registered trademarks of Lotus Development Corporation and/or IBM Corporation in the United States, other countries, or both. AIX, AS/400, DB2, IBM, iSeries, MQSeries, Netfinity, OfficeVision, OS/2, OS/390, OS/400, S/390, Tivoli, and WebSphere are registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Pentium is a trademark of Intel Corporation in the United States, other countries, or both. Microsoft, Windows, and Windows NT are registered trademarks of Microsoft Corporation in the United States, other countries, or both. UNIX is a registered trademark of The Open Group in the United States and other countries. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

All other trademarks are the property of their respective owners.

Contents

Preface xiii

License information xiii

System requirements xiii

Printed documentation and PDF files xiii

Related information xiii

Table of conventions xiv

Structure of Notes and Domino documentation xiv

1 Introduction to LotusScript 1-1

What is LotusScript? 1-1

Advantages of LotusScript 1-2

Working with scripts 1-4

Working in the script editor 1-4

Working with Lotus software 1-6

Debugging applications 1-7

Enhancements in LotusScript 5.0 1-8

2 Script and Statement Construction Rules 2-1

Statement construction rules 2-1

Literal number construction rules 2-2

Literal string construction rules 2-3

Identifier construction rules 2-4

Labels 2-5

Keywords 2-6

Alphabetical listing of LotusScript keywords 2-6

Special characters 2-10

3 Data Types, Constants, and Variables 3-1

Summary of LotusScript data types 3-1

Data type conversion 3-2

Explicit data type conversion 3-4

Automatic data type conversion 3-5

Constants and variables 3-9

Scope of declarations 3-10

Constants 3-13

Built-in constants 3-14

Constants defined in LSCONST.LSS 3-15

Constants defined in LSPRVAL.LSS 3-15

Product-specific constants 3-15

User-defined constants 3-15

Variables 3-19

Declaring scalar variables explicitly 3-19

Declaring scalar variables implicitly 3-24

Examples of scalar variables 3-26

Arrays 3-29

Fixed arrays 3-32

Dynamic arrays 3-38

Lists 3-42

Working with lists 3-44

Variants 3-48

Boolean values 3-50

Dates/time 3-51

Referring to Variants 3-55

4 Expressions and Operators 4-1

Overview of expressions and operators 4-1
LotusScript operators 4-2
Operator order of precedence 4-3
Table of numeric operators 4-5
Arithmetic operators 4-6
Exponentiation operator 4-7
Negation operator 4-8
Multiplication operator 4-8
Division operator 4-9
Integer division operator 4-10
Mod operator 4-10
Addition operator 4-11
Subtraction operator 4-13
Relational (comparison) operators 4-13
Logical operators 4-18
Bitwise operators 4-19
Boolean operators 4-21
Not operator 4-23
And operator 4-24
Or operator 4-25
Xor operator 4-27
Eqv operator 4-28
Imp operator 4-29
Table of string operators 4-31
String concatenation operators 4-31
String relational (comparison) operators 4-33
Like operator 4-34
Is operator 4-38
IsA operator 4-39

5 Procedures: Functions, Subs, and Properties 5-1

Procedures 5-1
Functions 5-1
Defining functions 5-2
Passing arguments by reference and by value 5-4
Assigning a return value to a function 5-7
Executing a user-defined function 5-9
Values that a function can manipulate . . . 5-12
Subs 5-17
Defining subs 5-18
Executing a sub 5-19
Specialized subs 5-21
Properties 5-22
Declaring and defining properties 5-23
Using properties 5-24

6 File Handling 6-1

File operations 6-1
Sequential files 6-2
Opening sequential files 6-2
Writing to sequential files 6-3
Reading from sequential files 6-3
Random files 6-4
Opening random files 6-4
Defining record types 6-5
Writing to random files in LotusScript 6-5
Reading from random files 6-6
Binary files 6-7
Opening binary files 6-7
Using variable-length fields 6-7
Writing to binary files 6-7
Reading from binary files 6-7
Reading, writing, and closing files 6-8
Opening files 6-10
Reading from files and writing to them . . . 6-10
Closing files 6-13

7 Error Processing 7-1

Types of errors 7-1

Run-time error processing 7-2

Informational functions used in run-time errors 7-2

Using the informational functions 7-3

Statements used in run-time errors 7-6

Managing error number and message: Err and Error statements 7-6

Handling errors: the On Error statement 7-7

On Error Resume Next 7-12

Multiple On Error statements 7-14

8 User-Defined Data Types and Classes 8-1

Overview of user-defined data types and classes 8-1

User-defined data types 8-2

Declaring a variable of a user-defined data type 8-3

Referring to member variables 8-4

Conserving memory when declaring member variables 8-4

Working with data stored in files 8-5

User-defined classes 8-7

Benefits of classes 8-8

Base classes 8-8

Declaring member variables 8-9

Defining member properties and methods 8-9

Public and Private class members 8-13

Private class members 8-13

Initializing member variables 8-14

Public class members 8-14

Testing object references 8-16

Deleting objects 8-17

Managing memory for objects 8-18

Derived classes 8-19

Property and method overriding 8-21

Arrays and lists of classes 8-28

Working with object reference variables 8-29

9 Managing Flow in Scripts 9-1

Flow of execution 9-1

Flow control statements 9-1

Comments and the compiler directive 9-2

Declarations 9-2

Definition statements 9-2

Errors 9-3

Statement labels 9-3

Block statements 9-4

Selecting one or the other with the If...Then...Else statement 9-4

Specifying multiple test conditions with the If...Then...ElseIf statement 9-6

Making a choice with the Select Case statement 9-8

Branching statements 9-10

Transferring control with the GoTo statement 9-10

Using the If...GoTo...Else statement to transfer unconditionally 9-11

Conditional control transfer with the On...GoTo statement 9-12

Transferring control within the same procedure with the GoSub, On...GoSub, and Return statements . . . 9-13

Iterative statements 9-15

Do and Do...While loops 9-15

For...Next loops 9-19

ForAll loops for lists and arrays 9-25

Using the While statement 9-30

Early termination statements 9-30
Stopping procedure execution early using the End statement 9-30
Using the Exit statement for early procedure termination 9-32
10 Managing Asynchronous Web Agents in Domino 10-1
Introduction to multithreading and synchronization in LotusScript 10-1
Advantages of thread-safe agents 10-1
Synchronization functions 10-3
How synchronization works 10-3
Running asynchronous agents on the Domino server 10-7
11 Beyond Core LotusScript 11-1
Lotus software environments 11-1
Determining which product file is being used 11-1
Product classes and objects 11-2
Interacting with the user 11-6
Interacting with other programs 11-9
Functions and statements for working with other programs 11-9
OLE automation 11-12
Calling external C language functions 11-13
Declaring C functions 11-14
Passing arguments to C functions 11-15
Passing strings 11-18
Passing arrays, types, and objects 11-20
Using user-defined data type variables . . 11-24
Return values 11-25
Calling C language functions extended example 11-26
LS2J: Connecting with Java 11-27
About LS2J 11-27
Java security 11-28
System requirements 11-28
Using LS2J 11-28
Using Script Libraries with LS2J 11-29
Invoking a method in a Java object 11-30
Error handling with LS2J 11-32
LS2J limitations 11-34
LS2J classes 11-34
JavaClass class 11-35
ClassName property 11-36
CreateObject method 11-36
getClassMethods method 11-38
getClassProperties method 11-39
GetMethod method 11-40
GetProperty method 11-41
JavaError class 11-42
ErrorMsg property 11-43
StackTrace property 11-44
JavaMethod class 11-46
JClass property 11-47
MethodName property 11-47
Modifier property 11-49
Signature property 11-50
Invoke method 11-50
JavaMethodCollection class 11-51
Count property 11-51
Current property 11-53
getFirst method 11-55
getNext method 11-56
getNth method 11-57
JavaObject class 11-58
JavaProperty class 11-60
JClass property 11-61
PropertyName property 11-61
Modifier property 11-62

Type property 11-63
getValue method 11-64
setValue method 11-66
JavaPropertyCollection class 11-67
Count property 11-68
Current property 11-69
getFirst method 11-70
getNext method 11-71
getNth method 11-72
JavaSession class 11-73
ClearJavaError method 11-74
getClass method 11-75
getLastJavaError method 11-76
Data type mappings 11-77
Basic data types 11-77
Java reference types 11-80
Processing arguments 11-81
Limitations 11-81
LS2J extended example 11-81

12 LotusScript Language Reference 12-1

Abs function 12-1
ACos function 12-2
ActivateApp statement 12-3
ArrayAppend function 12-4
ArrayGetIndex function 12-9
ArrayReplace function 12-10
ArrayUnique function 12-13
Asc function 12-15
ASin function 12-16
ATn function 12-16
ATn2 function 12-17
Beep statement 12-18
Bin function 12-19
Boolean data type 12-20
Bracket notation 12-21
Byte data type 12-22
Call statement 12-23
CBool function 12-25
CByte function 12-27
CCur function 12-27
CDat function 12-28
CDbl function 12-30
ChDir statement 12-31
ChDrive statement 12-32
Chr function 12-33
CInt function 12-33
Class statement 12-34
CLng function 12-38
Close statement 12-39
CodeLock function 12-39
CodeLockCheck function 12-42
CodeUnlock function 12-43
Command function 12-43
Const statement 12-44
Cos function 12-46
CreateLock function 12-47
CreateObject function 12-48
CSng function 12-50
CStr function 12-50
CurDir function 12-51
CurDrive function 12-52
Currency data type 12-52
CVar function 12-53
DataType function 12-54
About data types 12-56
Date function 12-57
Date statement 12-58

DateNumber function 12-59
DateValue function 12-60
Day function 12-61
Declare statement (external C calls) 12-62
Declare statement (forward reference) 12-66
Deftype statements 12-69
Delete statement 12-71
DestroyLock function 12-73
Dim statement 12-73
Dir function 12-79
Do statement 12-81
Dot notation 12-83
Double data type 12-84
End statement 12-84
Environ function 12-85
EOF function 12-86
Erase statement 12-87
Erl function 12-88
Err function 12-89
Err statement 12-91
Error function 12-92
Error statement 12-93
Evaluate function and statement 12-95
Execute function and statement 12-96
Exit statement 12-98
Exp function 12-100
FileAttr function 12-101
FileCopy statement 12-102
FileDateTime function 12-103
FileLen function 12-103
Fix function 12-104
For statement 12-105
ForAll statement 12-107
Format function 12-110
Formatting codes 12-111
Formatting dates and times in Asian languages 12-118
Fraction function 12-123
FreeFile function 12-124
FullTrim function 12-125
Function statement 12-125
Get statement 12-129
GetFileAttr function 12-132
GetObject function 12-134
GetThreadInfo function 12-136
GoSub statement 12-138
GoTo statement 12-139
Hex function 12-140
Hour function 12-141
If...GoTo statement 12-142
If...Then...Else statement 12-144
If...Then...ElseIf statement 12-145
%If directive 12-146
IMESetMode function 12-149
IMEStatus function 12-151
Implode function 12-152
%Include directive 12-153
Input # statement 12-155
Input function 12-158
InputB function 12-160
InputBox function 12-161
InputBP function 12-163
InStr function 12-164
InStrB function 12-166
InStrBP function 12-167
InStrC function 12-168
Int function 12-169
Integer data type 12-170

IsArray function 12-171
IsDate function 12-171
IsDefined function 12-172
IsElement function 12-174
IsEmpty function 12-176
IsList function 12-177
IsNull function 12-177
IsNumeric function 12-178
IsObject function 12-179
IsScalar function 12-180
IsUnknown function 12-181
Join function 12-182
Kill statement 12-183
LBound function 12-184
LCase function 12-185
Left function 12-186
LeftB function 12-186
LeftBP function 12-187
LeftC function 12-188
Len function 12-188
LenB function 12-190
LenBP function 12-191
LenC function 12-193
Let statement 12-194
Line Input # statement 12-196
ListTag function 12-197
LOC function 12-198
Lock and Unlock statements 12-199
LOF function 12-201
Log function 12-201
Long data type 12-202
LSet statement 12-203
LTrim function 12-204
MessageBox function and statement 12-204
Mid function 12-208
Mid statement 12-209
MidB function 12-210
MidB statement 12-210
MidBP function 12-210
MidC function 12-211
Minute function 12-212
MkDir statement 12-213
Month function 12-213
Name statement 12-215
Now function 12-216
Oct function 12-216
On Error statement 12-217
On Event statement 12-220
On...GoSub statement 12-222
On...GoTo statement 12-223
Open statement 12-225
Option Base statement 12-229
Option Compare statement 12-230
Option Declare statement 12-233
Option Public statement 12-233
Print statement 12-234
Print # statement 12-236
Property Get/Set statements 12-238
Put statement 12-242
Randomize statement 12-245
ReDim statement 12-246
Rem statement 12-249
%Rem directive 12-249
Replace function 12-251
Reset statement 12-253
Resume statement 12-254
Return statement 12-255
Right function 12-256

RightB function 12-257
RightBP function 12-257
RightC function 12-258
RmDir statement 12-259
Rnd function 12-259
Round function 12-261
RSet statement 12-262
RTrim function 12-263
Run statement 12-264
Second function 12-264
Seek function 12-265
Seek statement 12-266
Select Case statement 12-268
SendKeys statement 12-270
Set statement 12-273
SetFileAttr statement 12-275
Sgn function 12-277
Shell function 12-277
Shellid function 12-279
Sin function 12-280
Single data type 12-281
Sleep statement 12-281
Space function 12-282
Spc function 12-283
Split function 12-284
Sqr function 12-286
Stop statement 12-286
Str function 12-287
StrCompare function 12-288
StrConv function 12-289
StrLeft function 12-291
StrLeftBack function 12-292
StrRight function 12-293
StrRightBack function 12-294
StrToken function 12-295
String data type 12-297
String function 12-298
Sub statement 12-299
Sub Delete 12-302
Sub Initialize 12-303
Sub New 12-304
Sub Terminate 12-306
Tab function 12-307
Tan function 12-309
Time function 12-309
Time statement 12-310
TimeNumber function 12-310
Timer function 12-311
TimeValue function 12-312
Today function 12-312
Trim function 12-313
Type statement 12-314
TypeName function 12-317
UBound function 12-319
UCase function 12-320
UChr function 12-321
Uni function 12-321
Unlock statement 12-322
Use statement 12-322
UseLSX statement 12-323
UString function 12-325
Val function 12-326
Variant data type 12-327
Weekday function 12-329
While statement 12-330
Width # statement 12-331
With statement 12-332
Write # statement 12-333

Year function 12-335
Yield function and statement 12-337

Appendix A Language and Script Limits A-1

Appendix B Platform Differences B-1

Appendix C LotusScript/REXX Integration C-1

Appendix D LotusScript Aliases D-1

Appendix E Charset Names E-1

Appendix F Compile-time Error Messages F-1

Appendix G Run-time Error Messages G-1

Index Index-1

Preface

The documentation for IBM Lotus Notes, IBM Lotus Domino, and IBM Lotus Domino Designer is available online in Help databases and, with the exception of the Notes client documentation, in print format.

License information

Any information or reference related to license terms in this document is provided to you for your information. However, your use of Notes and Domino, and any other IBM program referenced in this document, is solely subject to the terms and conditions of the IBM International Program License Agreement (IPLA) and related License Information (LI) document accompanying each such program. You may not rely on this document should there be any questions concerning your right to use Notes and Domino. Please refer to the IPLA and LI for Notes and Domino that is located in the file LICENSE.TXT.

System requirements

Information about the system requirements for Lotus Notes and Domino is listed in the *Release Notes*.

Printed documentation and PDF files

The same documentation for Domino, and Domino Designer that is available in online Help is also available in printed books and PDF files.

You can order printed books from the IBM Publications Center at www.ibm.com/shop/publications/order.

You can download PDF files from the IBM Publications Center and from the Documentation Library at the Lotus Developer Domain at www-10.lotus.com/ldd.

Related information

In addition to the documentation that is available with the product, other information about Notes and Domino is available on the Web sites listed here.

- IBM Redbooks are available at www.redbooks.ibm.com.

- A technical journal, discussion forums, demos, and other information is available on the Lotus Developer Domain site at www-10.lotus.com/ldd.

Table of conventions

This table lists conventions used in the Notes and Domino documentation.

Convention	*Description*
italics	Variables and book titles are shown in italic type.
`monospaced type`	Code examples and console commands are shown in monospaced type.
file names	File names are shown in uppercase, for example NAMES.NSF.
hyphens in menu names (File - Database - Open)	Hyphens are used between menu names, to show the sequence of menus.

Structure of Notes and Domino documentation

This section describes the documentation for Notes, Domino, and Domino Designer. The online Help databases are available with the software products. Print documentation can be downloaded from the Web or purchased separately.

Release Notes

The Release Notes describe new features and enhancements, platform requirements, known issues, and documentation updates for Lotus Notes 6, Lotus Domino 6, and Lotus Domino Designer 6. The Release Notes are available online in the Release Notes database (README.NSF). You can also download them as a PDF file.

Documentation for the Notes client

The Lotus Notes 6 Help database (HELP6_CLIENT.NSF) contains the documentation for Notes users. This database describes user tasks such as sending mail, using the Personal Address Book, using the Calendar and Scheduling features, using the To Do list, and searching for information.

Documentation for Domino administration

The following table describes the books that comprise the Domino Administration documentation set. The information in these books is also found online in the Lotus Domino Administrator 6 Help database (HELP6_ADMIN.NSF).

The book *Installing Domino Servers* ships with Domino. The other books are available for purchase, or for free download as PDF files.

Title	*Description*
Upgrade Guide	Describes how to upgrade existing Domino servers and Notes clients to Notes and Domino 6. Also describes how to move users from other messaging and directory systems to Notes and Domino 6.
Installing Domino Servers	Describes how to plan a Domino installation; how to configure Domino to work with network protocols such as Novell SPX, TCP/IP, and NetBIOS; how to install servers; and how to install and begin using Domino Administrator and the Web Administrator.
Administering the Domino System, Volumes 1 and 2	Describes how to register and manage users and groups, and how to register and manage servers including managing directories, connections, mail, replication, security, calendars and scheduling, activity logging, databases, and system monitoring. This book also describes how to use Domino in a service provider environment, how to use Domino Off-Line Services, and how to use IBM Tivoli Analyzer for Lotus Domino.
Administering Domino Clusters	Describes how to set up, manage, and troubleshoot Domino clusters.

Documentation for Domino Designer

The following table describes the books that comprise the Domino Designer documentation set. The information in these books is also found online in the Lotus Domino Designer 6 Help database (HELP6_DESIGNER.NSF) with one exception: *Domino Enterprise Connection Services (DECS) Installation and User Guide* is available online in a separate database, DECS User Guide Template (DECSDOC6.NSF). The printed documentation set also includes Domino Objects posters.

In addition to the books listed here, the *Domino Designer Templates Guide* is available for download in NSF or PDF format. This guide presents an in-depth look at three commonly used Designer templates: TeamRoom, Discussion, and Documentation Library.

Title	*Description*
Application Development with Domino Designer	Explains how to create all the design elements used in building Domino applications, how to share information with other applications, and how to customize and manage applications.
Domino Designer Programming Guide, Volume 1: Overview and Formula Language	Introduces programming in Domino Designer and describes the formula language.

continued

Title	*Description*
Domino Designer Programming Guide, Volumes 2A and 2B: LotusScript/COM/OLE Classes	Describes the LotusScript/COM/OLE classes for access to databases and other Domino structures.
Domino Designer Programming Guide, Volume 3: Java/CORBA Classes	Provides reference information on using the Java and CORBA classes to provide access to databases and other Domino structures.
Domino Designer Programming Guide, Volume 4: XML, Domino DTD, and JSP Tags	Describes the XML and JSP interfaces for access to databases and other Domino structures.
LotusScript Language Guide	Describes the LotusScript programming language.
Domino Enterprise Connection Services (DECS) Installation and User Guide	Describes how to use Domino Enterprise Connection Services (DECS) to access enterprise data in real time.
Lotus Connectors and Connectivity Guide	Describes how to configure Lotus Connectors for use with either DECS or IBM Lotus Enterprise Integrator for Domino (LEI). It also describes how to test connectivity between DECS or LEI and an external system, such as DB2, Oracle, or Sybase. Lastly, it describes usage and feature options for all of the base connection types that are supplied with LEI and DECS. This online documentation file name is LCCON6.NSF.
Lotus Connector LotusScript Extensions Guide	Describes how to use the LC LSX to programmatically perform Lotus Connector-related tasks outside of, or in conjunction with, either LEI or DECS. This online documentation file name is LSXLC6.NSF.
IBM Lotus Enterprise Integrator for Domino (LEI) Installation Guide	Describes installation, configuration, and migration information and instructions for LEI. The online documentation file names are LEIIG.NSF and LEIIG.PDF. This document is for LEI customers only and is supplied with LEI, not with Domino.
IBM Lotus Enterprise Integrator for Domino (LEI) Activities and User Guide	Provides information and instructions for using LEI and its activities. The online documentation file names are LEIDOC.NSF and LEIDOC.PDF. This document is for LEI customers only and is supplied with LEI, not with Domino.

Chapter 1
Introduction to LotusScript

This chapter introduces LotusScript® and describes, in general terms, how to use the script editor to write and modify scripts, how to compile scripts, and how to use the debugger to locate problems in the logic of your applications.

What is LotusScript?

LotusScript is an embedded, BASIC scripting language with a powerful set of language extensions that enable object-oriented application development within and across Lotus software applications. LotusScript allows you to place more complex scripts in a greater variety of locations and events than traditional macros. LotusScript and its development toolset provide a common programming environment across Lotus applications on all platforms supported by Lotus software (such as Windows, AIX, Linux). It is available in:

- IBM Lotus Notes Release 4 and later
- IBM Lotus Approach® 96 Edition and later
- IBM Lotus Freelance Graphics® 96 Edition and later
- IBM Lotus Word Pro 96 Edition and later
- IBM Lotus 1-2-3 97 Edition and later
- IBM Lotus Enterprise Solution Builder

LotusScript offers a wide variety of features. Its interface to Lotus software is through predefined object classes. The products oversee the compilation and loading of user scripts and automatically include class definitions to allow more efficient coding. LotusScript extends the development capabilities of Lotus software by providing:

- The ability to place scripts in a variety of objects and events in many Lotus software applications. LotusScript has a set of extensions beyond Visual Basic, that provide additional power and utility when writing applications using Lotus software.
- A debugger and syntax-directed editor.

- Access to a broad range of product functions through the classes defined for each product.
- Access to external class libraries defined using the LSX Toolkit.

The environment in which you write, debug, and run scripts depends on your Lotus software application. To learn about your product's programming environment, see your product documentation.

Advantages of LotusScript

LotusScript offers the following advantages:

- Superset of BASIC

  Since LotusScript is a superset of the BASIC language, it is easy to learn, especially for Visual Basic users. You can write sophisticated scripts using conditions, branches, subroutines, while loops, and other conventions.

- Cross-platform

  LotusScript is a multi-platform BASIC-like scripting language. It works with platforms such as Windows, Macintosh, OS/2, UNIX, z/OS, and OS/400. Scripts developed on Windows execute unchanged on any other supported platform. This portability is important as desktop applications become workgroup-enabled and documents are e-mailed to or shared by users.

- Object-oriented

  Lotus software provides Object Classes that are available to LotusScript. You can write scripts to access and manipulate these objects. The scripts are event-driven, such as by an action, clicking the object or button, opening a document, or opening a view.

  LotusScript gives you the ability to create your own classes and objects, and easily subclass these classes.

- Included in Lotus software applications

  LotusScript is supported by Lotus software, so these products can access product classes using a product-supplied LotusScript extension. You can use one language to write scripts in different Lotus software applications.

- OLE support

  Using LotusScript, you can create Notes containers for documents created with IBM Lotus SmartSuite applications and other OLE-enabled applications, such as Microsoft Office. You can use external OLE 2.0 automation objects by scripting them, such as 1-2-3 worksheet objects.

  Notes registers itself as an OLE automation server. External applications can use these objects in scripts to create and reference them. LotusScript can combine all the parts and provide the means for controlling and manipulating objects.

- Interoperability with other languages

  You can call formula language and @functions from LotusScript. You can also call Java and JavaScript.

- Integrated Development Environment

  The LotusScript Integrated Development Environment (IDE) provides an interface to create, edit, and debug scripts, and to browse variables and properties of classes. The IDE allows you to write more complex scripts in Notes.

- LotusScript libraries

  You can create function and class libraries in the language and reuse them in other applications or Lotus software applications via the USE statement language extension.

- Extendable through Lotus Software Extensions (LSXs)

  LotusScript allows users to create their own classes and objects, called Lotus software extensions (LSXs). LotusScript classes support single inheritance, constructors/destructors and method overriding. This functionality allows users to take advantage of object-oriented programming, and to rapidly prototype their own custom business objects. For more information about LSXs, visit the Lotus Developer Network at http://www.lotus.com/home.nsf/welcome/developernetwork.

Working with scripts

A script is composed of statements in the LotusScript language. LotusScript code can be organized into applications, modules, sections, functions, and statements.

Lotus software 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.

Besides direct user manipulation, Lotus software applications or the system can also initiate events. For example, editing a document in a database is an event that is internal to a database application.

Working in the script editor

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

You write a script in a space associated with an object and an event; LotusScript then attaches your script to the object and event. The LotusScript language is the same for all products, but the properties, methods, and events are defined for your specific product's objects. 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 example, when the user clicks a command button, LotusScript runs the script that you defined for that command button "click" event.

Some products can automate parts of the scripting process, restricting or eliminating the need to use parts of LotusScript. For more information on your product extensions, see the product documentation.

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

Compiling scripts

An application must be compiled before it will load and execute.

When you compile a script, LotusScript displays messages about any errors it finds, listed in the order in which they are found. There are two types of errors:

- Compile-time error

  Occurs when a script contains an error that LotusScript detects during compilation. You need to fix it before the script can compile and run.

- Run-time error

  Occurs when a script contains an error that could not be predicted during compilation. When one occurs, script execution ends unless your script includes statements to handle the error.

As you fix errors, you 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 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.

For more information about errors, see "Error Processing."

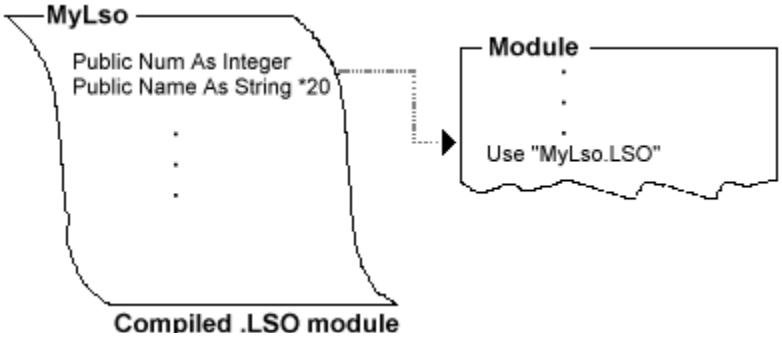
Creating and using script libraries

Script libraries are shared compiled script modules. Some Lotus software applications allow you to write and compile script modules as files with an .LSO extension and then use these files in your applications. You create one copy of a compiled script module to use in multiple applications.

You create the script using your script editor, or any text editor. The script can contain LotusScript declarations, subs, and functions, and can define and declare product classes, properties, subs, and functions.

Note Notes does not allow you to save .lso files directly. Notes saves the object modules within itself.

To load a compiled LotusScript module, put a Use statement in a script at module level, before all implicit 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 "used" module are available to the scripts in the corresponding module. If your Lotus software provides a Public script, place the Use statement in this script to make Public declarations and procedures in the "used" module available to the scripts in the application.

If you then change the name or extension of the module, LotusScript can't use the script module, because 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.

Working with Lotus software

Lotus software provides the environment in which you create, debug, and run LotusScript modules. Each Lotus software application that works with LotusScript 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.

Determining which product file is being used

On the Windows, and some other platforms, you can use command-line arguments (in the Windows 95 Open dialog, for example) to start programs and open program files.

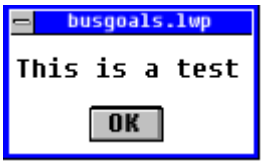
The Command function returns the command-line arguments used to start the Lotus software application from which LotusScript was started. You can use it to get the name of the product file. For example, you may use the file name to identify which product file is currently running, or to provide input for messages to the user.

For example, if 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 then 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$, messageTitle$
```



Debugging applications

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

- Run the application until LotusScript reaches a breakpoint or Stop statement. A breakpoint is a statement at which you want to interrupt application execution.
- Execute one statement, then stop and give control to the debugger.

When you run an application with the debugger, the application is either running or interrupted.

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

Enhancements in LotusScript 5.0

- New LS2J allows LotusScript applications to access Java programs, giving you a powerful cross-platform extension to LotusScript. From within LotusScript, you can access Java classes. See the section "LS2J: Connecting with Java" in "Beyond Core LotusScript" for details.
- An alias is an alternate spelling of a language keyword (usually VB compliant) such as msgbox for the LotusScript messagebox function. Appendix D contains the complete listing of the LotusScript aliases.
- The new Boolean data type specifies a variable that contains a True (-1) or False (0) value. This new data type is stored as only 2 bytes, rather than the 8 bytes required by a Variant, making it a more economical choice for working with Boolean values. See "LotusScript Language Reference" for details.
- The new Byte data type specifies a variable that contains a single, one-byte unsigned number. See "LotusScript Language Reference" for details.
- New function CBool returns an expression converted to the Boolean data type. See "LotusScript Language Reference" for details.
- New function CByte returns an expression that has been converted to a Variant of subtype Byte. See "LotusScript Language Reference" for details.
- New function Implode concatenates all members of an Array of Strings and returns a string. See "LotusScript Language Reference" for details.
- New function Join concatenates all members of an Array of Strings and returns a string. See "LotusScript Language Reference" for details.
- New function Replace replaces specific words or phrases in a string with new words or phrases that you specify. See "LotusScript Language Reference" for details.
- New function Split returns an Array of Strings that are the substrings of the specified String. See "LotusScript Language Reference" for details.
- New function StrToken returns a specified word from a text string. See "LotusScript Language Reference" for details.
- The Open statement has a new optional parameter, *Charset*, which allows you to specify the language to use for file I/O. See "LotusScript Language Reference" for details. See Appendix E for a list of valid MIME charset names.

Chapter 2
Script and Statement Construction Rules

This chapter describes the rules for writing the basic elements of a script in the LotusScript language.

Statement construction rules

- The statements of a script are composed of lines of text. Each text element is a LotusScript keyword, operator, identifier, literal, or special character.
- The script can include blank lines without affecting the meaning.
- The text on a line can begin at the left margin or be indented without affecting the meaning.
- Within a statement, elements are separated with white space, either spaces or tabs. Extra white space can be used between elements to make a statement more readable without affecting the meaning. Avoid using white space around a special character appended to a name.
- A statement, except for a block statement, must appear on a single line unless it includes the line-continuation character underscore (_), preceded by white space.
- The line-continuation character (_) must appear at the end of a line to be continued, preceded by at least one space or tab. Only white space or inline comments (those preceded with an apostrophe) can follow the underscore on the line.
- A new line marks the end of a statement. For block statements, the beginning of the next line starts a new statement.
- Multiple statements on a line must be separated by a colon (:).

Example

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

' One statement on two lines; extra white space
Print "One" & _    ' Comment allowed here
   "Two"
```

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

Literal number construction rules

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

Kind of literal	*Example*	*Legal range*	*Default data type*	*Optional type suffix*
Whole numbers	777	Long -2,147,483,648 to 2,147,483,647.	If the number falls within the range for Integer values, its data type is Integer; otherwise, its data type is Long.	% forces Integer & forces Long ! forces Single # forces Double @ forces Currency
Floating point number	7.7	Double	Double decimal point.	! forces Single # forces Double @ forces Currency
Scientific notation	7.77**E+**02	Double	Double.	! forces Single # forces Double @ forces Currency
Binary number	**&B**1100101	Long	The legal range is the range for Long values. A binary integer is expressible in 32 binary digits of 0 or 1. Values >= &B100000 … (31 zeroes) represent negative numbers. The legal prefix is **&B**.	% forces Integer & forces Long

continued

Kind of literal	*Example*	*Legal range*	*Default data type*	*Optional type suffix*
Octal number	**&O**1411	Long Values >= &O40000000000 are out of range. Values >= &O20000000000 represent negative numbers.	An octal integer is expressible in up to 11 octal digits of 0 to 7. If the number falls within the range for Integer values, its data type is Integer; otherwise, its data type is Long.	% forces Integer & forces Long
Hexadecimal number	**&H**309	Long. Values = > &H80000000 represent negative numbers. Negative signs (**-**) are not allowed.	A hexadecimal number is expressible in 1 to 8 significant hexadecimal digits (excluding leading zeroes). If the number falls within the range for Integer values, its data type is Integer; otherwise, its data type is Long.	% forces Integer & forces Long

Literal string construction rules

A literal string in LotusScript is a string of any characters enclosed in one of the following sets of delimiters:

- A pair of double quotation marks (“ ”)
  ```
  "A quoted string"
  ```
- A pair of vertical bars (| |)
  ```
  |A bar string|
  ```
- Open and close braces ({ })
  ```
  {A brace string}
  ```

Strings enclosed in vertical bars or braces can span multiple lines.

```
|A string
   on two lines|
```

To include one of the closing delimiter characters ", |, or } as text within a string delimited by that character, double it.

```
|A bar string with a bar || in it|
```

The empty string has no characters at all; it is represented by “”.

Strings delimited by vertical bars, braces, or double quotation marks cannot be nested.

```
"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}
```

Identifier construction rules

An identifier is the name you give to a variable, a constant, a type, a class, a function, a sub, or a property.

The following rules govern the construction of identifiers in a script.

- The first character must be an uppercase or lowercase letter.
- The remaining characters must be letters, digits, or underscore (_).
- A data type suffix character (%, &, !, #, @, or $) can be appended, but is not part of the identifier.
- The maximum length is 40 characters, not including the optional suffix character.
- Names are case insensitive. For example, VerED is the same name as vered.
- Characters with ANSI codes higher than 127 (those outside the ASCII range) are legal.

Escape character for illegal identifiers

Some Lotus software classes and OLE classes may define properties or methods whose identifiers use characters not legal in LotusScript identifiers. Variables registered by Lotus software applications might also use such characters. In these cases, prefix the illegal character with a tilde (~) to make the identifier valid.

Examples

```
' $ is illegal as character in identifier
Call ProductClass.LoMethod$            ' Illegal
Call ProductClass.LoMethod~$           ' Legal

X = OLEClass.Hi@Prop                   ' Illegal
X = OLEClass.Hi~@Prop                  ' Legal
```

Labels

A label gives a name to a statement. A label is built in the same way as an identifier.

These statements transfer control to a labeled statement by referring to its label:

- GoSub
- GoTo
- If…GoTo
- On Error
- On…GoSub
- On…GoTo
- Resume

The following rules govern the use of labels in a script:

- A label can only appear 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 starting after the line.
- The last character of a label must be a colon (:).
- A label can't be suffixed with a data type suffix character.
- A given statement can have more than one label preceding it; but the labels must appear on different lines.
- A given label can't be used to label more than one statement in the same procedure.

Keywords

A keyword is a word with a reserved meaning in the LotusScript language. The keywords name LotusScript statements, built-in functions, built-in constants, and data types. The keywords New and Delete can be used to name subs that you can define in a script. Other keywords are not names, but appear in statements: for example, NoCase or Binary. Some of the LotusScript operators are keywords, such as Eqv and Imp.

Note You cannot redefine keywords in a script, with one exception: keywords can name variables within a type, and variables and procedures within a class.

Alphabetical listing of LotusScript keywords

A				
Abs	Access	ACos	ActivateApp	Alias
And	Any	AppActivate	Append	ArrayAppend
ArrayGetIndex	ArrayReplace	ArrayUnique	As	Asc
ASin	Atn	Atn2		

B				
Base	Beep	Bin	Bin$	Binary
Bind	Boolean	Byte	ByVal	

C				
Call	Case	CBool	CByte	CCur
CDat	CDbl	ChDir	ChDrive	Chr
Chr$	CInt	Class	CLng	Close
CodeLock	CodeLockCheck	CodeUnlock	Command	Command$
Compare	Const	Cos	CreateLock	CSng
CStr	CurDir	CurDir$	CurDrive	CurDrive$
Currency	CVar	CVDate		

D				
DataType	Date	Date$	DateNumber	DateSerial
DateValue	Day	Declare	DefBool	DefByte
DefCur	DefDbl	DefInt	DefLng	DefSng
DefStr	DefVar	Delete	DestroyLock	Dim
Dir	Dir$	Do	DoEvents	Double

continued

E				
Else	%Else	ElseIf	%ElseIf	End
%End	Environ	Environ$	EOF	Eqv
Erase	Erl	Err	Error	Error$
Evaluate	Event	Execute	Exit	Exp
Explicit				

F				
FALSE	FileAttr	FileCopy	FileDateTime	FileLen
Fix	For	ForAll	Format	Format$
Fraction	FreeFile	From	FullTrim	Function

G				
Get	GetAttr	GetFileAttr	GetThreadInfo	GoSub
GoTo				

H				
Hex	Hex$	Hour		

I				
If	%If	IMESetMode	IMEStatus	Imp
Implode	Implode$	In	%Include	Input
Input$	InputB	InputB$	InputBox	InputBox$
InputBP	InputBP$	InStr	InStrB	InStrBP
InStrC	Int	Integer	Is	IsA
IsArray	IsDate	IsElement	IsEmpty	IsList
IsNull	IsNumeric	IsObject	IsScalar	IsUnknown

J				
Join				

K				
Kill				

continued

L				
LBound	LCase	LCase$	Left	Left$
LeftB	LeftB$	LeftBP	LeftBP$	LeftC
LeftC$	Len	LenB	LenBP	LenC
Let	Lib	Like	Line	List
ListTag	LMBCS	LOC	Lock	LOF
Log	Long	Loop	LSet	LSI_Info
LSServer	LTrim	LTrim$		

M				
Me	MessageBox	Mid	Mid$	MidB
MidB$	MidBP	MidBP$	MidC	MidC$
- (Minus sign)	Minute	MkDir	Mod	Month
MsgBox				

N				
Name	New	Next	NoCase	NoPitch
Not	NOTHING	Now	NULL	

O				
Oct	Oct$	On	Open	Option
Or	Output			

P				
PI	Pitch	+ (Plus sign)	Preserve	Print
Private	Property	Public	Published	Put

R				
Random	Randomize	Read	ReDim	Rem
Remove	Replace	Reset	Resume	Return
Right	Right$	RightB	RightB$	RightBP
RightBP$	RightC	RightC$	RmDir	Rnd
Round	RSet	RTrim	RTrim$	

continued

S				
Second	Seek	Select	SendKeys	Set
SetAttr	SetFileAttr	Sgn	Shared	Shell
Sin	Single	Sleep	Space	Space$
Spc	Split	Sqr	Static	Step
Stop	Str	Str$	StrComp	StrCompare
StrConv	String	String$	StrLeft	StrLeft$
StrLeftBack	StrLeftBack$	StrRight	StrRight$	StrRightBack
StrRightBack$	StrToken	StrToken$	Sub	

T				
Tab	Tan	Text	Then	Time
Time$	TimeNumber	Timer	TimeSerial	TimeValue
To	Today	Trim	Trim$	TRUE
Type	TypeName			

U				
UBound	UCase	UCase$	UChr	UChr$
Uni	Unicode	Unlock	Until	Use
UseLSX	UString	UString$		

V				
Val	Variant	VarType		

W				
Weekday	Wend	While	Width	With
Write				

X				
Xor				

Y				
Year	Yield			

Special characters

LotusScript uses special characters, such as punctuation marks, for several purposes:

- To delimit literal strings
- To designate variables as having particular data types
- To punctuate lists, such as argument lists and subscript lists
- To punctuate statements
- To punctuate lines in a script

Note: Special characters within literal strings are treated as ordinary text characters.

The following table summarizes the special characters used in LotusScript:

Character	*Usage*
" (quotation mark)	Opening and closing delimiter for a literal string on a single line.
| (vertical bar)	Opening and closing delimiter for a multi-line literal string. To include a vertical bar in the string, use double bars (||).
{ } (braces)	Delimits a multi-line 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)	(1) Separates multiple statements on a line. (2) When following an identifier at the beginning of a line, designates the identifier as a label.
$ (dollar sign)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as String. (2) When prefixed to an identifier, designates the identifier as a product constant.
% (percent sign)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as Integer. (2) When suffixed to either the identifier or the value being assigned in a constant declaration, declares the constant's data type as Integer. (3) Designates a compiler directive, such as %Rem or %If.

continued

Character	*Usage*
& (ampersand)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as Long. (2) When suffixed to either the identifier or the value being assigned in a constant declaration, declares the constant's data type as Long. (3) Prefixes a binary (&B), octal (&O), or hexadecimal (&H) number. (4) Designates the string concatenation operator in an expression.
! (exclamation point)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as Single. (2) When suffixed to either the identifier or the value being assigned in a constant declaration, declares the constant's data type as Single.
# (pound sign)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as Double. (2) When suffixed to either the identifier or the value being assigned in a constant declaration, declares the constant's data type as Double. (3) When prefixed to a literal number or a variable identifier, specifies a file number in certain file I/O statements and functions.
@ (at sign)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as Currency. (2) When suffixed to either the identifier or the value being assigned in a constant declaration, declares the constant's data type as Currency.
* (asterisk)	(1) Specifies the string length in a fixed-length string declaration. (2) Designates the multiplication operator in an expression.

continued

Character	*Usage*
( ) (parentheses)	(1) Groups an expression, controlling the order of evaluation of items in the expression. (2) Encloses an argument in a sub or function call that should be passed by value. (3) Encloses the argument list in function and sub definitions, and in calls to functions and subs. (4) Encloses the array bounds in array declarations, and the subscripts in references to array elements. (5) Encloses the list tag in a reference to a list element.
. (period)	(1) When suffixed to a type variable or an object reference variable, references members of the type or object. (2) As a prefix in a product object reference, designates the selected product object. (3) As a prefix in an object reference within a With statement, designates the object referred to by the statement. (4) 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)	Delimit names used by certain Lotus software applications to identify product objects.
, (comma)	(1) Separates arguments in calls to functions and subs, and in function and sub definitions. (2) Separates bounds in array declarations, and subscripts in references to array elements. (3) Separates expressions in Print and Print # statements. (4) Separates elements in many other statements.
; (semicolon)	Separates expressions in Print and Print # statements.
' (apostrophe)	Designates the beginning of a comment. The comment continues to the end of the current line.
_ (underscore)	When preceded by at least one space or tab, continues the current line to the next line.

Note Use white space to separate names and keywords, or to make the use of a special character unambiguous. White space is not needed with most non-alphanumeric operators. Avoid using white space around a special character, such as a data type suffix character appended to a name.

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.

Summary of LotusScript data types

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

Data type	*Value range*	*Size*
Boolean	0 (False) or -1 (True)	2 bytes
Byte Unsigned short integer	0 to 255	1 byte
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	-1.7976931348623158E+308 to 1.7976931348623158E+308	8 bytes
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	Limited by available memory	2 bytes/character

LotusScript also supports the following 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.	Dynamic or global (public) arrays limited by available memory
List	A one-dimensional set whose elements have the same data type and are referred to by name rather than by subscript.	Limited by available memory
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.	Limited 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, or an object reference to a Java Object.	4 bytes

For more information about language and script limits, see Appendix A.

Data type conversion

LotusScript implicitly converts data from one type to another in the following situations.

Numeric operations

When numeric values with different data types are used in a numeric operation, LotusScript converts the values to the same data type for evaluation.

In general, LotusScript converts to the higher data type, based on this list (lowest to highest): Byte, Integer, Long, Single, Double, Currency. For example, in an operation with one Integer operand and one Double operand, LotusScript converts the Integer value to a Double before evaluating the expression.

Specific rules for conversion in operations are detailed in the documentation of the individual operators.

Argument passing

When a numeric argument is passed by value to a procedure, LotusScript tries to convert the value if it is not the expected data type. If the value is too large, the operation generates an error.

When a numeric argument is passed by reference to a procedure, the data type of the reference must match that of the declared argument, unless the declared argument is Variant.

Variant variables

When a value is contained in a Variant variable, LotusScript tries to convert the value to a number or a string, depending on the context.

Data type conversion treats a value of one data type as though it were a value of a different data type or performs an operation on a value of one data type to produce a value of another data type. 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). You can perform a data type conversion explicitly with the functions that LotusScript provides, you can choose between the two methods of conversion, or LotusScript can perform the conversion automatically. 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.
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. These functions include CBool, CByte, CCur, CDat, CDbl, CInt, CLng, CSng, CStr, and CVar.

This example 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$)

' 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
```

Some conversion facts to keep in mind:

- Format[$] converts almost anything to a string.
- Fix truncates a floating point value to an integer always truncating towards zero.
- Int truncates a floating point value to an integer smaller than the input value.
- DateValue converts a string into a date (variant type 7).
- DateNumber converts a set of numbers into a date value (variant type 7).

Automatic data type conversion

LotusScript can automatically convert values from one data type to another. Automatic, or implicit data type conversion happens when:

- You assign a value of one numeric data type to a variable of a different numeric data type.

  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. For example, the values of the variable anInt% and the variable myLong& are both treated as Long:

```
If anInt% > myLong& Then
  Print "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 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.

  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.

Note It is not always possible to convert values. If the conversion is not possible, a type mismatch error is raised.

Note It is highly recommended that you use explicit conversion as much as possible to avoid unexpected results.

Example 1

```
' 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
```

Example 2

In this 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
```

Example 3

This example illustrates the way in which LotusScript handles data type conversion in Variant variables to accommodate numeric values.

```
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
```

Example 4

This example shows 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.

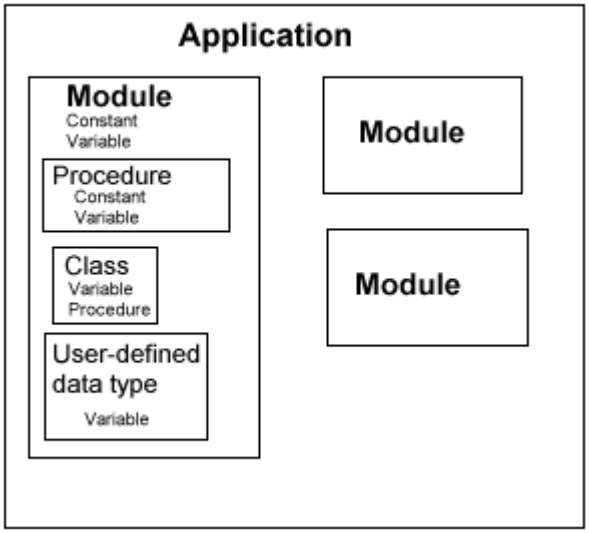
```
Dim aVariantV As Variant
aVariantV = 1040
Print TypeName(aVariantV)
' Output: INTEGER

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 5 can be interpreted as a number.
```

Constants and variables

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 containing data of one or another of the types that LotusScript recognizes. 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 a generic application model and the areas in which you can define constants and declare variables:



Scope of declarations

Scope is the context in which a variable, procedure, class, or type is declared. Scope affects the accessibility of an item's value outside that context. For example, variables declared within a procedure are typically not available outside of the scope of that procedure.

LotusScript recognizes three kinds of scope:

- Module scope
- Procedure scope
- Type or class scope

Name conflicts and shadowing

Two variables or procedures with the same name cannot be declared in the same scope. The result is a *name conflict*. The compiler reports an error when it encounters a name conflict in a script.

Variables or procedures declared in different scopes can have the same name. LotusScript interprets the name as referring to the variable or procedure declared in the innermost scope that is visible where the reference is used.

A variable or procedure of the same name declared at a scope outside of this innermost visible scope is not accessible. This effect is called shadowing: the outer declaration(s) of the name are shadowed, or made invisible, by the inner declaration.

Module scope

A variable is declared in module scope if the declaration is outside of any procedure, class, or type definition in the module. The variable name has a meaning as long as the module is loaded.

The variable name is visible anywhere within the module and has the meaning specified in the declaration, except within a procedure, type, or class where the same variable name is also declared.

The variable is Private by default and can be referred to only within the module that defines it. A variable can be referred to in other modules only if it is declared as Public and the other modules access the defining module with the Use statement.

The following situations result in a name conflict across modules:

- Two Public constants, variables, procedures, types, or classes with the same name
- A Public type with the same name as a Public class
- A Public module-level variable with the same name as a Public module-level constant or procedure
- A Public module-level constant with the same name as a Public module-level procedure

The following situations result in a name conflict within a module:

- A type with the same name as a class
- A module-level variable with the same name as a module-level constant or procedure
- A module-level constant with the same name as a module-level procedure

Procedure scope

A variable is declared in procedure scope if it is declared within the definition of a function, a sub, or a property. Only inside the procedure does the variable name have the meaning specified in the declaration. The variable name is visible anywhere within the procedure.

Ordinarily, the variable is created and initialized when the procedure is invoked, and deleted when the procedure exits. This behavior can be modified with the Static keyword:

- If the variable is declared with the Static keyword, its value persists between calls to the procedure. The value is valid as long as the module containing the procedure is loaded.
- If the procedure itself is declared Static, the values of all variables in the procedure (whether explicitly or implicitly declared) persist between calls to the procedure.

The following situations result in a name conflict within a procedure:

- Two procedure arguments with the same name
- Two labels with the same name
- Two variables with the same name
- A procedure argument and a variable with the same name
- A function that contains a variable or argument of the function name
- A property that contains a variable of the property name

Type or class scope

A variable is declared in type or class scope if it is declared within the definition of a type or a class (for classes, it must additionally be declared outside the definition of a procedure). The variable is called a member variable of the type or class.

- Type member variables: A type member variable is created and initialized when an instance of that type is declared. It is deleted when the type instance or instance variable goes out of scope.

  The visibility of a type member variable is automatically Public.

- Class member variables: A class member variable is created and initialized when an instance of that class is created. It is deleted when the object is deleted.

  Each class member variable can be declared Public or Private. A Private member can only be referred to within the class or its derived classes; class member variables are Private by default.

The visibility of a type member variable (which is always Public) and of a Public class member variable depends, for any particular type or object, on the declaration of the instance variable that refers to that instance:

- If the instance variable is declared Private, then the member variable is visible only in the owning module.
- If the instance variable is declared Public, then the member variable is visible wherever the instance variable is visible: it can be referred to in the other modules where the module that owns this instance variable is accessed with the Use statement.

The following situation results in a name conflict within a type:

- Two type members with the same name.

The following situation results in a name conflict within a class:

- Two class members (variables or procedures) with the same name.

Constants

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 LotusScript, in the file LSPRVAL.LSS. These constants contain information about a thread.
- By an individual product, internally or 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.

The value of a constant is actually compiled into the object code. If you want to change the value of a particular constant, all modules that use that constant must be recompiled.

Built-in constants

LotusScript provides several built-in constants that you can use in your scripts. LotusScript predefines other constants in the file LSCONST.LSS. To include this in your scripts, use the %Include directive.

Constant	*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.

LotusScript also includes an internal value named EMPTY. This is 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.

Language cross-reference

@Pi function in formula language

@Yes function in formula language

@True function in formula language

@False function in formula language

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"
```

Constants defined in LSPRVAL.LSS

LotusScript provides a set of constants that you can use to get information about a running thread. The values of the constants are defined in LSPRVAL.LSS, which is automatically included through LSCONST.LSS.

Product-specific constants

Individual Lotus software applications 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. For more information, see the product documentation.

User-defined constants

You can define your own constants within a module or a procedure, as long as the constants are the scalar data types that LotusScript recognizes. Use the following syntax to define a constant:

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

Where:

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 "Script and Statement Construction Rules"). 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 a data type suffix 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 in the Const statement that defined the constant. You need not use the suffix, though it makes your code easier to read.

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 determine the data type of a constant 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 example:

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

The scope of a constant

Like variables, 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 or variable 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

A variable names an area of storage whose value 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 (except for variables of type Variant). 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 type or scope for the variable, LotusScript chooses 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:

- The scalar types that LotusScript recognizes: Boolean, Byte, 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 defined with a Class...End Class statement, or a class defined by the Lotus software with which LotusScript is running

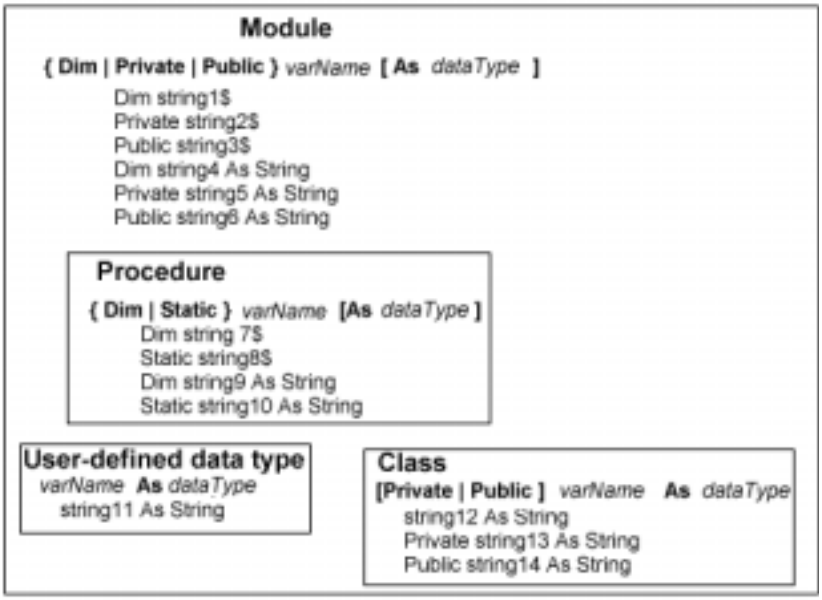
The next two sections describe the two ways you can declare a scalar variable in LotusScript: with an explicit statement or by implication. Subsequent sections describe how to declare arrays, lists, and variables of type Variant.

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. Declaring the variable explicitly is recommended. You declare a scalar variable explicitly with the Dim statement, or one of its variations. The variation you use depends on the application area in which you declare the variable, and on the scope and lifetime you want the variable to have.

Note You can specify OPTION DECLARE to force LotusScript to check for implicit declaration. It will generate a compile time error if any variables are not explicitly declared. It is recommended that applications be checked for implicit declaration before being released.

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 any of the data type suffixes that LotusScript recognizes. This determines the type of data that the variable can hold. You can append a data type suffix to a variable name when you declare it only if you do not include the As *dataType* clause in the declaration.
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.

Initial default values

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

Type of variable	*Initial value*
Numeric (Boolean, Byte, 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)

Note Because LotusScript assigns initial values to variables, you can encounter unexpected errors if the initial value is not what you wanted, or if the variable name is later misspelled, in which case it will be treated as a new variable.

Whether or not you append a data type suffix 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.

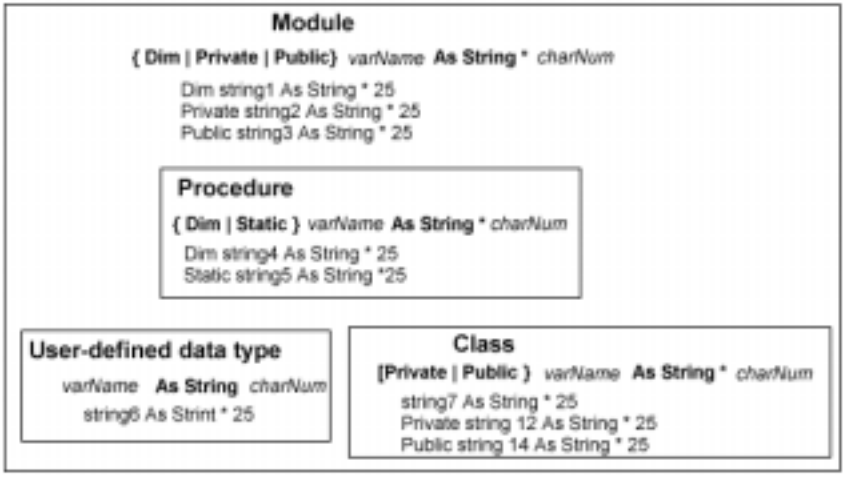
For example:

```
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*], ...

It's important to explicitly declare *all* variables. For example:

```
DIM X, Y AS INTEGER
```

results in Y being data-typed as INTEGER but X as Variant. The correct syntax is:

```
DIM X AS INTEGER, Y AS INTEGER
```

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 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
```

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 to the variable name when you declare it, that suffix determines data type of the variable. If you don't append a data type suffix, one of two things happens: if the name begins with a character covered by an existing Def*type* statement, 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, LotusScript requires you to use that character whenever you subsequently refer to that variable. Omitting the data type suffix 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, 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 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 when referring to a variable of type Variant, you receive 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, include 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.

Note The Boolean and Byte data types do not have a data type suffix character and therefore cannot be declared implicitly.

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
```

Examples of 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

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 (0).
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$.
Print Left$(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. A LotusScript array can have a maximum of eight dimensions.

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.

For example, in a one-dimensional array whose elements are the names of the states of the United States, a single subscript identifies 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, the first subscript identifies the state, and the second subscript 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, Mobile
```

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:

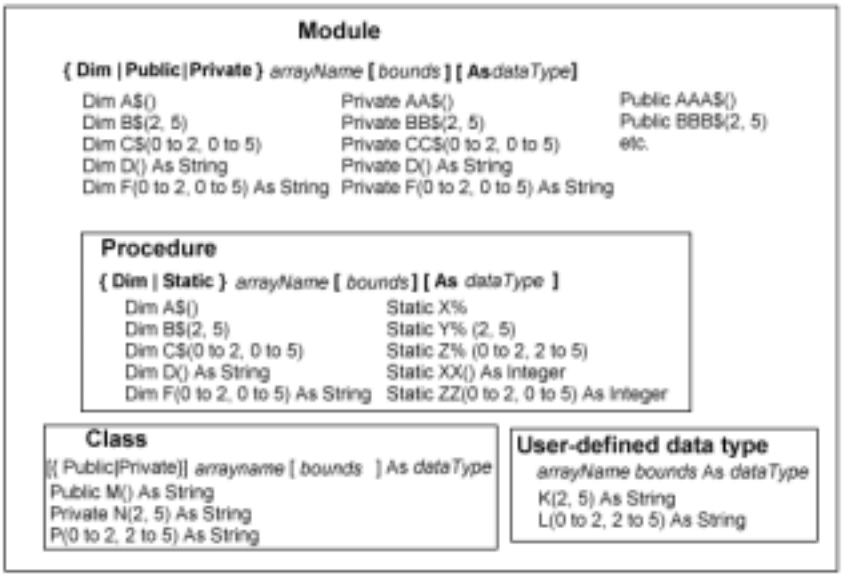
```
Dim statesAnd10CitiesAndPeople(1 to 50, 1 to 10, 1 to 3) _
   As Double
statesAnd10CitiesAndPeople(1,1,1) = 120748
' Number of adult males in Birmingham, Alabama.
statesAnd10CitiesAndPeople(1,1,2) = 145104
' Number of adult females in Birmingham, Alabama.
' ...
statesAnd10CitiesAndPeople(2,1,1) = 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
```

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 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 as follows:

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 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*

continued

Element	*Description*
	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.

Note In many programming languages, such as C, declaring an array

```
int a[5]
```

specifies 5 elements in array **a**, beginning with a[1] and continuing through a[5]. In LotusScript and other BASIC-type languages, declaring

```
DIM a(5) as Integer
```

specifies 6 elements in array **a**, beginning with a[0] and continuing through a[5]. In this case the lower bound of the array is 0 and the upper bound is 5. You can change the base default to 1 instead by using the Option Base statement.

Fixed arrays

You typically 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.

For example, 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 you can 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 size array

When you declare a fixed size 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 belong to 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 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 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 a fixed-sized array local to a procedure can take up no more than 32K bytes 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.

For example:

```
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 as 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, although it is usually 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 by including the appropriate Option Base statement in the module. 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
```

Another example is:

```
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 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.

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(1,1,1) = "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% + 39
     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 dimensions of the array, 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 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
```

Using the Preserve keyword

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. You can later change the values of any of the lower or upper bounds in the bounds list as long as the ReDim statement you use does not include the Preserve keyword. LotusScript then 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.

Using the Erase 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).

Using the built-in functions

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:

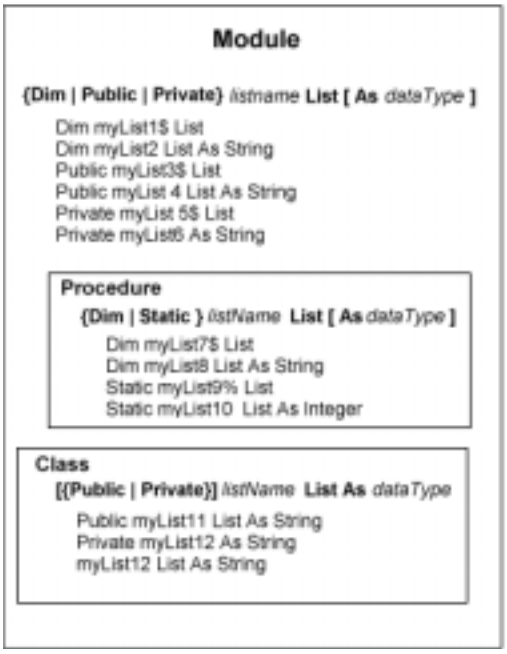
```
' Declare arrays with a base of 1 and containing 10 elements

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. You can change the size of a list at any time while the application is running and LotusScript does not allocate any storage space at compile time for the elements of a list. Lists automatically shrink or grow when elements are deleted from or added to them. You access each element in a list by a unique String value, called a list tag.

You can declare a list 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:



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, LotusScript checks to see if the list name is covered by any applicable Def*type* statement. 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.

A List tag is essentially a key of type STRING. You use this this "key" to uniquely retrieve its associated data once it gets stored.

List tags can be 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 different tags; if case sensitivity is not in effect, they are the same and are

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).

```
' 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.

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*.

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 Boolean
' 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 further 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 following data types that LotusScript recognizes, except for user-defined data types:

- A value of any of the scalar data types that LotusScript supports — Boolean, Byte, Integer, Long, Single, Double, Currency, String
- A date/time value
- 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, or an object reference to a Java Object.
- The NULL value
- The EMPTY value

You declare a Variant variable the same way you declare a scalar variable — explicitly or implicitly. If no Def*type* statements are applicable, a variable that you declare without using an As *dataType* clause or a data type suffix is of type Variant. Here, Variant variables appear with the suffix 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.) Declaring a Variant variable is less efficient than assigning it another data type, but is convenient. 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 a Boolean 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).

Note As of version 5.0, LotusScript also has a Boolean data **type**. This data type is used for variables with values of True (-1) or False (0). See Boolean data type in the LotusScript Language Reference for more information and examples.

```
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 _
k "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/time

LotusScript does not have a date/time data type as such: you can't declare a variable with 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. 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 -657434 (January 1, 100 AD) to 2958465 (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:

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 (0-59) from a date/time expression
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 time. The date part of the value is set to 0 or December 30, 1899.
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

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, 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
```

This example gets the current date and time by calling the function Now and then assigns the result to a Variant variable, the InstantV:

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

This example prints the integers corresponding to the day of the month and the hour of the day:

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

This example assigns the current date to the Variant variable, theDateV:

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

This example converts 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
```

This example gets the integer representation of the current year, month, and day; increments the month and day values and assigns the results to some Integer variables; passes 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
```

This example assigns a string that can be interpreted as a date to a String variable, myDate$; then converts it to a date/time value and performs a calculation on it (subtract a day), and returns 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
```

This example displays the date in a particular print format:

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

Note Various products have different interpretations of two-digit years. Notes, for instance, would write the same value as Oct-18-2014.

This example converts the date/time value of the current date to a value of type Double:

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

This example converts 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
```

This example calculates 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
```

This example determines which day of the week a particular day falls on — Sunday is 1.

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

If the integer part of a value is 0, the value is interpreted as a Time value.

```
Print CDat(0)          'Prints "12:00:00 AM"

Print CDat(.0)         'Prints "12:00:00 AM"

Print CDat(0.0)        'Prints "12:00:00 AM"

Print CDat(0.1)        'Prints "2:24:00 AM"
```

If the fractional part of a value is 0, the value is interpreted as a Date value.

```
Print CDat(1.0)        'Prints "12/31/1899"

Print CDat(2.0)        'Prints "1/1/1900"

Print CDat(123456.0)   'Prints "1/3/2238"
```

Referring to Variants

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.
```

The Variant data type allows you a great deal of freedom in manipulating values of different types (such as 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. It is easy to lose track of the specific data type of a value that you are manipulating, which can sometimes produce unexpected results. 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.

Chapter 4
Expressions and Operators

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

Overview of expressions and operators

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 is a sequence of operators and operands that evaluates to a single value at run time.

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).

LotusScript operators

LotusScript uses the following operators:

- Arithmetic, for performing basic addition operations

```
Print 3 + 4          'Prints 7
```

- Bitwise, for performing bitwise arithmetic

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

- Boolean, for testing two operand expressions for their truth value (True or False)

```
(4 > 0) And (4 < 10)          ' Output is True
```

- Relational (comparison), for comparing values

```
Print 7 <= 8                  ' Prints True
```

- String concatenation, for joining discrete elements to form a single string

```
Print "My cat " & "Geoffrey"  ' Prints My cat Geoffrey
```

- String relational (comparison), for determining the relative positions of two strings in ASCII sort order

```
Print "kid" < "kit"           ' Prints True
```

- Assignment, for assigning values to variables and properties

```
newInt% = 8 + 12

Print newInt%                 ' Prints 20
```

- The Is operator, for comparing the values of object reference variables to see if they are equal.

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

Operator order of precedence

You determine the value of an expression by the order in which the parts are evaluated. Operators with higher precedence are evaluated before operators with lower precedence. Operators with the same precedence are evaluated from left to right.

To override the normal order of evaluation in an expression, use parentheses. Subexpressions in parentheses are evaluated before the other parts of the expression, from left to right.

The following table summarizes the order of operator precedence. The operands in the table are binary except where noted. Operators on the same line have the same precedence. In order of highest-to-lowest, the precedence of LotusScript operators is:

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)	=, <>, ><, <, <=, =<, >, >=, =>, Like	Numeric and 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, Contains (substring matching)
Object reference comparison (Same precedence as Relational)	Is, IsA	Tests object type, refers to the same object
Logical	Not	Logical negation or ones 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
Assignment	=	Assignment

Examples

This example shows the order of precedence for Arithmetic operators.

```
Print 6 + 4 / 2                    ' Prints 8
Print (6 + 4) / 2                  ' Prints 5

Print -2 ^ 2                       ' Prints -4
Print (-2) ^ 2                     ' Prints 4
```

This example shows the order of precedence for Comparison operators:

```
Print 5 < 3                        ' Prints False
Print 5 > 3                        ' Prints True

Print "Alphabet" = "Alpha" & "bet" ' Prints True

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.
```

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
```

Table of numeric operators

You can use these operators 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<CrusaderID_8239_42074650>
	Mod	Modulo division (remainder)
	-, +	Subtraction, addition
Relational (comparison)	=	Equal
	<>	Not equal
	><	Not equal
	<	Less than
	<=	Less than or equal to
	=<	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
	Xor	Logical exclusive Or
	Eqv	Logical equivalence
	Imp	Logical implication

Arithmetic operators

- Exponentiation raises a number to a power.
- Negation returns a number's negative value.
- Multiplication multiplies two numbers.
- Division divides a number and returns a floating-point number.
- Integer division rounds numbers to integers before dividing them.
- Modulo divides numbers and returns the remainder.
- Addition finds the sum of two numbers.
- Subtraction finds the difference between two numbers.

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
```

Only variables of type Variant may be assigned a value of NULL without causing an error.

This example is valid:

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

This example is not valid:

```
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 automatically converts the data type, if possible, or an overflow error results.

```
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 rounds numbers when performing floating point division on integer operands:

```
aDouble# = 42.5
anInt% = 64
anInt% = anInt% / 7
Print anInt%
' Output: 9
' The Mod operator rounds its two operands to Integer
' expressions, divides the first Integer by the second,
' and returns the remainder.
Print aDouble# Mod anInt%
' Output: 6
```

For more information on data type conversion and rounding, see "Automatic data type conversion" in "Data Types, Constants, and Variables."

Exponentiation operator

Raises a number to a power.

Syntax

number **^** *exponent*

Elements

number

Any numeric expression.

exponent

Any numeric expression. If *number* is negative, *exponent* must be an integer value.

Return value

The resulting data type is a Double or a Variant of type Double.

If either or both operands are NULL expressions, the result is a NULL.

Usage

Multiple ^ operators in a single expression are evaluated from left to right.

Language cross-reference

@Power function in formula language

Example

```
Print 4 ^ 3       ' Prints 64
Print 4.5 ^ 3     ' Prints 91.125
Print -2 ^ 3      ' Prints -8
Print 2 ^ 3 ^ 2   ' Prints 64
' Use parentheses to change order of evaluation.
Print 2 ^ (3 ^ 2) ' Prints 512
```

Negation operator

Returns the negative value of a number.

Syntax

-numExpr

Elements

numExpr

Any numeric expression. An EMPTY operand (DataType 0) is considered a 0.

Return value

The result is of the same data type as *numExpr*. The data type of *-v*, where *v* has the value EMPTY, is Long.

If *numExpr* is a NULL, the result is a NULL.

Example

```
Dim x As Integer
x% = 56
Print -x%              ' Prints -56
```

Multiplication operator

Multiplies two numbers.

Syntax

*numExpr1 * numExpr2*

Elements

numExpr1, *numExpr2*

Any numeric expressions. An EMPTY operand is considered a 0.

Return value

The result is a value whose data type is generally the same as that of the operand whose data type is latest in this ordering: Integer, Long, Single, Currency, Double. For example, if one operand is a Double and the other is a Long, then the data type of the result is Double.

The exceptions are:

- If either *numExpr1* or *numExpr2* are NULL expressions, the result is a NULL.
- If *numExpr1* and *numExpr2* are both EMPTY, the result is Integer.
- When the result has a Variant data type of Long, Single, or Date/Time that overflows its legal range, it's converted to a Variant of Double. When the result is a Variant of type Integer that overflows its legal range, it's converted to a Variant of Long.

Example

```
Dim x As Integer
x% = 2 * 3
Print x% * 3.4                ' Prints 20.4
```

Division operator

Divides two numbers and returns a floating-point result.

Syntax

numExpr1 **/** *numExpr2*

Elements

numExpr1, *numExpr2*

Any numeric expressions. An EMPTY operand is considered a 0.

Return value

The resulting data type is a Double or a Variant of Double.

If either or both operands are NULL expressions, the result is a NULL.

Example

This example contrasts ordinary division with integer division. Integer division rounds, divides, and then drops the fractional part. Because the operands are rounded before division, the result may differ from the integer part of an ordinary division operation.

```
Print 8 / 5                   ' Prints 1.6
Print 8 \ 5                   ' Prints 1
Print 16.9 / 5.6              ' Prints 3.01785714285714
Print 16.9 \ 5.6              ' Prints 2
```

Integer division operator

Performs integer division on two numbers and returns the result.

Syntax

numExpr1 \ *numExpr2*

Elements

numExpr1, *numExpr2*

Any numeric expressions. An EMPTY operand is considered a 0.

Return value

The result is of data type Integer, Long, or Variant of type Integer or Long.

If either or both operands are NULL expressions, the result is a NULL.

Usage

LotusScript rounds the value of each operand to an Integer or Long value. Then *numExpr1* is divided by *numExpr2* as an ordinary numerical division; and any fractional part of the result is dropped.

Example

This example contrasts ordinary division with integer division. Integer division rounds, divides, and then drops the fractional part. Because the operands are rounded before division, the result may differ from the integer part of an ordinary division operation.

```
Print 8 / 5                    ' Prints 1.6
Print 8 \ 5                    ' Prints 1
Print 16.9 / 5.6               ' Prints 3.01785714285714
Print 16.9 \ 5.6               ' Prints 2
```

Mod operator

Divides two numbers and returns the remainder.

Syntax

numExpr1 **Mod** *numExpr2*

Elements

numExpr1, *numExpr2*

Any numeric expressions. An EMPTY operand is considered a 0.

Return value

The result is of data type Integer, Long, or Variant of type Integer or Long.

If either or both operands are NULL expressions, the result is a NULL.

Usage

The remainder operator divides *numExpr1* by *numExpr2* and returns the remainder.

The operands are rounded to Integer expressions before the division takes place.

Language cross-reference

@Modulo function in formula language

Example

This example contrasts Modulo division with ordinary division. Mod returns the remainder, while ordinary division returns the dividend.

```
Print 12 Mod 8                ' Prints 4
Print 12 / 8                  ' Prints 1.5
```

Addition operator

Finds the sum of two numbers.

Syntax

numExpr1 **+** *numExpr2*

Elements

numExpr1, *numExpr2*

Any numeric expressions. An EMPTY operand is considered a 0.

Return value

When adding expressions of numeric data types, the result is a value whose data type in most cases is the same as that of the operand whose data type is latest in this ordering: Integer, Long, Single, Double, Currency. For example, if one operand is a Double and the other is an Integer, then the data type of the result is Double.

The exceptions are:

- When the resulting data type is a Variant of Integer that overflows its legal range, the result is converted to a Variant of Long.
- If *numExpr1* and *numExpr2* are both EMPTY, the result has Integer.
- When the resulting data type is a Variant of Long, Single, or Date/Time that overflows its legal range, the result is converted to a Variant of Double.

Usage

LotusScript interprets the + operator as either addition or string concatenation, depending on the data types of *expr1* and *expr2*. The following table lists these interpretations. The numeric data types are Integer, Long, Single, Double, Currency, and (in a Variant variable only) Date/Time.

One expression	*Other expression*	*Operation*
Numeric	Numeric	Addition
Numeric	String	(Type mismatch error occurs)
Numeric	Variant (other than NULL)	Addition
String	Variant (other than NULL)	String concatenation
String	String	String concatenation
Any type	Variant that contains EMPTY	Returns first expression
Any type	NULL	Returns NULL
Variant of numeric data type	Variant of numeric data type	Addition
Variant of numeric data type	Variant of String data type	Addition, if string can be converted to a numeric data type; otherwise a type mismatch error occurs
Variant of String data type	Variant of String data type	String concatenation

To avoid confusion, use the & operator, not the + operator, for string concatenation.

Language cross-reference

@Sum function in formula language

Example

```
Dim a As Variant
Dim b As Integer
a = "8"
b% = 7

' Use operator for addition.
Print 8 + 7                          ' Prints 15
Print a + 7                          ' Prints 15
Print 8 + b%                         ' Prints 15
Print a + b%                         ' Prints 15

' Use operator for string concatenation.
Print "Hello " + "there"             ' Prints "Hello there"
Print a + "7"                        ' Prints "87"
```

Subtraction operator

Finds the difference between two numbers.

Syntax

numExpr1 - numExpr2

Elements

numExpr1, *numExpr2*

Any numeric constant, variable, or expression; or any function that returns a number. An EMPTY operand is considered a 0.

Return value

The result is a value whose data type is generally the same as that of the operand whose data type is latest in this ordering: Integer, Long, Single, Double, Currency. For example, if one operand is a Long and the other is a Currency, then the data type of the result is Currency.

The exceptions are:

- When the result is a Variant of Integer that overflows its legal range, the result is converted to a Variant of Long.
- When the result is a Variant of Long, Single, or Date/Time that overflows its legal range, the result is converted to a Variant of Double.
- If *numExpr1* and *numExpr2* are both EMPTY, the result has Integer.
- If either or both operands are NULL expressions, the result is a NULL.

Example

```
Print 5 - 3.4       ' Prints 1.6
```

Relational (comparison) operators

Relational operators (also called comparison operators) compare two expressions.

Syntax

expr1 operator expr2

Elements

expr1, *expr2*

Any expressions.

operator

One of the following operators: <, >, <=, =<, >=, =>, <>, ><, =.

Return value

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 the chapter "Data Types, Constants, and Variables".

Comparing two expressions, neither of which is NULL, returns the following values:

Operator	*Operation*	*TRUE if*	*FALSE if*
<	Less than	*expr1* < *expr2*	*expr1* >= *expr2*
<= or =<	Less than or equal to	*expr1* <= *expr2*	*expr1* > *expr2*
>	Greater than	*expr1* > *expr2*	*expr1* <= *expr2*
>= or =>	Greater than or equal to	*expr1* >= *expr2*	*expr1* < *expr2*
=	Equal to	*expr1* = *expr2*	*expr1* <> *expr2*
<> or ><	Not equal to	*expr1* <> *expr2*	*expr1* = *expr2*

Usage

LotusScript interprets the relational operator as either numeric comparison or string comparison, depending on the data types of *expr1* and *expr2*. The following table lists these interpretations. The numeric data types are Integer, Long, Single, Double, Currency, and (in a Variant variable only) Date/Time.

One expression	*Other expression*	*Operation*
Numeric	Numeric	Numeric comparison.
Numeric	Variant of numeric data type or Variant containing string value that can be converted to a number	Numeric comparison.
Numeric	Variant containing String value that cannot be converted to a number	Type mismatch error occurs.
Numeric	Variant that contains EMPTY	Numeric comparison, with 0 substituted for the EMPTY expression.
String	String	String comparison.
String	Variant (other than NULL)	String comparison.
String	Variant that contains EMPTY	String comparison.

continued

One expression	*Other expression*	*Operation*
Variant containing string value	Variant containing string value	String comparison.
Variant that contains EMPTY	Variant containing string value	String comparison, with the empty string ("") substituted for the EMPTY expression.
Variant of numeric data type	Variant of numeric data type	Numeric comparison.
Variant that contains EMPTY	Variant of numeric data type	Numeric comparison, with 0 substituted for the EMPTY expression.
Variant of numeric data type	Variant containing string value	Numeric comparison. The numeric expression is less than the string expression.
Variant that contains EMPTY	Variant that contains EMPTY	Expressions are equal.

String comparison

For string comparison, the Option Compare statement sets the comparison method:

- Option Compare Case and Option Compare NoCase specify comparison using the character collating sequence determined by the Lotus software that you are using. Case specifies case-sensitive comparison, and NoCase specifies case-insensitive comparison.
- Option Compare Pitch and Option Compare NoPitch specify comparison using the character collating sequence determined by the Lotus software that you are using. Pitch specifies pitch-sensitive comparison, and NoPitch specifies pitch-insensitive comparison. These options apply to Asian (double byte) characters.
- Option Compare Binary specifies string comparison in the platform's collating sequence. The effect is platform sort-order, case-sensitive comparison.

If you omit the Option Compare statement, the default method of string comparison is the same as Option Compare Case, Pitch.

To compare strings, LotusScript examines the two strings character by character, starting with the first character in each string. The collating sequence values (positions in the character sort sequence) of the two characters are compared.

- If these values are not equal, the string whose character has the larger collating sequence value (appears later in the sort sequence) is the larger string.
- If the collating sequence values of the pair of characters are the same, and both strings contain more characters, then the character comparison proceeds to the next character.

If the end of both strings is reached simultaneously by this process, then neither string has been found larger than the other, and the strings are equal. Otherwise the longer string is the larger string.

Data type conversion

When the operands in a comparison are of different data types, LotusScript automatically converts data types where possible to make the operands compatible before comparing them:

- 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 to highest is:

  BOOLEAN

  BYTE

  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.
- Strings containing values that can be interpreted as numeric types will be converted to numeric types, where necessary.

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 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.

Examples

This example compares numbers.

```
Print 1 < 2                                ' Prints True
Print 2 > 1                                ' Prints True
Print 1 <> 2                               ' Prints True
Print 2 >= 2                               ' Prints True
Print 2 <= 2                               ' Prints True
Print 2 = 2                                ' Prints True
```

This example compares strings.

```
Print "hello" < "hellp"                    ' Prints True
```

```
Dim myVar As Variant, myStr As Variant
myStr = "34"
myVar = 34
Print myVar < myStr                        ' Prints True
Print 45 > myStr                           ' Prints True
Print "45" > myVar                         ' Prints True
```

This example compares two numbers in a more detailed manner:

```
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.
```

Logical operators

You use the logical operators And, Or, Xor, Eqv, and Imp to perform two kinds of operations:

- Bitwise

  Compares the bits in the binary representation of two numeric values and returns 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

  Tests the truth value of a two-operand expression and returns a value of True (-1), False (0), or NULL. 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: True
Print CInt(True And True)
' Output: True
```

You use the logical operator Not to perform the same sorts of operations on expressions consisting of a single operand. Not 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:

- LotusScript converts an EMPTY-valued operand to 0.
- LotusScript rounds a floating-point operand to an integer. 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 results of bitwise operations on two-operand expressions are:

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

For example:

```
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 if the expression is true, and False 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 possibilities are:

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
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.

This example has the user enter two integers between 1 and 10. It tests to see if the first (num1%) is less than 6 and if the second (num2%) is greater than 5. Then it displays 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.
```

Not operator

Performs logical negation on an expression. The Not operator has the effect of rounding its argument to the nearest integer, changing the sign, and subtracting 1.

Syntax

Not *expr*

Elements

expr

Any expression. Its value must lie within the range for Long values.

Usage

The following table explains how LotusScript determines the result of the Not operation.

expr	*Result*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

In addition to performing logical negation, the Not operator reverses the bit values of any variable and sets the corresponding bit in the result according to the following table.

Bit n in expr	*Bit n in result*
0	1
1	0

Example

```
Print Not TRUE                    ' Prints False
Print Not 12.4                    ' Prints -13
```

And operator

Performs a logical conjunction on two expressions. LotusScript rounds to the nearest integer before performing the And operation.

Syntax

expr1 **And** *expr2*

Elements

expr1, *expr2*

Any expressions. Their values must lie within the range for Long.

Usage

When using the And operator, any FALSE expression will cause the result to be FALSE.

expr1	*expr2*	*Result*
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE
TRUE	NULL	NULL
NULL	TRUE	NULL
FALSE	NULL	FALSE
NULL	FALSE	FALSE
NULL	NULL	NULL

Besides combining expressions, And compares identically positioned bits in two numeric expressions (known as a bit-wise comparison) and sets the corresponding bit in the result.

Bit n in expr1	*Bit n in expr2*	*Bit n in result*
1	1	1
1	0	0
0	1	0
0	0	0

Example

```
' Boolean usage

Dim johnIsHere As Boolean, jimIsHere As Boolean

Dim bothAreHere As Boolean

johnIsHere = TRUE

jimIsHere = FALSE

bothAreHere = johnIsHere And jimIsHere

Print bothAreHere                 ' Prints 0 (False)

' Bit-wise usage
Dim x As Integer, y As Integer
x% = &b11110000
y% = &b11001100
Print Bin$(x% And y%)              ' Prints 11000000
```

Or operator

Performs a logical disjunction on two expressions.

Syntax

expr1 **Or** *expr2*

Elements

expr1, *expr2*

Any expressions. Their values must lie within the range for Long values.

Usage

In using the Or operation, both expressions must be FALSE for the result to be FALSE.

expr1	*expr2*	*Result*
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
TRUE	NULL	TRUE
NULL	TRUE	TRUE
FALSE	NULL	NULL
NULL	FALSE	NULL
NULL	NULL	NULL

In addition to performing a logical disjunction, the Or operator compares identically positioned bits in two numeric expressions (known as a bit-wise comparison) and sets the corresponding bit in the result according to the following table.

Bit n in expr1	*Bit n in expr2*	*Bit n in result*
1	1	1
1	0	1
0	1	1
0	0	0

Example

```
' Boolean usage

Dim johnIsHere As Boolean, jimIsHere As Boolean

Dim oneOrMoreIsHere As Boolean

johnIsHere = TRUE

jimIsHere = FALSE

oneOrMoreIsHere = johnIsHere Or jimIsHere

Print oneOrMoreIsHere                ' Prints True

' Bit-wise usage
Dim x As Integer, y As Integer
x% = &b11110000
y% = &b11001100
Print Bin$(x% Or y%)                  ' Prints 11111100
```

Xor operator

Performs a logical exclusion on two expressions.

Syntax

expr1 **Xor** *expr2*

Elements

expr1, *expr2*

Any expressions. Their values must lie within the range for Long values.

Usage

The following table explains how LotusScript determines the result of the Xor operation.

expr1	*expr2*	*Result*
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
TRUE	NULL	NULL
NULL	TRUE	NULL
FALSE	NULL	NULL
NULL	FALSE	NULL
NULL	NULL	NULL

In addition to performing a logical exclusion, the Xor operator compares identically positioned bits in two numeric expressions (known as a bit-wise comparison) and sets the corresponding bit in the result according to the following table.

Bit n in expr1	*Bit n in expr2*	*Bit n in result*
1	1	0
1	0	1
0	1	1
0	0	0

Example

```
' Boolean usage
Dim johnIsHere As Boolean, jimIsHere As Boolean

Dim oneButNotBothIsHere As Boolean

johnIsHere = TRUE

jimIsHere = FALSE

oneButNotBothIsHere = johnIsHere Xor jimIsHere

Print oneButNotBothIsHere         ' Prints True

' Bit-wise usage
Dim z As Integer, y As Integer
z% = &b11110000
y% = &b11001100
Print Bin$(z% Xor y%)              ' Prints 111100
```

Eqv operator

Performs a logical equivalence on two expressions.

Syntax

expr1 **Eqv** *expr2*

Elements

expr1, expr2

Any expressions. Their values must lie within the range for Long values.

Usage

The following table explains how LotusScript determines the result of the Eqv operation.

expr1	*expr2*	*Result*
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	TRUE
TRUE	NULL	NULL
NULL	TRUE	NULL
FALSE	NULL	NULL
NULL	FALSE	NULL
NULL	NULL	NULL

In addition to performing a logical equivalence, the Eqv operator compares identically positioned bits in two numeric expressions (known as a bit-wise comparison) and sets the corresponding bit in the result according to the following table.

Bit n in expr1	*Bit n in expr2*	*Bit n in result*
1	1	1
1	0	0
0	1	0
0	0	1

Example

```
Dim a As Variant, b As Variant, c As Variant
a = &HF
b = &HF0
c = &H33
Print TRUE Eqv TRUE                ' Prints True
Print FALSE Eqv FALSE              ' Prints True
Print TRUE Eqv FALSE               ' Prints False
Print Hex$(a Eqv b)                ' Prints FFFFFF00
Print Hex$(a Eqv c)                ' Prints FFFFFFC3
Print Hex$(b Eqv c)                ' Prints FFFFFF3C
```

Imp operator

Performs a logical implication on two expressions.

Syntax

expr1 **Imp** *expr2*

Elements

expr1, *expr2*

Any expressions. Their values must lie within the range for Long values.

Usage

The following table explains how LotusScript determines the result of the Imp operation.

expr1	*expr2*	*Result*
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	TRUE
FALSE	FALSE	TRUE
TRUE	NULL	NULL
NULL	TRUE	TRUE
FALSE	NULL	TRUE
NULL	FALSE	NULL
NULL	NULL	NULL

In addition to performing a logical implication, the Imp operator compares identically positioned bits in two numeric expressions (known as a bit-wise comparison) and sets the corresponding bit in the result according to the following table.

Bit n in expr1	*Bit n in expr2*	*Bit n in result*
1	1	1
1	0	0
0	1	1
0	0	1

Example

```
Dim youCanSee As Boolean, lightIsOn As Boolean

' You don't need the light to see.

youCanSee = TRUE

lightIsOn = FALSE

Print youCanSee Imp lightIsOn          ' Prints False

' You need the light to see.

youCanSee = FALSE

lightIsOn = FALSE

Print youCanSee Imp lightIsOn          ' Prints True
```

Table of string operators

You can use these operators 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)

String concatenation operators

Concatenate two expressions as strings.

Ampersand (&) operator

Syntax

expr1 **&** *expr2*

Elements

expr1, *expr2*

Any String expressions, or any of the following:

- Numeric expression: LotusScript converts it to its text representation. In the case of Boolean data types, 0 becomes "False," and any other number becomes "True."
- NULL: LotusScript treats it as an zero-length String value when concatenated with the other expression. If both expressions are NULL, the result is NULL.
- EMPTY: LotusScript treats it as a zero-length String value.

Return value
The result is a String or a Variant of type String, if either of the operands is a Variant.

Usage
Use the ampersand (&) operator to ensure a concatenation operation. The plus (+) operator also concatenates two character strings, but 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.

Examples
```
Dim x As Variant
x = 56 & " Baker St."
Print x                          ' Prints "56 Baker St."

anInt% = 123
aString$ = "Hello"
anotherString$ = "world"
varV = NULL
Print aString$ & ", " & anInt% & " " & varV & _
    anotherString$ & "."
' Output: Hello, 123 world.
```

Plus (+) operator

Syntax
expr1 **+** *expr2*

Elements
expr1, *expr2*

Any String expressions, or any of the following:

- Numeric expression: LotusScript converts it to its text representation (if plus is interpreted as concatenation).
- NULL: LotusScript treats it as NULL. If either expression is NULL, the result is NULL.
- EMPTY: LotusScript treats it as a zero-length String value.

Return value
The result is a String or a Variant of type String, if either of the operands is a Variant.

Usage

Use the ampersand (&) operator to ensure a concatenation operation. The plus (+) operator concatenates two character strings, but 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.

For example:

```
Print 100 & "200"

' Output is 100200, because & is always
' a concatenation operator
```

while

```
Print 100 + "200"

' Output is 300, because + was interpreted
' as addition operator

Print "100" + "200"

' Output is 100200, because + was interpreted
' as concatenation operator
```

String 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, False, 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. Option Compare Case (the default) makes string comparison case sensitive; Option Compare NoCase makes string comparison case insensitive.

You can also make string comparison case sensitive with Option Compare Binary. This specifies that string comparison is case sensitive, and the sort order is determined by the platform and character set on which your product is running LotusScript.

For Asian (double-byte) characters, whether the comparison is pitch sensitive or pitch insensitive depends on the setting of the Option Compare statement in the module in which the comparison takes place. Option Compare Pitch (the default) makes string comparison pitch sensitive; Option Compare NoPitch makes string comparison pitch insensitive.

This example illustrates using of relational operators to perform string comparison. 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 operator

Determines whether a string expression matches a pattern string.

Syntax

stringExpr **Like** *patternString*

Elements

stringExpr

Any String expression.

patternString

A string expression that can include any individual ANSI characters and any of the wildcard characters or collections that follow. You can use as many wildcards as you need within a single *patternString*.

Wildcard	*Matches*
?	Any one character
#	Any one digit from 0 through 9
*	Any number of characters (zero or more)
[*characters*]	Any one of the *characters* in the list or range specified here
[**!***characters*]	Any one character not included in the list or range of *characters* specified here

Matching characters in a list

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).

Matching characters in a range

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.

If binary comparison (Option Compare Binary) is in effect, LotusScript uses the international ANSI character collating sequence. This is the sequence of values Chr(0), Chr(1), Chr(2), ... It is the same on all LotusScript platforms. A range specified in ascending order will produce a valid pattern string. However, if Option Compare Case, NoCase, Pitch, or NoPitch is in effect, then the collating sequence order depends on the Lotus software that you are using. The order for alphanumeric characters will be the same as international ANSI, but using other characters to define a range may produce an invalid pattern string. If you define a range using nonalphanumeric characters, specify Option Compare Binary in your script to be certain of defining a valid pattern string.

Matching special characters

To match one of these characters, include it in a *characters* list:

- Hyphen (**-**)
- Question mark (**?**)
- Asterisk (*****)
- Number sign (**#**)
- Open bracket (**[**)

Be sure to place the hyphen at the beginning of the list; if you're using the [!*characters*] format, the hyphen immediately follows the exclamation point, as in [!-123]. The other characters can appear anywhere in the *characters* list. For example, [-?A-Z] matches the hyphen, the question mark, and any uppercase letter from A through Z.

To match one of these characters, place the character anywhere within your wildcard specification except in a *characters* list or range:

- Comma (,)
- Close bracket ( ])
- Exclamation mark (!)

For example, !,[1-6] matches the exclamation point, the comma, and any digit from 1 through 6.

Return value

If *stringExpr* matches *patternString*, the result is TRUE; if not, the result is FALSE. If either *stringExpr* or *patternString* is NULL, the result is NULL.

Usage

By default, the comparison is case sensitive. You can modify case sensitivity with the Option Compare statement.

Language cross-reference

@Like function in formula language

@IsMember function in formula language

@Matches function in formula language

Examples

Example 1

This example prints the two-digit numbers from 1 to 100 that end in 3 and don't begin with 2.

```
For x = 1 To 100
   If CStr(x) Like "[!2]3" Then Print x
Next x
' Output:
' 13 33 43 53 63 73 83 93
```

Example 2

This example uses Like as a validation formula for city and zip fields.

```
if doc.city(0) like "*[0-9]*" then messagebox _
   "city field contains a number"

if doc.zip(0) like "*[a-z,A-Z]*" then messagebox _
   "zip code field contains a character"
```

Example 3

This example shows some ways you can test a string with Like to see if it contains a given substring:

```
' Make string comparison case-sensitive.
Option Compare Case
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 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
```

Is operator

Compares two object reference variables.

Syntax

obj1 **Is** *obj2*

Elements

obj1, *obj2*

Expressions whose values are object references.

Usage

The result of the Is operator is TRUE only if *obj1* and *obj2* refer to the same object or if both operands evaluate to NOTHING. Otherwise, the Is operation returns False (0).

The operands *obj1* and *obj2* may be Variant variables, object reference variables, the constant NOTHING, or any variable elements that accept an object reference, such as an element of an array, list, or user-defined type.

Example

```
Class MyClass
   ' ...
End Class
Dim x As MyClass
Dim y As MyClass
Set x = New MyClass
Set y = New MyClass
Print x Is y         ' Prints False
Set x = y            ' x now refers to the same object as y.
Print x Is y         ' Prints True
```

IsA operator

Determines if an object reference variable is of a specified class or a class derived from the specified class.

Syntax

obj **IsA** *objName*

Elements

obj

Expression whose value is an object reference.

objName

String expression representing an object name.

Usage

The result of the IsA operator is TRUE if *obj* is of the class *objName* or a class derived from *objName*.

Obj can be a native (user defined) object, a product object, or an OLE object.

Obj can be a Variant variable, an object reference variable, or any variable element that accepts an object reference, such as an element of an array, list, or user-defined type or class. *Obj* can be an expression, for example, a function that returns an object or array of objects.

ObjName represents the class that is visible in the current scope if the class name defines more than one class.

Example

```
Sub PrintIt(objA)
     If objA IsA "ClassA" Then
          objA.Print
     Else
          Print "Not a ClassA object"
     End If
End Sub
```

Chapter 5
Procedures: 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 describes the former, while "User-defined Data Types and Classes" describes the latter.

Procedures

Procedures are named sections of a script that you can invoke by name. A procedure in LotusScript takes the form of a function, a sub, or a property. Procedures are primarily ways to organize your code to make it easier to understand and maintain.

A **function** is a named procedure that returns a single value. A **sub** is a named procedure that performs one or more operations without returning a value to its caller. 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.

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.

LotusScript also lets you create your own functions. You define a function by specifying a series of one or more statements that are executed as a block when the application calls the function. You enclose these statements between the function signature and the End Function statement.

A function signature specifies the function 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 and functions (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

Statements and directives that declare or define a function, sub, property, or user-defined data type or class are not allowed within the body of a function, including:

- Declare
- Function
- Sub
- Property Get
- Property Set

Additionally, you may not include the following statements in the body of a function:

- Option
- Use statements
- UseLSX statements

Defining functions

When you define a function, you provide the function's signature and the set of statements that are to be executed when the application calls the function.

The syntax for defining a function is:

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

statements

Element	*Description*
Public, Private	When you declare a function at module level, Public lets the application refer to the function outside the module in which the function is defined, as long as that module is loaded. Private means 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 a LotusScript data type suffixes (%, &, !, #, @, and $). These determine the data type of the function's return value. You can append a data type suffix to a function name when you declare the function 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. ByVal is optional. *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 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 (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. If you omit this clause and *functionName* doesn't end in a data type suffix (and isn't covered by an existing Def*type* statement), the function's return value is Variant.

Declaring functions

In releases of LotusScript before 4.0, there were situations where it was required to declare functions before they were referenced. In LotusScript 4.0, this is no longer needed and forward declarations of LotusScript functions are accepted and ignored.

The syntax for declaring a function is:

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

Passing arguments by reference and by value

LotusScript provides two ways to pass arguments to functions and subs:

- By reference (default)

  When you pass an argument by reference, you pass a pointer to the value in memory. The function operates on the argument. When a function changes the value of an argument passed by reference, the original value changes.

- By value

  When you pass an argument by value, you pass a copy of the value in memory. The function operates on the copy. This means that when a function changes the value of an argument passed by value, the effect is local to that function; the copy changes but the original value in memory is not affected.

Whether an argument is passed by reference or by value depends on the data type and other characteristics of the argument:

- Arrays, lists, type instances, and objects must be passed by reference.

Note An array parameter should not be declared as "ByVal"; a Function or Procedure call should not have parentheses around an array argument.

- Constants and expressions are automatically passed by value.
- Other arguments can be passed either way, as specified in the definition or the call. Arguments to functions and subs are passed by reference unless the definition or the call specifies passing by value.

Passing by reference

The variable must have the same data type as the corresponding parameter in the function definition, unless the parameter is declared as Variant or is an object variable. An object variable can be passed to an object of the same, base, or derived class. In the latter, the base class must contain an instance of the derived class or a class derived from the derived class.

If the variable is then modified by the function or sub, the variable has the modified value when the function or sub returns.

Passing by value

You can do the following:

- Use the ByVal keyword in the argument's declaration in the function or sub definition.

  The argument is passed by value whenever the function or sub is called.

- Insert parentheses around the argument in the function or sub call.

  You can control whether an argument is passed by reference or by value at the time when the function or sub is called.

A value passed to a function or sub is automatically converted to the data type of the function or sub argument if conversion is possible. A Variant argument will accept a value of any built-in data type; and any list, array, or object. A Variant argument will not accept a value of a user-defined type. Keep in mind, however, that lists, arrays, objects, and user-defined types cannot, and therefore should not, be passed by value.

If the variable argument is then modified by the function or sub, the variable has its original value after the function or sub returns. The function or sub operates only on the passed copy of the variable, so the variable itself is unchanged.

Examples

Example 1

```
' Define a function FOver with three Integer parameters:
' a variable, an array variable, and a list variable.
Function FOver(a As Integer, b() As Integer, c List _
As Integer)
   ' ...
End Function

Dim x As Integer
Dim y(5) As Integer
Dim z List As Integer
```

```
' Call the function FOver correctly, with arguments
' whose types match the types of the declared parameters.
Call FOver(x, y, z)
```

Example 2

```
' Define a function GLevel with one Integer list parameter.
Function GLevel (b List As Integer)
   ' ...
End Function

Dim z List As Integer

' Call the function GLevel incorrectly, passing a list
' argument by value.
Call GLevel ((z))
' Output:
' Error: Illegal pass by value: Z
' A list argument cannot be passed by value.
```

Example 3

```
' Define a function FExpr with two Integer parameters;
' the second must always be passed by value.
Function FExpr(a As Integer, ByVal b As Integer)
   ' ...
End Function

Dim x As Integer, w As Integer
Dim y(5) As Integer
Dim z List As Integer

' Call the function FExpr correctly with two Integer
' arguments: a constant and a variable.
Call FExpr(TRUE, x)
' Both arguments are passed by value:
' the first, TRUE, because it is a constant;
' and the second, x, because of the ByVal declaration
' in FExpr.

' The following call produces two error messages:
Call FExpr(TRUE, y)
' Output:
' Error: Illegal pass by value: Y
' Error: Type mismatch on: Y
' Because Y is an array variable, it is an illegal argument to
' pass by value and its type does not match the declared
' parameter type.
```

Example 4

```
' When a function modifies one of its parameters,
' the argument value is changed after the function returns
' if the argument was passed by reference. The value is not
' changed if the argument was passed by value.

Function FTRefOrVal(a As Integer) As Integer
   FTRefOrVal = a + 1
   a = a + 5
End Function

Dim x As Integer, y As Integer

' Show results of passing argument by reference.
Print x, FTRefOrVal(x), x
' Output:
' 0            1            5
' The value of x was changed from 0 to 5 in FTRefOrVal.

' Show results of calling with argument by value
' (note the extra parentheses around y%).
Print y, FTRefOrVal((y)), y
' Output:
' 0            1            0
' The value of the copy of y was changed from 0 to 5
' in FTRefOrVal. The value of y is unchanged.
```

Assigning a return value to a function

One of the statements that you typically include in the function definition assigns the function a return value, that is, a value that it returns to the caller.

The syntax is:

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

The way you execute a user-defined function depends on 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 name can end in empty parentheses or no parentheses.

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
```

You can call a parameterless function by entering the function 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 means that you are passing the argument by value rather than by reference.

For example:

```
Call Cubit2#(anInt%)
' Output: 4 cubed is 64. (anInt% is passed by reference.)
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 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 recursive function is a function that calls itself. A call to itself from within the function is called a recursive call.

The definition of a recursive function must provide a way to end the recursion.

The depth of recursion is limited by a 32K byte stack size.

When recursively calling a function that has no arguments, you must insert empty parentheses following the function name in the call if you use the function's return value. The parentheses show that the function is being called. The function name without parentheses is interpreted as the variable that represents the return value of the function.

Example 1

```
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
```

Example 2

This example shows a recursive function without arguments:

```
Function Recurse As Integer
   ' ...
   ' Call Recurse and assign the return value to x.
   x = Recurse()
   ' ...
   ' Assign the current value of the Recurse variable to x.
   x = Recurse
   ' ...
End Function
```

Values that a function can manipulate

The values that a function can manipulate are:

- 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.

Module-level variables

As long as a function doesn't define a local variable with the same name, it can access a variable defined at module level.

For 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 because you can 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 to 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.

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. 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.

When you define a local variable with the same name as a module-level variable, the procedure uses the local variable and ignores the module-level variable. This is known as shadowing.

For example, defining counter% as a local variable makes this example work properly. 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
```

This example shows static and nonstatic local variables and 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 (the 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 asking the user to try again or quit. 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 Boolean
  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 Boolean
    Dim goAgain As Boolean
    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 application prompts "Please enter your
name."
' The user enters the name "Martin"
' The application answers  "Your ID is valid, Martin."
' Output: (2)The application prompts "Please enter your name."
' The user enters the name "Fred"
' The application answers "That's not a valid name. Would you
' like to try again?"
' The user selects No
```

```
' The application answers "Have a nice day, Fred."
' Output: (3)he application prompts "Please enter your name."
' The user enters the name "Fred"
' The application answers "That's not a valid name. Would you
' like to try again?"
' The user selects Yes, then enters "Frank,"
' The application answers "That's not a valid name. Would you
' like to try again?"
' The user selects Yes, then enters "Joe":
' The application answers "Go away, friend."
```

Subs

A sub is a named procedure that performs one or more operations without returning a value to its caller. You define a sub by specifying a series of one or more statements that are to be executed as a block and enclose these statements between the sub signature and the End Sub statement. You can't include a statement that assigns the sub a value.

A sub signature specifies the sub 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).

You can define a sub at module level or as a member of a user-defined class. Declaring a sub before you define it lets you refer to that sub before you actually define it. You use the Declare statement to explicitly declare a sub as a member of a user-defined class or at module level in a product that does not support the Integrated Development Environment (IDE). The IDE automatically generates a Declare statement for each sub that you define at module level, so you should not include any.

For information on the four specialized kinds of sub that you can define — Sub Initialize, Sub Terminate, Sub New, and Sub Delete, see "Specialized subs" later in this chapter.

Defining subs

The syntax for defining a sub is

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

statements

End Sub

Element	*Description*
Public, Private	When you declare a sub at module level, Public lets the application refer to the sub outside the module in which it is defined, as long as that module is loaded. Private means 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.

Declaring a sub

In releases of LotusScript before 4.0, there were situations where it was required to declare subs before they were referenced. In LotusScript 4.0, this is no longer needed and forward declarations of LotusScript subs are accepted and ignored.

The syntax for declaring a sub is:

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

Executing a sub

You can execute a user-defined sub in either of two ways: by including it in a Call statement or by 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 name can end in either empty parentheses or no parentheses.

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 entering the sub 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, enclose the argument in parentheses when you include it in a Call statement. Enclose the argument in single parentheses to pass it by reference, or in double parentheses to pass it by value.

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, enclose the arguments in parentheses when you include the sub in a Call statement, and do 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 subs to automate set-up and clean-up in an application.

Sub Initialize

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 is:

Sub Initialize

statements

End Sub

Sub Initialize is 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

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
```

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 "User-defined Data Types and Classes."

Properties

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. This is especially useful in object-oriented programming. 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: Property Set assigns the value of the property to a variable you want to manipulate, and Property Get assigns the current value of that variable to the property. You execute the Property Set procedure by assigning the property a value, and you execute the Property Get 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

Declaring a property before you define it allows you to refer to that property before you actually define it.

The syntax for declaring a property is:

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:

[**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 lets the application refer to the property outside the module in which it is defined, as long as that module is loaded. Private means 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.

continued

Element	*Description*
propertyName	The name of the property, which can end in a LotusScript data type suffix (%, &, !, #, @, and $). These determine the data type of the property's return value. You can append a data type suffix when you declare the property 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

Properties are good for manipulating protected variables, that is, Private members of a user-defined class to which the application has no direct access.

For more information see "User-defined Data Types and Classes."

Example 1

In the following example, the sub KeepGoing uses the property theCube# to manipulate three variables (anInt%, aDouble#, and bigNum#) that are not 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 Boolean
  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.
```

Example 2

You can perform the same operations 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$("Enter an integer:"))
End Sub
Function GetTheCube(anInt As Integer) As Double
   aDouble# = anInt% ^ 3
   If aDouble# > bigNum# Then
       bigNum# = aDouble#
  End If
  GetTheCube# = anInt%
End Function
Sub KeepGoing
  Dim goAgain As Boolean
  Dim msg As String
  Dim msgSet As Integer
  Dim more As Integer
  goAgain = TRUE
  msg$ = "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.
```

Chapter 6
File Handling

This chapter describes file handling in the LotusScript language.

File operations

The following table describes the three kinds of files in LotusScript: sequential, random, and binary.

File type	*Description*
Sequential	The simplest and most common. It is the equivalent of a common text file. Data in sequential files are delimited with the platform's end-of-line indicator (carriage return, line feed, or both). You can read the file with a text editor.
Random	The most useful for structured data. It is not readable except through LotusScript programs. This is the default.
Binary	The most complex. It requires detailed programming to manipulate, because access is defined at the level of bytes on the disk.

To store and manipulate data in a file, the file must be opened first. When a file is opened in LotusScript, it has a file number, between 1 and 255, which is used in most input and output operations. (A few file operations use the file name instead of a number.) The number remains until the file is closed.

Some file operations that can be performed on these files are:

Close	Close one or more open files.
Get, Input	Read data from a file.
Kill	Delete a file.
Open	Open a file.
Put, Write	Write data to a file.

Sequential files

A sequential file is an ordinary text file. Each character in the file is assumed to be either a text character or some other ASCII control character such as newline. The character is in the character set specified when the file is opened. By default this is the platform-native character set.

Sequential files provide access at the level of lines or strings of text: that is, data that is *not* divided into a series of records. However, a sequential file is not well suited for binary data, because a number in a sequential file is written as a character string.

Opening sequential files

A sequential file can be opened in one of three modes: input, output, or append. After opening a file, you must close it before opening it in another mode.

The syntax is:

Open *fileName* [**For** {**Input** | **Output** | **Append**}] **As** *fileNumber* [**Len =** *bufferSize*] [**Charset** = *MIMECharsetName*]

Where Input means read-only access to the file, Output means write-only access, and Append means write-only access starting at the end of the file. Access in all three sequential modes is one line at a time. To get an unused *fileNumber*, use the FreeFile function.

bufferSize is the number of characters loaded into the internal buffer before being flushed to disk. This is a performance-enhancing feature: the larger the buffer, the faster the I/O. However, larger buffer sizes require more memory. The default buffer size for sequential files is 512 bytes.

MIMECharsetName designates the character set. The default is the platform-native character set, except that if a UTF-16 or UTF-8 byte order mark (BOM) is present, the BOM character set is used. See MIME charset names for a list of valid MIME charset values.

When you try to open a file for sequential input, the file must already exist. If it doesn't, you get an error. When you try to open a nonexistent file in output or append mode, the file is created automatically.

Writing to sequential files

You can write the contents of variables to a sequential file that was opened in output or append mode using the Print # or Write # statement.

The parameters to Print can be strings or numeric expressions; they are converted to their string representations automatically.

This example writes the contents of Var1 and Var2 (separated by tabs, because of the commas in the statement) to the file numbered idFile.Print #idFile, Var1, Var2

```
Print #idfile, Var1, Var2
```

The Write # statement generates output compatible with the Input # statement by separating each pair of expressions with a comma, and inserting quotation marks around strings.

For example:

```
Dim supV As Variant, tailV As Variant
supV = 456
tailV = NULL
Write #idFile, "Testing", 123, supV, tailV
```

The statements generate the following line in the file numbered idFile:

```
"Testing",123,456,#NULL#
```

Note True, False, and NULL are stored as strings "#True#", "#False#", and "#NULL#".

Reading from sequential files

To read data from a sequential file, open the file in input mode. Then use the Line Input # statement, the Input # statement, or the Input function, to read data from the file into variables.

Line Input # reads one line of text from a file, up to an end-of-line. The end-of-line is not returned in the string.

This example shows reading a file one line at a time until end-of-file. The Print statement displays the line and appends an end-of-line sequence.

```
Do Until EOF(idFile)
   Line Input #idFile, iLine
   Print iLine
Loop
```

Input # reads in data that was formatted and written with the Write # statement.

For example:

The file numbered idFile contains the line:

```
"Testing",123,456,#NULL#
```

Then the following statements read "Testing" into liLabel, 123 into infA, 456 into supA, and the value NULL into tailV:

```
Dim liLabel As String, tailV As Variant
Dim infA As Integer, supA As Integer
Input #idFile, liLabel, infA, supA, tailV
```

If you find that you are using Write # and Input # with sequential files often, you should consider using a random file instead. Random files are better suited for record-oriented data.

The Input function reads data from a sequential file. This function takes the number of characters to read as an argument, and returns the characters. The Input$ function returns a string to the caller. The Input function returns a Variant variable.

This example reads an entire file at once into a string variable.

```
' LOF returns the length of the file in characters.

Dim fulFile As String
fulFile = Input$(LOF(idFile), idFile)
```

Random files

A random file is made up of a series of records of identical length. A record can correspond to a scalar data type, such as Integer or String, or to a user-defined type, in which each record is broken down into fields corresponding to the members of the type.

Opening random files

The syntax is:

Open *fileName* **For Random As** *fileNumber* [**Len =** *recordLength*]

where *recordLength* is the length of each record in the file. The default length is 128 bytes.

If the file does not exist, it is created.

Defining record types

Because records in a random file must have the same length, elements of a type should be fixed-length. If a string copied into a file record contains fewer characters than the record's fixed length, the remainder of the record is left unchanged. However, if a string is too long for a record, it is truncated when written.

String fields inside the user-defined type should also be fixed-length. If you do use variable-length, make sure that the Len part of the Open statement specifies a length large enough to hold the longest strings. The Len function can't give you a reliable value for the length of the record; you will need to estimate that. You also can't navigate between records by omitting the record number in the Get and Put statements.

User-defined types can be used to define compound records.

For example:

```
Type emploRec
   id As Integer             ' Integers are 2 bytes long
   salary As Currency        ' Currency is 8 bytes
   hireDate As Double        ' Dates are also 8 bytes
   lastName As String * 15   ' Fixed-length string of 30 bytes
   firstName As String * 15  ' Fixed-length string of 30 bytes
End Type
```

The length of a type can be determined at run time using the Len function.

For example, this record is 78 bytes long, so supply Len = 78 in the Open statement.

```
Dim recLen As Integer, idFile As Integer
Dim recHold As emploRec
idFile = 1               ' The file number to use for
                         ' this file
recLen = Len(recHold)    ' The record length for this file
Open "DATA.DAT" For Random As idFile Len = recLen
```

Writing to random files in LotusScript

Use the Put statement to write to a random file. Put takes three parameters: the file number, the record number, and a variable containing the data you wish to write. You can use Put to add or replace records, but not to delete them. To replace a record in a random file, use its record number.

For example:

```
Dim recNum As Integer
recNum = 5
' Replace record 5 with the contents of recHold.
Put idFile, recNum, recHold
```

To add new records to a random file, use a record number equal to one more than the number of records in the file. To add a record to a file that contains 5 records, for example, use a position of 6.

To replace a record from a random file, create a new file and copy all the valid records from the original file into the new file. Close the original file and use the Kill statement to delete it. Use the Name statement to rename the new file to the same name as the original. You can also move each record, following it "up" by one position, thus writing over the record. The problem with this technique is that it leaves a duplicate record at the end of the file.

For example:

```
Dim tempRec As emploRec
For I = recNum To lastRec - 1
   Get idFile, I + 1, tempRec
   Put idFile, I, tempRec
Next I
```

Reading from random files

Use the Get statement to read from a random file into variables.

This example reads from the file numbered idFile, at record number 5, into the variable recHold.

```
' The record number to retrieve from the file
Dim recNum As Integer
recNum = 5
' The variable to read into
Dim recHold As emploRec
Get idFile, recNum, recHold
```

Binary files

Binary files are designed to provide the most control over the organization of your data for both reading and writing. However, you must know exactly how the file was written.

Opening binary files

The syntax is:

Open *fileName* **For Binary As** *fileNumber*

Record-length arguments are ignored.

If the file does not exist, it is created, regardless of the access type supplied to the Open statement.

Using variable-length fields

Binary files can hold variable-length records. Since you need to know the string sizes to read them, you should assign a length field to each variable-length record (each string). This is not necessary if the string is a component of a user-defined type; in this case, LotusScript automatically assigns one.

Binary access provides a byte-by-byte view of a file. A file appears to be a continuous stream of bytes, which may or may not be alphanumeric characters.

Writing to binary files

To write to a binary file, use this Put statement:

Put *fileNumber*, *bytePosition*, *variableName*

Here, the *bytePosition* parameter is the position in the file at which to start writing. The first byte in a file is at position 1; position zero is illegal, and results in an error.

Reading from binary files

To read data from a binary file, use the following:

- Get

  The Get statement reads the correct number of bytes into any variable of known length, such as a fixed-length string or an integer. For variable-length strings, the number of characters read equals the *current length* of the string. This will be zero for uninitialized variable-length strings so you should first set the current length to the length of the

string to be read. If the string in the file is within a user-defined type, the string length was stored by LotusScript with the string.

- Seek

  The Seek statement sets the byte position in an open file. The syntax is:

  Seek [#] *fileNumber, position*

  where *fileNumber* is the number assigned to the file when it was opened and *position* is the desired file position for the next read operation. In a binary file, this is a non-zero byte location. The record number in a Get statement or Put statement overrides a file position set by a Seek statement.

- Input

  The Input function or the Input$ function also reads data from a binary file. The syntax is:

  dataHold = **Input (***numBytes***,** *fileNumber***)**

  where dataHold is a Variant. (If the Input$ function were used instead of the Input function, dataHold is a String.) This function reads *numBytes* bytes from the file and returns them in the variable dataHold.

Reading, writing, and closing 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 creates it automatically.

LotusScript provides three modes of file access:

- Sequential (input, output, or append)

  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.

- Random

  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 “User-defined Data Types and Classes.”

- Binary

  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 also use binary access to write a stream of characters to an unstructured text file.

This table summarizes the statements and functions that operate on Sequential, Random, and Binary files.

Statements & Functions	*Sequential*	*Random*	*Binary*
Dir	X	X	X
EOF	X	X	X
FileCopy	X	X	X
FileDateTime	X	X	X
FileLen	X	X	X
FreeFile	X	X	X
GetAttr	X	X	X
Loc	X	X	X
LOF	X	X	X
Seek	X	X	X
SetAttr	X	X	X
Open (This is where you set the access mode.)	X	X	X
Close	X	X	X
Get	X	X	X
Input( )	X		X
InputB( )	X		X
InputBP( )	X		X
Input #	X		
Line Input #	X		
Print #	X		
Put		X	X
Write #	X		

Opening files

Use the FreeFile function to get a file number, and then use an Open statement to open a file.

The syntax is:

```
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%]

                    [Charset = MIMECharsetName]
```

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. You can also specify a character set to use for sequential access. See the Open statement for usage details.

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:

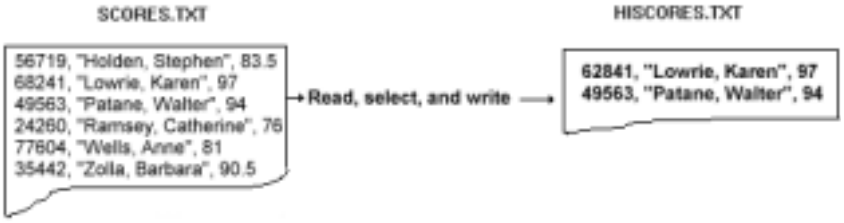
```
fileNumber% = FreeFile
Open "DATA.DAT" For Input As fileNumber%
fileContents$ = Input(LOF(fileNumber%), fileNumber%)
```

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 writes over 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 write over to the end of the file.

```
Open "DATA.DAT" 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.DAT 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.DAT. The process continues until the EOF function returns TRUE (-1), indicating that the script has reached the end of SCORES.DAT.




```
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.DAT" For Input As fileNum1%
   fileNum2% = FreeFile
   Open "HISCORES.DAT" For Append As fileNum2%
   While Not EOF(fileNum1%) ' Read until end of file.
     Input #fileNum1%, undergrad.ID, undergrad.Name,
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 open the file in input mode (replace the previous contents of the file) or append to the file. 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 multi-line strings).

Note Newline does not mean either chr(10) or chr(13) on all platforms. Newline is the character or sequence of characters that is used to mark the end of a line. This may be chr(10), or chr(13), but it may also be something else, because the actual value of newline depends on the platform.

Note The Line Input # statement will handle the line end character appropriate for the current platform. It will not necessarily handle line ends properly if the file is written on one platform and read on another.

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 and 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.DAT" For Random Access Read As fileNum1% _
                         Len = Len(undergrad)
   fileNum2% = FreeFile
   Open "GOODSCORES.DAT" For Random Access Write _
                         As fileNum2% Len = Len(undergrad)
```

```
   While Not EOF(fileNum1%) ' Read until end of file.
      Get #fileNum1%,, 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 also close the file, then open it again.

Chapter 7
Error Processing

This chapter describes error processing in the LotusScript language.

Types of errors

LotusScript recognizes two kinds of errors:

- Compile-time errors

  Errors that are found and reported when the compiler attempts to compile the script. Common compile-time errors are syntax or naming errors.

  The compiler reports the error, together with a message and a link to online Help, which explains how to correct the error. You must correct the error and re-compile before the script can run.

- Run-time errors

  Errors that are found when LotusScript attempts to execute the script. A run-time error can't be predicted at compile time (e.g., "out of memory"). 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. Examples of run-time errors are attempting to open a file that doesn't exist, or attempting to divide a number by a variable with a zero value.

  LotusScript recognizes many run-time errors, and identifies each with a name, a number, and a standard message to describe it. Within a script, you can also define your own run-time errors and associate a number and a message with each one.

Note Compile-time errors are also possible at run time. LotusScript has an execute statement. If there are syntactic problems inside the string expression, a compile-time error is generated at run time.

Run-time error processing

A run-time error occurs either when executing a statement results in an error or when LotusScript executes an Error statement.

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. LotusScript records the line number in the script where the error occurred, the error number, and the error message associated with that number, if any. Until an error handling routine is invoked for this error, or another error is encountered, these are, respectively, the return values of the functions Erl, Err, and Error$. (Exception: The Err statement can be used to reset the current error number returned by the Err function.)

LotusScript then looks in the current procedure for an On Error statement associated with this error first, or more commonly, to "clear" the error: an On Error *n* statement, where *n* is the error number; if none, an On Error statement with no error number specified. If none is found, the search continues in the procedure that called this procedure, if any; and so on. For the error to be handled in the current procedure, the procedure must include an On Error statement already executed that refers to the error. If no associated On Error statement is found in any calling procedure, execution ends and LotusScript displays the associated error message.

If an associated On Error statement is found, LotusScript executes the command contained in the On Error statement.

Informational functions used in run-time errors

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.

- Err function

  Returns the LotusScript error number for the current error, or the number you specify with the Err statement. If there is no current error, Err returns FALSE (0). 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.

- Erl function

  Returns the current line number within the procedure where the error handler is defined.

- 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. If there is no current error, Error and Error$ return the empty string "".

Using the informational functions

These examples show how LotusScript manages the error number, its associated error message, and its line number.

Example 1

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
```

Example 2

This example shows 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 written 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
```

Example 3

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
```

Statements used in run-time errors

You include statements in a script to explicitly manage the flow of control when an error occurs.

- The Err statement sets the error number and optionally specifies an error message for it.
- The Error statement creates an error and optionally specifies an error message for it.
- The On Error statement specifies how to handle an error.
- The On Error Resume Next specifies that program execution continues with the next statement after the statement that generates the error.

Managing error number and message: Err and Error statements

The Err statement

The Err statement sets the error number. The Err statement corrresponds to the Err function, which returns the current error number.

The syntax is:

Err = *errNumber*

The error number can be set 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 (“”).

The Err statement does not create an error as the Error statement does. It only resets the error number (and also the value 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 syntax is:

Error = *errNumber* **[** , *msgExpr* **]**

If you do not include the optional *msgExpr* string in the statement, it 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.

Handling errors: the On Error statement

Every error recognized at run time has its own error number that identifies it. When a recognized error happens during script execution, LotusScript records the error number, and then proceeds as directed by an On Error statement that refers to that number.

For example, you can write either one of these On Error statements to tell LotusScript how to respond to an occurrence of error number 357:

```
On Error 357 GoTo apoc600
On Error 357 Resume Next
```

When referring to a pre-defined error in an On Error statement, you can use the defined constant for the error instead of the error number.

For example, here are the statements in LSERR.LSS that define the error numbers and constants for two common errors:

```
Public Const ErrDivisionByZero      = 11
' Division by zero
Public Const ErrIllegalFunctionCall = 5
' Illegal function call
```

On Error statements then do not need to mention the numbers 11 and 5. Write the statements in this form instead, making the script easier to read:

```
On Error ErrDivisionByZero ...
On Error ErrIllegalFunctionCall ...
```

You can define constants for your own error numbers. (You should define your error numbers to be above ErrLast.) Then the constant names can be used instead of numbers in any Error statements and On Error statements that refer to the error.

For example:

```
Const ooBounds = 677
' A specific out-of-bounds error
' ...
Error ooBounds
' ...
On Error ooBounds ...
```

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 number of 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 these statements executes.

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.

Error handling routines outside procedures

LotusScript can handle an error in the procedure where it occurs or 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 syntax is:

On Error *errNumber* **GoTo 0**

This says that the error numbered *errNumber* is not handled in the current procedure.

This example shows that 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
```

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.

Ending the error handling routine

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.

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.

Example 1

This extended example shows 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.

The 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
```

Example 2

In this 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
```

On Error Resume Next

On Error Resume Next specifies that program execution continues with the next statement after the statement that generates the error, instead of specifying an error-handling routine that executes when the error occurs.

```
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.
```

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.

Resuming execution in a calling procedure

On Error Resume Next has a special meaning in handling an error that occurred in a lower-level procedure. 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 re-execute the line where the error occurs, even if that line is in a lower-level procedure.

For example:

```
' 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 ProcPo   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).
```

Multiple On Error statements

Handling individual errors

An On Error statement refers to only one error-handling routine. For more than one, you include multiple On Error statements, one for each error-handling routine in your script.

For example:

You include a Print statement in a script that can generate a division-by-zero error. To handle a division-by-zero error, you could include an 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
```

To ensure that all other errors are handled without terminating script execution, include an On Error statement that doesn't specify a particular error.

This example shows a script that specifically manages file-open failures and division-by-zero errors. All others are included in a general On Error statement.

```
%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
' number 3 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
```

This example of a call to GetLine shows how the sub works. For all errors other than file-open failures errors and division-by-zero errors, the error-handling routine at Leave displays a message box and returns from the sub GetLine.

```
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
```

However, if the user enters 99999 instead of 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 error-handling routine (or none) in effect at any given time for any particular error 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.

In this example, 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
```

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.

Chapter 8
User-Defined Data Types and Classes

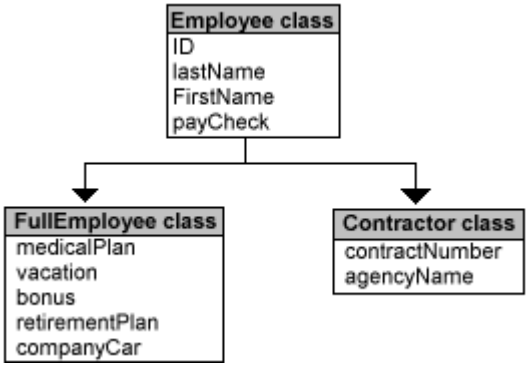
This chapter describes two kinds of custom data structures that you can define in LotusScript. Each can hold data of different types in a single data structure.

Overview of user-defined data types and classes

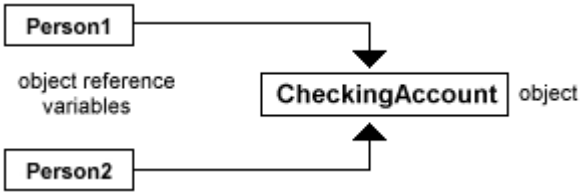
User-defined classes are common to object-oriented programming and are used to represent objects whose data can be protected, initialized, and accessed by a specific set of procedures.

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

User-defined data types are a common feature in BASIC programming and are used to support database, file read/write, and print operations. 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. A variable of a user-defined data type holds actual data, not a pointer to that data.

The syntax is :

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

member variable declarations

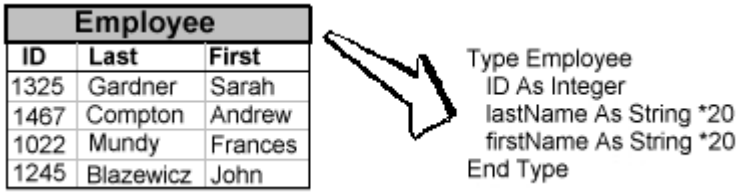
End Type

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.

User-defined data types cannot contain procedures (properties and methods) and cannot be extended.

This example 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:



Declaring a variable of a user-defined data type

After you define a user-defined data type, you can declare a member variable.

For example:

```
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 member variables.

For example:

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

Referring to member variables

Use dot notation (*object.memberVariable*) to refer to member variables. Use an assignment statement to assign values.

```
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. You can use data space efficiently by declaring members with the highest boundary first and those with the lowest boundary last. Wasted space in the definition becomes wasted space in every variable of that user-defined data type.

This example shows a well-aligned variable:

```
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.

This example, 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
```

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:

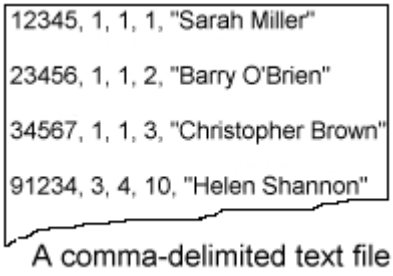
  ```
  Dim x As Long
  Dim y As String
  ```

  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
  employee As String             ' Employee name
  theSection As Integer          ' Car parking section
  theSpace As Integer            ' Designated parking space
  theFloor As Integer            ' Car parking level


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 Boolean

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%).employee, _
     arrayOfRecs(countrec%).theSection, _
     arrayOfRecs(countrec%).theSpace, _
     arrayOfRecs(countrec%).theFloor
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.
```

User-defined 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.

You can create two types of LotusScript classes:

- Base class

  Defines common member variables, properties, and methods that can be inherited by other classes.

- 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.

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 work only with the member variables, properties, and methods of the class, 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 software.

Base classes

The syntax is:

[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

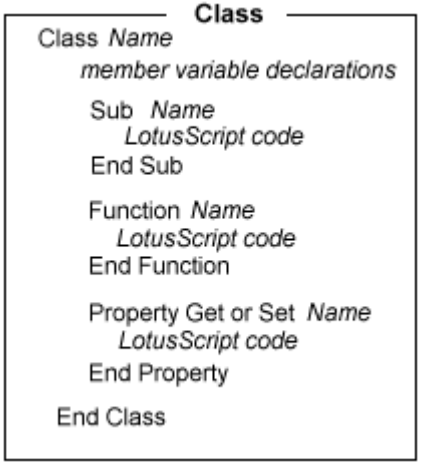
Properties and methods are tied to their class and can be used only with an object belonging to that class. You define properties and methods inside the Class statement.

- Property

  A pair of functions used to manipulate protected variables, that is, Private members of a user-defined class to which the application has no direct access.

- Method

  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 create two special subs. They are always Public; you cannot declare them as Private.

- Sub New

  A sub that LotusScript executes automatically when an object is created. Sub New executes 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. You create Sub New by defining a sub named New and the parameters to initialize the newly created object. A class can have only one Sub New.

- Sub Delete

  A sub that LotusScript executes automatically when the object for which it is defined is deleted. You create a Sub Delete by defining a sub named Delete, without specifying parameters. A class can have only one Sub Delete.

You can use these subs as events in your scripts. 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 margins, and that uses Sub Delete to terminate the print job.

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.
```

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). 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 users of the class.

Private class members

Class scope is everything within the *Class...End Class* statement. Class members are accessible to all of the properties and methods of the class.

You can refer to an individual member of a class by using its member name.

This example prints the value in a member variable called employeeName$:

```
Print employeeName$
```

Within a property or method, you can use the keyword Me to access the class 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:

- 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.

This class definition example 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
```

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 Sub New to initialize member variables in a class, 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.

If the class is derived from a base class, Sub New of the base class is also called. The Sub New of the subclass is called first, then the Sub New of the parent class is called.

Public class members

Outside a class's scope, you can access only its Public members. You use dot notation to refer to Public class members.

In this 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 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.

This 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 do, the expression evaluates to True if they refer to the same object, or if both have the value NOTHING, and evaluates to False if they don't.

This example tests 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.

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 this 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.

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 this 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

A derived class is created from a previously defined class.

The syntax is:

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

classBody

End Class

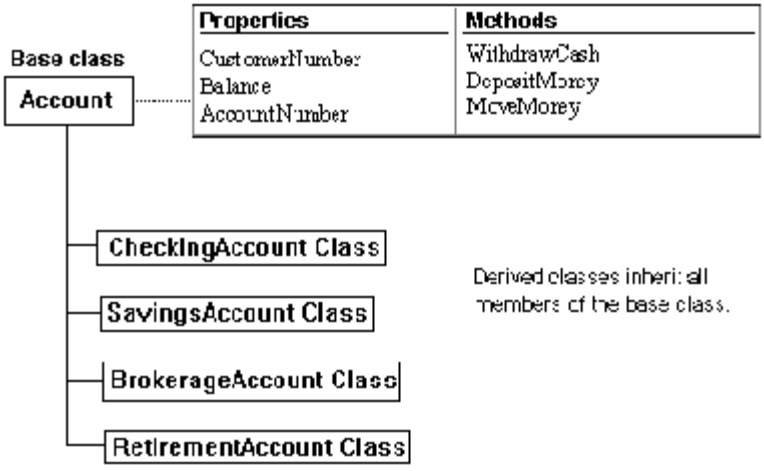
Element	*Description*
Public, Private	Public makes the derived class accessible outside the module in which it is defined. Private (default) makes the derived class accessible only within the module in which it is defined.
className	The name of the derived class.
baseClass	The name of the base class 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 MyClass ' Object x has members
                     ' a%, b%, c%, and d%.
x.c% = 12
x.d% = 35
'...
```

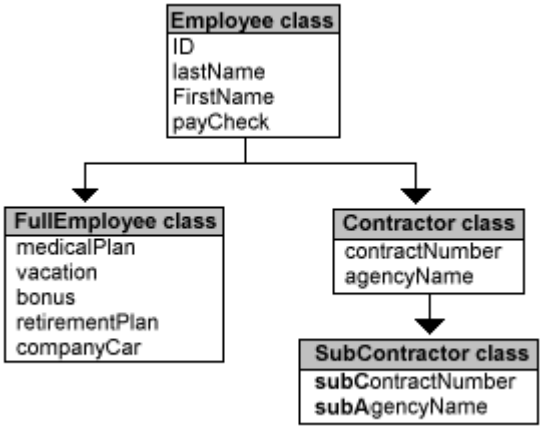
Usually you use a derived class when an existing class provides members that the new class can use, or when you want to extend or embellish existing class properties and methods. This is called inheritance: the new class inherits, and has direct access to, all Public and Private members of the existing base class.

For example, 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 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.

Property and method overriding

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.

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 Sub New definition of its base class. Sub New for a derived class must provide the base class Sub New with its expected parameters.

The parameter list for the Sub New of the base class 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 Sub New of the base class if the arguments for the derived class Sub New and the base class Sub New are the same.

The syntax is:

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.

continued

Element	*Description*
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.

This derived class 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 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.

This example shows 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 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.

This example 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.

This example creates both an array and a list of objects of class Fruit:

```
' Declare an array of references to base class: a Fruit
Basket.

Dim Basket( 1 to 4 ) 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)

' Declare a list of references to base class: a Fruit Bucket.

Dim Bucket List As Fruit

Set Bucket("1") = New Apple(0.86, "Green", "Macintosh", 24)

Set Bucket("2") = New Apple(0.98, "Red", "Delicious",33)

Set Bucket("3") = New Banana(0.32, "Yellow")

Set Bucket("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

' Prepare all of the fruit in the Bucket.

ForAll FruityThing in Bucket
```

```
    FruityThing.Prepare   ' Call each object's Prepare sub.
End ForAll
```

Working with object reference variables

You use an object reference variable to create, manage, and delete objects. 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 is not the object itself but a 4-byte pointer to the object data, called an object reference. LotusScript uses this pointer to access the object data.

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 and assign objects using the LotusScript New keyword.

- 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 this 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

The Set statement is an assignment statement used only to assign values (object references) to object reference variables. You cannot use any other a 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.

For 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
```

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
```

Chapter 9
Managing Flow in Scripts

The flow of execution of a script generally follows the sequence of statements in the script. This chapter describes the behavior of particular statements that alter the flow of execution.

Flow of execution

Flow control statements

The flow control statements that alter the flow of execution fall into several functional groups:

- The block statements If...Then...Else, If...Then...ElseIf, and Select Case

  These specify executing a group of subsidiary statements, depending on specified conditions.

- The branching statements GoTo, If...GoTo...Else, On...GoTo, GoSub, On...GoSub, and Return

  These specify continuing execution at some other point in the script, possibly depending on specified conditions.

- The iterative block statements Do, For, ForAll, and While

  These specify repeating a group of subsidiary statements some number of times, or while or until some specified condition is satisfied.

- The early termination statements End and Exit

  These 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 remaining sections in this chapter discuss these statements in the order listed above.

There is no built-in limit on the level or type of nesting of these statements. For example, a Do statement may contain another Do statement that contains a third Do statement, or a Do statement may contain a For statement that contains another Do statement.

Comments and the compiler directive

Comments are not executed. 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.

Note %if is not directly supported in all products (for example, Notes). You cannot enter %if directly in the IDE. You must enter this directive in a file and insert the file in the IDE with the %Include directive.

Declarations

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 produce 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.

Definition statements

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.

Errors

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.

Statement labels

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.

Block statements

Selecting one or the other with the If...Then...Else statement

The block statement If...Then...Else 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.

The syntaxes are:

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.

Note Newline does not mean either chr(10) or chr(13) on all platforms. Newline is the character or sequence of characters that is used to mark the end of a line. This may be chr(10), or chr(13), but it may also be something else, because the actual value of newline depends on the platform.

This example shows a Then clause consisting of the single statement Exit Do. There is no Else clause. The script 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 in this chapter.

To include more than one statement in the Then clause, separate the statements by the colon (:) statement separator.

```
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.

```
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.

This example shows 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.

Specifying multiple test conditions with the If...Then...ElseIf statement

The block statement If...Then...ElseIf 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, and the contents of the If clause, the ElseIf clauses, and the Else clause must be written in the correct order.

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.

An If...Then...ElseIf statement 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, using GoTo, GoSub, and labels. All of these statements must be part of a procedure.

This example uses If..Then...ElseIf to determine whether a Timer value represents Morning, Afternoon, or Evening.

```
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
```

If you change the order of the contents of the If clause and the ElseIf clause, you can get a wrong result. For example, a Timer() value of 38017, 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
```

This example uses If...Then...ElseIf statements to demonstrate changing a user-supplied whole number to 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. There are three nesting levels of If...Then...ElseIf. Each statement needs its own End If phrase. An End If phrase ends the innermost statement running.

```
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. If the main logic of this script were made into the contents of a procedure, it could be rewritten more simply.

Making a choice with the Select Case statement

The block statement Select Case 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.

This example adds a suffix to a whole number to turn it into an ordinal number. The script defines and calls the function SetOrd, which 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:")))
```

The last line of the example is the only executable code outside of the function SetOrd and 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.

Branching statements

Transferring control with the GoTo statement

The branching statement GoTo transfers control unconditionally.

The syntax is:

GoTo *label*

When this statement is executed, LotusScript transfers control to the statement labeled label. GoTo and its target must be in the same procedure. The flow of control is determined at run time.

This 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.

This 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
```

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.

Using the If...GoTo...Else statement to transfer unconditionally

The branching statement If...GoTo...Else is 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 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%
```

Conditional control transfer with the On...GoTo statement

The branching statement On...GoTo transfers control conditionally.

The syntax is:

On *expression* **GoTo** *label,* [*, label*]...

The statement 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.

On...GoTo and its target labeled statements must be in the same procedure. The 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 prints 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
```

Transferring control within the same procedure with the GoSub, On...GoSub, and Return statements

The branching statements 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 they can't take arguments, don't establish a separate scope, and aren't available from other procedures or scripts. It is more common and useful to write the statements as an ordinary sub.

The syntax is:

GoSub *label*

On *expression* **GoSub** *label* [, *label*]...

Return

The statement GoSub label transfers control to the statement labeled label and executes statements beginning at label, continuing until one of the following occurs:

- A Return statement is encountered.

  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.

  Execution of the script ends (End statement), or execution of the enclosing procedure ends (one of the other statements).

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, ... transfers control similarly to GoSub label, 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 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.

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.

These statements differ from the GoTo and On...GoTo statements, which transfer control without establishing a point of return.

This example uses On...GoSub to run 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()
```

Iterative statements

Do and Do...While loops

The iterative block statement Do executes a block of statements repeatedly while a given condition is true, or until it becomes true. The block of statements executes infinitely only if the condition for termination is never satisfied.

The three kinds of Do statements 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 is:

- **Do...Loop**

  There is no condition.

- **Do While** *condition*...**Loop** or **Do Until** *condition*...**Loop**

  The condition is evaluated before each iteration.

- **Do...Loop While** *condition* or **Do...Loop Until** *condition*

  The condition is evaluated after each iteration.

This example illustrates the first form of Do statement. The Do loop 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.

```
doCount% = 0
Do
   doCount% = doCount% + 1
   If doCount% >= 1000 Then Exit Do
Loop
```

In this 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 or a Do Until condition statement is tested before the first iteration, whereas the condition in a Do...Loop While 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 or Do Until...Loop statement may not be executed at all.
- The body of a Do...Loop While or 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 this 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)
```

For...Next loops

The iterative block statement For executes a block of statements a specified number of times.

The syntax is:

For *countVar* = *first* **To** *last* [**Step** *increment*]

[*statements*]

Next [*countVar* [, *countVar*]...]

This example shows a For statement that does not use the Step or Next optional items.

```
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.

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.) You should always declare your loop variable: additional computing resources are necessary to convert the value to a Variant in a tight loop.

For example:

```
' If the variable iV was not 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 this 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.

Other features of this algorithm are:

- To print the coefficients only for n, rather than for every integer up to n, 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, 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 can 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. 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 show some 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 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 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 i% = 1 To 3
   For i% = 1 To 3
   Next
Next
' Output:
' Error 52: FOR count variable already in use: I
```

ForAll loops for lists and arrays

The iterative block statement ForAll 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 but 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
   Print ListTag(idAttrib)": " idAttrib
    ' For each item in persStats, print its tag and value.
End ForAll
' Output:
' Name: Ian
' Age: 36
' Home state: MD
```

This example shows 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

You cannot 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
```

You cannot 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 container can be an array or a list.)

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

This example shows how a ForAll statement references the current value of each element in the container array or list. 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 "User-Defined Data Types and Classes."

In this 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. Although you can use the Erase statement on the container or its elements; or use the ReDim statement on an array, it is not recommended, as the results are unpredictable.

Similarly, it is possible to transfer control from outside a ForAll statement to a labeled statement inside. 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, 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)
```

Using the While statement

The iterative block statement While 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.

Early termination statements

Stopping procedure execution early using the 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 software application 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.

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.

```
' 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
```

Using the Exit statement for early procedure termination

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.

When you use Exit with a Do, For, or ForAll statement, execution continues at the first statement following the end of the block statement.

For example:

```
' 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 procedure, execution continues as it would following a normal return from the procedure.

This example incorporates the Do statement from the preceding example within a sub. 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.

```
' 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
```

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
```

Calls to TwoVerge within Print statements 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
```

Chapter 10
Managing Asynchronous Web Agents in Domino

This chapter describes how to use multiple threads and synchronization to manage HTTP agents with Domino.

Introduction to multithreading and synchronization in LotusScript

LotusScript is thread safe; multiple LotusScript Web agents can run concurrently within the Domino server.

A thread is an instance of LotusScript, in this case an agent. All threads execute in the same memory space. LotusScript threads have no protection against updates on global variables or contention on the various internal data structures. By running multiple agents, you synchronize instances of LotusScript running against each other.

LotusScript agents run as separate threads in the same HTTP process. A process is a collection of one or more threads executing a single application.

Context switching is the act of saving the current state (hardware and software) and switching to another thread or process by restoring its state.

A time slice is the amount of time given to a thread or process to execute before switching context to the next thread or process.

A thread is blocked if a necessary resource is unavailable.

An atomic update is one in which another thread observing the update always sees a complete update and cannot interfere.

Advantages of thread-safe agents

Threading offers the following advantages over serial agents:

- Computer resources on Domino Web servers are used more efficiently.
- The system allows concurrent use by many people.
- The client/server programming model can be used.

Domino Release 4.5.1 and later supports multiple Web agents, allowing each LotusScript agent to run in a separate thread in the same process. In Domino, if multiple users activate Web agents simultaneously and the server is not thread-enabled, the agents will be serialized. To enable

Domino synchronized agents, see the section “Running asynchronous agents on the Domino server.”

Agents run serially

In this example, User A's agent had control over the server until it completed. User B saw no activity until Agent 1 was finished.

Time	*Operation*	*Comments*
0		User A activates Agent 1.
1	Compute	Agent 1 starts. User B activates Agent 2.
2	Compute	Agent 1 running.
3	Compute	Agent 1 running.
4	Compute	Agent 1 running.
5	Print	Agent 1 ends.
6	Compute	Start User B's Agent 2.
7	Print	Agent 2 ends.

Threaded agents

In this example, User B sees results sooner. User A sees response later, but the time difference is not noticeable.

Time	*Thread1*	*Thread2*	*Comments*
0			User A activates Agent 1.
1	Compute		Agent 1 starts. User B activates Agent B.
2	Compute		Agent 1 running.
3		Compute	Agent 1 swapped out. Agent 2 starts.
4		Print	Agent 2 ends.
5	Compute		Agent 1 running.
6	Compute		Agent 1 running.
7	Print		Agent 1 ends.

Synchronization functions

LotusScript 4.0 (in Domino 5.0) includes a new set of primitives to allow LotusScript agents to synchronize with one another:

CreateLock — finds the lock ID associated with Name. If none exists, the Lock ID is created.

DestroyLock — removes the current link to the lock specified. If the number of links is zero, the lock is destroyed.

CodeLock — acquires the lock specified by ID. If the lock is already held by another agent, the thread stalls until the lock becomes available.

CodeUnlock — releases the lock, making it available for the next agent requesting it.

CodeLockCheck — returns the number of agents waiting for the the specified lock, plus 1.

Sleep — causes a script to pause for at least the number of seconds specified.

How synchronization works

Synchronization involves sharing a single CPU among multiple tasks (or threads) in a way designed to minimize the time required to switch threads. On a thread-enabled server, agents take turns performing their tasks, which saves time and gives the illusion of the tasks occuring at the same time.

This sample agent, *Comm1*, run at the same time as an identical one named *Comm2,* illustrates how you can use code locks to synchronize agent execution.

Run concurrently, these agents create a named lock called "Update," then:

- If Comm1 is the first agent to start execution, it gets the lock; the second (Comm2) must wait for Comm1 to release it before starting. Comm1 outputs a message that it has the lock and provides the reference count.
- As soon as Comm1 is done with the lock, it releases it and Comm2 takes it. Comm1 tries to obtain the lock again, but now must wait until Comm2 is done with it. Comm2 outputs a message that it now has the lock and reports the reference count again.
- As soon as Comm2 is done with the lock, it releases it and Comm1 takes it.

The process repeats for the duration of "For loop," in this case, 5 iterations.

```
'Comm1:

Option   Public

' Remove the following line if you do not have a
' resource library defined.
Use "ThreadsLib"

Sub Initialize
   Dim lockName As String
   Dim lockID As Integer, refcnt As Integer
   Dim gotLock As Variant,  releaseLock As Variant, _
    deleteLock As Variant

   On Error Goto syn_error

   ' Provide some unique name here to distinguish the agents.
   ID = "Comm1  tuvwx:5706 "
   Msgbox  ID & "Started"

   lockName = "Update"
   On Error Goto syn_error

   ' Create the lock
   lockID = Createlock(uName)
   If (lockID <> -1) Then
      Msgbox ID & "Created lock: " & lockID
   End If

   ' Put agent to sleep for a second.
   ' This gives the second agent time to start.
   Sleep 1

   For x = 1 To 5

      ' Attempt to get the lock and report the outcome
      ' as well as the reference count
      gotLock = CodeLock(lockID)
      If (gotLock) Then
         Msgbox ID & " Got lock: " & lockID & " - at: " & _
           Now()
         refcnt = Codelockcheck(lockID)
         Msgbox ID & "      Reference count is " & refcnt

         ' Do some meaningful work here, or just sleep
         Sleep 1
      Else
         Msgbox ID & "Failed to get lock"
      End If

      ' Release the lock so the other agent can get it.
      releaseLock = Codeunlock(lockID)
```

```
      If (releaseLock) Then
         Msgbox ID & " Releasing lock: " & lockID & _
           " - at: " & Now()

         ' Sleep here allows the other agents to obtain
         ' the lock before this agent has a chance to.
         Sleep 1
      Else
         Msgbox ID & "Failed to release lock"
      End If

   Next

   ' When we are finished, destroy this agent's reference
   ' to the lock
   deleteLock = Destroylock(lockID)
   If (deleteLock ) Then
      Msgbox ID & "Destroyed lock " & lockID
   Else
      Msgbox ID & "Failed to destroy lock"
   End If

   Msgbox  ID & "Done "
   Exit Sub

syn_error:
   errormsg  = "  * * Error: " & Err & " - " &  Error() & _
    " in " & ID & " at " & Erl()
   Msgbox errormsg
   Resume Next

End Sub
```

A sample output of Comm1 (with an ID of tuvwx:5706) and Comm2 (with an ID of uvwxy:5742) running concurrently as agents through the Domino Web server would look something like the following:

Note Your results will not be identical; due to the nature of asynchronous system locks, you can never predict when specific events will occur.

```
Addin: Agent message box:    Comm1 tuvwx:5706 Started
Addin: Agent message box:    Comm1  tuvwx:5706 Created lock: 0
Addin: Agent message box:    Comm2  uvwxy:5742 Started
Addin: Agent message box:    Comm2  uvwxy:5742 Created lock: 0
Addin: Agent message box:    Comm1  tuvwx:5706  Got lock: 0 - at:
2/10/99 1:57:06 PM
Addin: Agent message box:    Comm1  tuvwx:5706      Reference count is  1
Addin: Agent message box:    Comm2  uvwxy:5742  Got lock: 0 - at:
2/10/99 1:57:07 PM
Addin: Agent message box:    Comm2  uvwxy:5742      Reference count is  1
Addin: Agent message box:    Comm1  tuvwx:5706  Releasing update_lock:
```

```
0 - at:    2/10/99 1:57:07 PM
Addin: Agent message box:    Comm2  uvwxy:5742  Releasing update_lock:
0 - at:    2/10/99 1:57:08 PM
Addin: Agent message box:    Comm1  tuvwx:5706  Got lock: 0 - at:
2/10/99 1:57:08 PM
Addin: Agent message box:    Comm1  tuvwx:5706       Reference count is  1
Addin: Agent message box:    Comm2  uvwxy:5742  Got lock: 0 - at:
2/10/99 1:57:09 PM
Addin: Agent message box:    Comm2  uvwxy:5742       Reference count is  1
Addin: Agent message box:    Comm1  tuvwx:5706  Releasing lock: 0 - at:
 2/10/99 1:57:09 PM
Addin: Agent message box:    Comm2  uvwxy:5742  Releasing lock: 0 - at:
 2/10/99 1:57:10 PM
Addin: Agent message box:    Comm1  tuvwx:5706  Got lock: 0 - at: 2/10/99
1:57:10 PM
Addin: Agent message box:    Comm1  tuvwx:5706       Reference count is  1
Addin: Agent message box:    Comm2  uvwxy:5742  Got lock: 0 - at:
2/10/99 1:57:12 PM
Addin: Agent message box:    Comm2  uvwxy:5742       Reference count is  1
Addin: Agent message box:    Comm1  tuvwx:5706  Releasing lock: 0 - at:
2/10/99 1:57:12 PM
Addin: Agent message box:    Comm2  uvwxy:5742  Releasing lock: 0 - at:
2/10/99 1:57:13 PM
Addin: Agent message box:    Comm1  tuvwx:5706  Got lock: 0 - at: 2/10/99
1:57:13 PM
Addin: Agent message box:    Comm1  tuvwx:5706       Reference count is  1
Addin: Agent message box:    Comm2  uvwxy:5742  Got lock: 0 - at:
2/10/99 1:57:14 PM
Addin: Agent message box:    Comm2  uvwxy:5742       Reference count is  1
Addin: Agent message box:    Comm1  tuvwx:5706  Releasing lock: 0 - at:
2/10/99 1:57:14 PM
Addin: Agent message box:    Comm2  uvwxy:5742  Releasing lock: 0 - at:
2/10/99 1:57:15 PM
Addin: Agent message box:     Comm1  tuvwx:5706  Got lock: 0 - at:
2/10/99 1:57:15 PM
Addin: Agent message box:    Comm1  tuvwx:5706       Reference count is  1
Addin: Agent message box:    Comm2  uvwxy:5742  Got lock: 0 - at:
2/10/99 1:57:16 PM
Addin: Agent message box:    Comm2  uvwxy:5742       Reference count is  1
Addin: Agent message box:    Comm1  tuvwx:5706  Releasing lock: 0 - at:
2/10/99 1:57:16 PM
Addin: Agent message box:    Comm2  uvwxy:5742  Releasing lock: 0 - at:
 2/10/99 1:57:18 PM
Addin: Agent message box:    Comm1  tuvwx:5706 Destroyed lock 0
Addin: Agent message box:    Comm1  tuvwx:5706 Done
```

```
Addin: Agent message box:    Comm2  uvwxy:5742 Destroyed lock 0
Addin: Agent message box:    Comm2  uvwxy:5742 Done
```

These primitives are used only for communication between instances of cooperating LotusScript agents within a single process. They are designed specifically for asynchronous Web agents.

Supported platforms include Win32, OS/2, UNIX (Solaris, HPUX, AIX), and Alpha-NT.

Running asynchronous agents on the Domino server

To enable multiprocessing agents on the server:

1. Add the following line to your NOTES.INI file:

   ```
   DominoAsynchronizeAgents=1
   ```

2. Restart your HTTP server.

Note Enabling multiprocessing is not the same as increasing the number of agent managers.

Thread-safe LSX, C/C++ code

Notes is thread-safe. To write multithreaded agents, you must make sure your LSX or C/C++ code is thread-safe.

Thread-safe code means one of the following:

- The code shares no resources with other execution threads.
- The code shares resources with other execution threads but locks prevent concurrent access.

To design thread-safe code:

- Use only stack-allocated and dynamically-allocated memory in C or C++, such as, function/method local or calloc/malloc/new.
- Do not pass pointers between threads.
- If you must use static declarations, module level variables, or pass pointers, use locks.

Thread-specific bugs

Threading problems are usually non-deterministic.

Common threading problems include:

- Lost updates
- Partial updates
- Deadlock

- Thread calls non-thread-safe code

Creating and destroying locks

To create locks, use the command:

LockID=CreateLock(*LockName* **as String)**

This command creates a link to the specified lock and returns the lock ID used by other lock primitives. It creates a lock if one doesn't exist.

To remove locks, use the command

DestroyLock (*LockID* **as Integer)**

This command removes the current link to the lock specified and destroys the lock if no links remain.

Chapter 11
Beyond Core LotusScript

This chapter discusses the role that LotusScript plays with Lotus products, your operating environment, other programs, and interactive user applications.

Lotus software environments

Lotus software provides the environment in which you create, debug, and run LotusScript modules. Lotus software applications that work with LotusScript supply their own application programming interface (API), which lets you use product functionality and create and manipulate product objects from within a LotusScript program. A Lotus software API is effectively an extension to the LotusScript language that is available when you are running that product.

Determining which product file is being used

On Windows, and some other platforms, you can use command-line arguments (in the Windows 95 Open dialog, for example) to start programs and open product files.

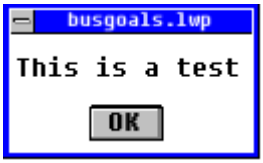
The Command function returns the command-line arguments used to start the Lotus software application where LotusScript was invoked. You can use it to get the name of the product file. For example, you may use the file name to identify which product file is currently running, or to provide input for messages to the user.

For example, if 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 then 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$, messageTitle$
```



Product classes and objects

Lotus software you use with LotusScript provides a number of predefined classes. Product objects (instances of product classes) are like user-defined objects (instances of user-defined classes) but can have their own existence apart from the scripts in which you manipulate them. For example, you can 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.

For information about user-defined classes, see "User-Defined Data Types and Classes."

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, see the Lotus software documentation.

Creating objects

The product automatically creates some objects (cells in a spreadsheet for example). You can use the product user interface to create objects, and 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. The syntax is usually:

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***])]**

You can use a method to create the object. For example, in Lotus Notes Release 4, you use the NotesDatabase Create method to create a new .NSF file.

You can also use a container method to create objects in scripts. A container method applies to the object that contains the object you are creating. For example, Freelance Graphics for Windows provides container methods for creating objects.

Referring to objects

To refer in a script to an object that already exists, you can usually use the name that the product or user gave to the object. You 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, such as:

Dim *objRef* **As** *prodClass*

and bind it to the product object. For example:

Set *objRef* **= Bind [***prodClass***] [(***objName***])]**

The product can supply 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 software 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 "LotusScript Language Reference," and check 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 "LotusScript Language Reference."

Properties are object attributes. Like variables, properties have values. You can get and set a property value. However, some properties you can only get, and some you can only set.

Methods are object operations. You call methods to perform actions on classes.

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 can set the value of the Title property in the Click event script:

```
db.Title = "HQEVB Group Discussion"
```

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.

For example, a db object is an instance of the Notes/Domino NotesDatabase class. It has a number of properties, including FileName, FilePath, and Title.

The value of the Title property is a string specifying the title of the database. In a script, you can get and set the value of Title. You can only get the values of FileName and FilePath, which specify the location the database in the file system.

Deleting objects

You can sometimes use the Delete statement to delete a product object or the object reference variable. 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. Some products supply methods to remove actual objects. For example, in Notes, 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.

For example, in Freelance Graphics an Application object contains an instance of the Documents collection class. Each Document object contains an instance of the Pages collection class and each page object contains an instance of the ObjectCollection class. The ObjectCollection object can include text boxes, charts, tables, and other objects belonging to classes derived from the DrawObject class.

For more info on deriving classes (also known as class inheritance), see "Derived classes" in "User-defined Data Types and Classes."

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
```

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. While each individual Lotus software application provides its own user interface for interacting with scripts, LotusScript supplies some fundamental tools that you can use with any Lotus software.

- InputBox function

  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, CBool, CByte, 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.

- Print statement or the MessageBox function or statement

  The Print statement displays a message in the current output window, which varies depending on the Lotus software application in which you are working. MessageBox displays a message box, which contains an optional title, the message, an optional icon, and one or more command buttons.

  If you want to display a message, 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.

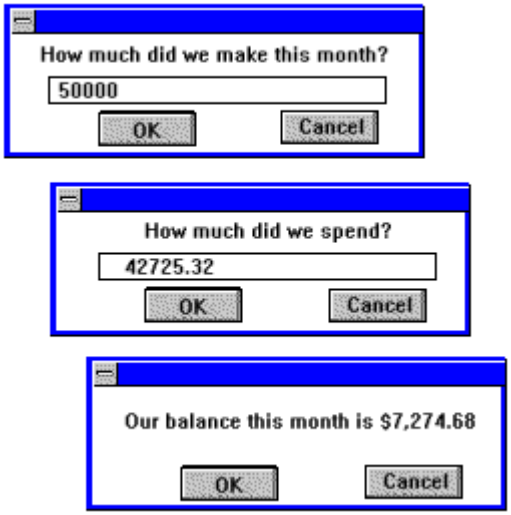
  To offer the user two or more options, use the MessageBox function and include two or more command buttons. For example, you can include OK and Cancel buttons. You can use an If statement or Case statement to respond to the user's response accordingly.

This 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
```

The two input boxes with sample entries and the resulting message box are:

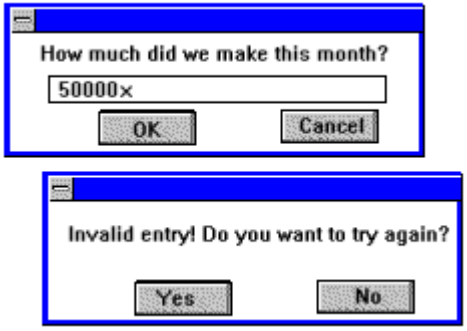


If the user enters a string that the CCur function cannot convert to Currency, an error condition occurs. You can use an On Error statement to branch to an error-handling routine in such a case.

This expanded version of the example uses the MessageBox function to ask the user whether to try again. 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 the chapter "Error Processing."

MsgBox on Notes server context

When you run LotusScript agents on the Notes server, the commands MsgBox, Inputbox, and Print will be re-directed to the status bar and will be put into the agents log.

For HTTP servers, these commands redirect the output to the browser. You can create HTML pages dynamically using these commands to serve to any browser.

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) to incorporate functionality and data from other Windows applications into your LotusScript applications.

Functions and statements for working with other programs

LotusScript provides several functions and statements that you can use to work with other programs and with the operating system.

Function/Statement	*Purpose*
Shell function	Starts another program
Shellid function	Starts another program and returns its task ID.
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. The OS/2 and UNIX platforms and the Macintosh support some. Also, different client products may choose not to support certain functions. For example, Notes does not support SendKeys and Yield. Additionally, Yield is only useful in a Win 16 environment. For more information, see Appendix B, "Platform Differences."

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.

There are 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. It then yields control to Windows for 10 seconds so the user can 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 10 seconds, an ActivateApp statement insures that Notepad has the focus (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+C 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 executes. An ActivateApp statement shifts focus to Notepad, in case the user moved the focus to another window. A SendKeys statement sends ALT+F4 to Notepad, which closes Notepad. The sub then 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 software application from which you are running LotusScript.

For example, Shapeware Visio 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:

```
For i% = 1 To Ubound(manager$)
      Print manager$(i%)
Next
```

For information about Visio classes, including their properties and methods, see the Visio documentation.

Calling external C language functions

LotusScript allows you to call external C language functions. You implement external C functions inside a named library module that generally contains several C functions. With Windows, this is a Dynamic Link Library (DLL). All Windows users have access to the libraries in the Windows application programming interface (API).

Note The C functions that are in DLLs/shared libraries must be exported. Different platforms will have different rules and ways for exporting them.

To work with C functions, you need documentation that explains their input and output parameters, return values, and what operations they 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.

To call C functions contained in an external library module from LotusScript, use a Declare statement for external C calls for each function you want to call. To avoid declaring external library functions in multiple scripts, use Declare Public statements in a module which remains loaded.

The following table shows the convention that function calls from LotusScript must use to external functions.

Platform	*Calling convention*
Windows 3.1	Pascal
Windows 95, Windows NT	STDCALL
OS/2	_System
UNIX	CDECL
Macintosh	CDECL

If you are using a C++ compiler, the name of any function becomes corrupted. Use the extern "C" {. . .} construct to keep the exact name intact.

If you are using Windows 95 or Windows NT, the name of an exported DLL function is case sensitive, although LotusScript automatically converts the name to uppercase in the Declare statement. To successfully call an exported DLL, use the Alias clause in the Declare statement to specify the function name with correct capitalization. LotusScript leaves the alias alone.

Note The "_" is reserved for Notes-specific DLLs. This is a change effective as of Notes 4.5.1. If you attempt to load a DLL in Notes 4.51 or greater using LotusScript, and the name of the DLL is preceded by an underscore, you will receive the error "Error in loading DLL."

Example

```
' The following statements declare an exported DLL with an
'alias (preserving case sensitivity), and then call that
'function.
Declare Function DirDLL Lib "C:\myxports.dll" _
   Alias "_HelpOut" (I1 As Long, I2 As Long)
DirDLL(5, 10)
```

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 User32 library in the Windows API.

GetActiveWindow takes no parameters and returns the handle (an integer) of the active window (the window with focus).

```
Declare Function GetActiveWindow Lib "User32" () As Long
```

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 "User32" Alias "SetWindowTextA"
_
    (ByVal hWnd As Long, ByVal lpString As String)
```

Note Be aware that you are actually calling a C function which needs to be supplied by you. This may cause your script to be platform-dependent.

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 arguments by reference

When an argument is passed by reference, the C function receives a 4-byte pointer to the value area.

In some cases, the actual stack argument is changed to a publicly readable structure. In all cases, the data may be changed by the called function, and the changed value is reflected in LotusScript variables and in the properties of product objects. For such properties, this change occurs directly after the call has returned.

Data type	*How it is passed to a C function*
String	A 4-byte pointer to the string in the platform-native character set format
Product object (including a collection)	A 4-byte product object handle
Array	A 4-byte pointer to the array stored in the LotusScript internal array format
Type	A 4-byte pointer to the data in the type instance (This may include strings as elements)
User-defined object	A 4-byte pointer to the data in the object (this data may include strings, arrays, lists, product objects, etc., as elements)

Note Lists cannot be passed by reference. They also cannot be passed by value. Using a list as an argument produces a run-time error.

Passing arguments by value

When an argument is passed by value, the C function receives a copy of the actual value of the argument.

- To specify that the argument should always be passed by value, use the keyword ByVal preceding the parameter declaration for that argument in the Declare statement for the C function.
- To specify that the argument should be passed by value in a particular call to the function, use parentheses around the argument in the call.

The C routine cannot change this value, even if the C routine defines the argument as passed by reference.

Data type	*Keyword*	*How it is passed to a C function*
Boolean		A 2-byte Integer, of value 0 or -1, is pushed on the call stack.
Byte		A 1-byte Integer value is pushed on the call stack.
Integer		A 2-byte Integer value is pushed on the call stack.
Long		A 4-byte Long value is pushed on the call stack.
Single		A 4-byte Single value is pushed on the call stack.
Double		An 8-byte Double value is pushed on the call stack.
Currency		An 8-byte value, in the LotusScript internal Currency format, is pushed on the call stack.

continued

Data type	*Keyword*	*How it is passed to a C function*
String		A 4-byte pointer to the characters is pushed on the call stack. The C function should not write to memory beyond the end of the string. If the call is made with a variable, changes to the string by the C function are reflected in the variable. This is not true for a string argument to a LotusScript function declared as ByVal.
Variant		A 16-byte structure, in the LotusScript format for Variants, is pushed on the call stack.
Product object		A 4-byte product object handle is pushed on the call stack.
Any		The number of bytes of data in the argument is pushed on the call stack. For example, the argument contains a Long value, then the called function receives 4 bytes. The function may receive a different number of bytes at run time.

No other data types — arrays, lists, fixed-length strings, types, classes, or voids — can be passed by value. It is a run-time error to use these types as arguments.

Any of the data types that can be passed by value can also be passed by reference.

The argument ByVal &0 specifies a null pointer to a C function, when the argument is declared as Any.

Example

```
Declare Sub SemiCopy Lib "mylib.dll" _
   (valPtr As Long, ByVal iVal As Long)
Dim vTestA As Long, vTestB As Long
vTestA = 1
vTestB = 2

SemiCopy vTestA, vTestB

' The C function named SemiCopy receives a 4-byte pointer to a
' 2-byte integer containing the value of vTestA, and a 2-byte
' integer containing the value of vTestB.
' Since vTestA is passed by reference, SemiCopy can
dereference
' the 4-byte pointer and assign any 2-byte integer to that
' location. When control returns to LotusScript, vTestA
' contains the modified value. Since vTestB was passed by
' value, any changes made by the C function are not reflected
' in vTestB after the function call.
```

Passing strings

When you pass a string by reference, LotusScript passes a 4-byte pointer to a copy of the string in an internal buffer allocated in memory. The C function cannot safely modify the contents of this buffer unless the function is written specifically for LotusScript.

When a string is passed by value, LotusScript passes a 4-byte pointer to a null-terminated string (which is what the C function expects) and passes a pointer to the string. The C function can modify this string, but can't lengthen it. Any changes to the string will be reflected in the script variable on return from the function. If you are passing a pointer to something other than a string, then pass the parameter by reference.

You can specify the use of non-platform-native characters as arguments and return values using the LMBCS and Unicode keywords.

- Unicode specifies a Unicode string of two-byte characters (words) using the platform-native byte order.
- LMBCS specifies a LMBCS optimization group 1 string (multibyte characters).

The following table summarizes the behavior of string arguments to a C function.

Declaration for the string argument in the Declare statement for the C function cF	*How the arg is passed when cF is called in one of these forms:* *cF ((arg))* *cF (ByVal (arg))*	*How the arg is passed when cF is called in one of these forms:* *cF (arg)* *cF (ByVal arg)*
ByVal String	As a 4-byte pointer to a copy of *arg*'s character bytes. If cF changes the bytes, the value of *arg* does not change. If cF writes past the end of the string, it produces an error.	As a 4-byte pointer to *arg*'s character bytes. If cF changes the bytes, the value of *arg* changes. If cF writes past the end of the string, it produces an error.
String	As a 4-byte pointer to a copy of the string in the platform-native character set format. If cFchanges the bytes, the value of *arg* does not change.	As a 4-byte pointer to the string in the platform-native character set format. cF can change the bytes only by dereferencing the existing string and adding a reference to the new one.

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 Long, winTitle As String, _
winLength as Long
   winTitle = Stringe(255,0)
   activeWin = GetActiveWin
   winLength = GetWindowText(activeWin, winTitle, 255)
   winTitle = Left(winTitle, winlength)
   Messagebox winTitle, "Window title"
End Sub
```

To get a window title at run time, use the GetWindowText function. GetWindowText has one input parameter (the window handle, of data type long) 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 "User32" Alias _
"GetWindowTextA"   (ByVal hWnd As Long, _
                    ByVal lpString As Long _
                    ByVal chMax As Long) As Long
```

Note 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 Long
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)
```

Passing arrays, types, and objects

Array, type, and object arguments to C functions are not supported under OS/2.

Passing arrays as arguments

Because LotusScript stores an array in a private format, you can pass an array by reference to a C function only if the function is specifically written for LotusScript. The following example shows how to declare and implement a C function that takes a LotusScript array of long values.

In LotusScript:

```
Declare Function LSArrayArg Lib "MYDLL" (ArrLng () As Long)_
 As Long
Dim MyArr(0 to 5) As Long
Print LSArrayArg(MyArr)
```

In C:

```
long C_CALL_TYPE LSArrayArg(LSsValueArray *pLSArr)
{
     long *pData=pLSArr->Data;
      //pData points to first array element
     return pData[0]+pData[1]; //Sum first 2 array elements
}
```

Or:

```
long C_CALL_TYPE LSArrayArg(long **pLSArr)
{
     long *pData=*pLSArr;
     //pData points to first array element
     return pData[0]+pData[1]; //Sum first 2 array elements
```

C_CALL_TYPE is the calling convention: Pascal, STDCALL, _System, or CDEL.

Other C functions may require an array, such as the Windows function SetBitmapBits. You can still pass the array by passing the first array element by reference with the Any keyword, as shown in the following example.

In LotusScript:

```
Declare Function FncArrayArg(A As Any) As Long
Dim MyArr(0 to 5) As Long
Print FncArrayArg(MyArr(0))
```

In C:

```
long C_CALL_TYPE FncArrayArg(long *pArr)
{
     return pArr[0]+pArr[1]; //Sum first 2 array elements
}
```

Passing types as arguments

Some C functions can require a data structure as a parameter. An example is the Windows API function GetBrushOrgEx, which requires a pointer to a point structure. You can define a suitable data type, such as Point, and use that type definition to declare the C function. Since type variables are passed by reference, the C function receives a 4-byte pointer to the storage for the type variable.

LotusScript allows you to specify an optional string type, Unicode or LMBCS, on a type parameter in the Declare statement for a C function. The declarations have this form, for a function UniTest with one type argument and a function LMBCSTest with one type argument, where t1 is a user-defined data type:

```
Declare Function UniTest Lib "Unilib" (typArg As Unicode t1)_
 As Long
Declare Function LMBCSTest Lib "lmbcslib" _
 (typArg As LMBCS t1) As Long
```

In the first example, all strings (fixed-length and variable-length) in type t1 and any of its nested types will be passed as Unicode strings. In the second example, all strings in type t1 (fixed- and variable-length) and any of its nested types will be passed as LMBCS strings.

If Unicode or LMBCS is not specified in this way, the default is to pass all strings in a type argument in the platform-native character set. This is compatible with LotusScript Release 2.

Strings contained in Variants in the type will not be affected. This change is incompatible with LotusScript Release 2, because translation to platform will be invoked by default on types containing strings (previously, these strings would have been passed as platform-native character set strings).

If the type contains a fixed-length non-Unicode string, the entire structure must be copied and its size adjusted. The size of the structure will be smaller (each fixed-length string will contain half as many bytes when translated to platform or LMBCS, since the length of the string is fixed and must be preserved). This implies that the string may be truncated (lose information) because a Unicode string of length 20 may require more than 20 bytes to represent in platform (DBCS). The loss cannot occur with variable-length strings, since they are represented as pointers.

LotusScript aligns type members to their natural boundaries for cross-platform transportability:

Data type	*Alignment*
Boolean	2 bytes
Byte	1 byte
Integer	2 bytes
Long	4 bytes
Single	4 bytes
Double	8 bytes
Currency	4 bytes
String (LMBCS)	2 bytes
String (Unicode)	2 bytes
String (Platform)	1 byte
Variant	8 bytes

The resulting alignment will not match the platform-specific alignment on Windows 3.1 and Windows 95. For example, LotusScript aligns the type member B on a 4-byte boundary, while the default alignment in Windows 3.1 will be on a 2-byte boundary.

```
Type telMet
   A As Integer
   B As Long
End Type
```

Passing objects as arguments

When an object is passed to a C function, the function receives a 4-byte pointer to the unpacked data in the object. Because the data may include pointers to strings, arrays, lists, and product objects, it is unlikely that the C function has full knowledge of the internal structure of the object. You should use a C function to manipulate only the simplest objects.

Example 1

```
' The following statements declare the C function
' SetBitmapBits.Its 3rd argument is an array argument. This is
' declared as type Any. In the function call, passing
' bitArray(0) passes the array by reference.
Declare Sub SetBitmapBits Lib "_privDispSys" _
   (ByVal hBM As Integer, ByVal cBytes As Long, pBits As Any)
' ...
SetBitmapBits(hBitmap, cBytesInArray, bitArray(0))
```

Example 2

```
type mytype
     mName as string
end type

class myclass
     mName as string
end class

function VariantParam( v as Variant) as string
     dim tempstr as string
     tempstr = TypeName(v)
     VariantParam = tempstr
end function

sub initialize
     dim myinteger as integer
     dim mylong as long
     dim mystring as string
     dim myintlist list as integer
     dim myintarray() as integer
     dim mymytype as mytype
     dim mymyclass as myclass

     messagebox variantparam(myintlist)

     messagebox variantparam(myintarray)

'    Error: Type mismatch on: MYMYTYPE
'    messagebox variantparam(mymytype)

     messagebox variantparam(mymyclass)

end sub
```

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 resize the window. This example also uses data type suffix characters to save space in the Declare statements.

```
Declare Function GetActiveWindow Lib "User32" () As Long
Type Rect
      left As Long
      top As Long
      right As Long
      bottom As Long
End Type
Declare Sub GetWindowRect Lib "User32" (Byval hWnd As Long, _
lpRect As Rect)
Declare Sub MoveWindow Lib "User32" _
(Byval hWnd As Long, Byval x As Long, Byval y As Long, _
Byval nWidth As Long, nHeight As Long, Byval bRepaint As Long)
Sub Initialize
      Dim activeWin As Long, winRect As Rect
      activeWin = GetActiveWindow
      Call GetWindowRect(activeWin, winRect)
      Call MoveWindow(activeWin, winRect.left, winRect.top, _
      winRect.right/2, winRect.bottom/2, True)
End Sub
```

Return values

The data type of a C function can be established by explicit data type declaration in the Declare statement, by a data type suffix on the function name in the Declare statement, or by an applicable Def*type* statement. One of these must be in effect. If not, the data type of the return value defaults to Variant, which is illegal for a C function.

LotusScript data type	*Legal as C function return type?*	*C data type*
Boolean	Yes	bool
Byte	Yes	byte
Integer	Yes	int
Long	Yes	long
Single	Yes	float
Double	Yes	double
Currency	No	
String	Yes, except for fixed-length string	char * or char[]
Variant	No	
Product object	Yes (as a 4-byte object handle of type Long)	See LSX toolkit for details
User-defined object	Yes	See LSX toolkit for details
Type instance	No	
Any	No	
Array	No	
List	No	

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 asks 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 "User32" () As Long

' Gets the handle of the next window.
Declare Function GetNextWindow Lib "User32" _
                    (ByVal hwnd As Long, _
                    ByVal uFlag As Long)
```

```
                    As Long
' 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 "User32" (ByVal hwnd As Long)

' Gets the text in the window title bar.
Declare Function GetWindowText Lib "User32" Alias
"GetWindowTextA" _
                    (ByVal hwnd As Long, _
                    ByVal lpString As String,_
                    ByVal chMax As Long) As Long

' Sets the text in the window title bar.
Declare Sub SetWindowText Lib "User32" Alias "SetWindowTextA"
_
                    (ByVal hwnd As Long, _
                    ByVal lpString$)
```

Calling C language functions extended example

```
Sub Initialize

     Dim winTitle As String, winTitleLength As Long

     winTitle$ = String(255, 0)

     Dim findWinTitle As String, tempWinTitle As String

     Dim curWin As Long, found As Integer

     tempWinTitle = "I'm working here now!"

     findWinTitle = Inputbox("What window do you want to
use?")

     If findWinTitle = "" Then Exit Sub

     curWin = GetActiveWindow

     Do While curWin <> 0

          curWin = GetWindow(curWin, GW_HWNDNEXT)

          winTitleLength = GetWindowText(curWin, winTitle, _
          255)

          If Instr(1, winTitle, findWinTitle, 1) > 0 Then

               found = True

               Exit Do

          End If
```

```
        Loop
        If found Then
              Call SetWindowText(curWin, tempWinTitle)
              Call SetActiveWindow(curWin)
              While GetActiveWindow = curWin
                    Yield
              Wend
              winTitle = Left(winTitle, winTitleLength)
              Messagebox "Done working with " & winTitle & "!"
              Call SetWindowText(curWin, winTitle)
        Else
              Messagebox "Window not found!"
        End If
End Sub
```

LS2J: Connecting with Java

Using LotusScript and LS2J, you can access Java classes, giving you a powerful cross-platform extension to LotusScript. Developers can access Java in LotusScript programs as a set of predefined LotusScript objects. This set of objects allow LotusScript to use already created Java classes that are available in script libraries or found using the classpath.

About LS2J

LS2J is the interface that allows data to transfer from the Java data type to the LotusScript data type and allows LotusScript to execute Java object methods. LS2J allows LotusScript to create Java objects as if they are native to the LotusScript environment.

This set of LotusScript objects is implemented by way of a LotusScript Extension. You can use this LSX in any existing LotusScript implementation, standalone or embedded in another application, such as Enterprise System Builder (ESB), Lotus SmartSuite, or Lotus Notes.

Java security

LS2J enforces Java security as follows:

- Only public methods and fields are available.
- LS2J has the same access rights as a Java program which does not contain a package statement.

System requirements

LS2J is implemented on all Domino platforms. Your system must meet the following requirements:

- The system must have enough memory to support both the Java Virtual Machine (JVM) and the LotusScript client applications.
- To use LS2J from within Notes, remember that your LotusScript code must include:

```
Uselsx "*javacon"
```

Note LS2J is implemented entirely as an LSX (in C++). There is no Java component to distribute.

Using LS2J

To use LS2J from within Notes, your LotusScript code must include this line:

```
Uselsx "*javacon"
```

This loads the LS2J Dynamic Link Library (DLL) on Win32 and registers all the Application Data Types (ADTs). LotusScript provides a JavaSession ADT to be used as an instance to connect with the JVM.

This statement:

```
Set mySession = New JavaSession
```

creates a new Java session. If the JVM has not been started, one is created at this time.

It is up to the LotusScript client to load the LSX. The environment determines how the Java Virtual Machine (JVM) is set up and the limits on how LotusScript can access the Java data.

LotusScript locates the Java class files in two ways:

1. In the Script Libraries that the user specified through the USE statement.
2. On the file system as specified in the NOTES.INI variable JavaUserClasses.

To tell Notes where the Java classes are, include the following line in your NOTES.INI file:

```
JavaUserClasses = <classpath1>;<classpath2>; ... <classpathn>
```

For example, if the Java classes are in one directory, such as E:\LSI\test\java, the Notes.ini file would include the following line:

```
JavaUserClasses=E:\LSI\test\java
```

In the Notes environment, LotusScript locates Java classes and uses them as if they were LotusScript objects. For example, if you have a set of common classes that are written in Java, you may use those classes in LotusScript without modification.

Using Script Libraries with LS2J

This demonstrates how you might create your own Java Script Library and include it in your LotusScript application using LS2J.

1. Create a Java Script Library called xlib containing the following:

```
public class calculator {
  public int add(int a, int b) { return a + b; }
  public int div(int a, int b) { return a / b; }
  public int mul(int a, int b) { return a * b; }
  public int sub(int a, int b) { return a - b; }
}
```

2. Create a LotusScript Agent which uses the library

```
Option Public
Use "xlib"
Uselsx "*javacon"
Sub Initialize
  Dim mySession  As JavaSession
  Dim myClass As JavaClass
  Dim calculator As JavaObject
```

```
    Dim a,b,c As Integer
    Set mySession = New JavaSession()
    Set myClass = mySession.GetClass("calculator")
    Set calculator = myClass.CreateObject()
    a = 10
    b = 5
    c = calculator.mul(a,b)
    MessageBox "a * b = " & c
End Sub
```

Run the program. The answer is 50.

Description of the USE statement

The syntax of the LotusScript USE statement is:

```
USE <script_library>
```

The Use statement examines the type of the Script Library. If the Script Library contains LotusScript, processing proceeds as before. If the Script Library contains Java classes, the contained Java classes are available to the LotusScript program by using LS2J.

Note The restriction on using the LotusScript "Use" statement to incorporate a Script Library containing Java has been lifted.

Invoking a method in a Java object

There are two ways to invoke a method in a Java object.

1. Use the dot notation in JavaObject.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myObject As JavaObject
Set mySession = new JavaSession
Set myClass  = mySession.getClass("myjavaapp")
Set myObject = myClass.CreateObject()
Call myObject.myMethod(arg1, arg2)
```

2. Use the JavaMethod ADT.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
```

```
Dim myObject As JavaObject
Dim myMethod as JavaMethod
Set mySession = new JavaSession
Set myClass  = mySession.getClass("myjavaapp")
Set myMethod = myClass.getMethod("MyMethod", "()V")
Set myObject = myClass.CreateObject()
myMethod.Invoke(myObject);
```

The dot notation method is easier and more intuitive, but certain restrictions apply. The JavaMethod ADT method is significantly harder to use but is more appropriate for general use. Dot notation is ambiguous if any of the the conditions listed below occur. If they do, you can use the more general mechanism to resolve the ambiguity.

- Case sensitivity

  LotusScript is case insensitive while Java is case sensitive. While theoretically you can have two Java methods differ only in case — for example, MyMethod and mymethod — they are distinctly different methods. There is no way for LotusScript to identify the correct method to invoke using the dot notation. The result is JVM-dependent if you try to access the function; that is, the results may differ depending on what operating system you are using.

- Long method names

  LotusScript has an internal limitation of 40 characters for names. If you use the dot notation method, you won't be able to get to methods with names longer than 40 characters.

- Method overloading

  LotusScript currently does not support method overloading. Because Java does, it is fairly common for a Java class to contain methods of the same name but with different signatures. If you use the dot notation, LotusScript uses trial-and-error to try and match the method. It is somewhat JVM-dependent, because the method that is matched depends on the order which the methods are presented to LotusScript by the JVM through JNI. The following algorithm is used to match the method:

  a. Enumerate all methods with the specified name.

  b. Retrieve the signature and check for the number of arguments. If they don't match, move on.

c. If the number of arguments matches, try to convert arguments in LotusScript to arguments in Java. Move to the next method if the number of arguments don't match.

LS2J calls through to the Java method of the first successful match.

Most Java implementations enumerate methods in the order they were declared in the Java source file. However, that is not always the case: for example, the AIX JVM seems to enumerate the method in reverse order.

Error handling with LS2J

While using LS2J, LotusScript returns error messages when things go wrong. However, these messages may be misunderstood without additional accurate information.

The error model

The Java error model is catch and throw. The LotusScript error model is the ON ERROR statement and the error handling block. LotusScript catches the Java error and maps it to its error processing model. This allows the LotusScript user to manipulate the Java error with LotusScript error semantics through the use of the JavaError class.

The JavaError object, when retrieved from the JavaSession, contains the last error and the last StackTrace. Use this object to get the most accurate picture of what is occurring with your program when an error occurs using LS2J.

```
Example code using JavaError

Sub Initialize
      Dim mySession As New JavaSession
      Dim myError As JavaError
      On Error GoTo ErrorHandling
'...
' code here
'....
      Exit Sub
ErrorHandling:
      Set myError = mySession.getLastJavaError
      print Error
      print myError.ErrorMsg
      print myError.StackTrace
```

```
End Sub
```

This code sample prints the LotusScript error, the Java error, and the Java StackTrace.

Examples of LotusScript errors that might be misunderstood

Example 1:

You try to instantiate an object, but you have the wrong signature or number of arguments.

LotusScript says:	LS2J: Null JavaObject
Java ErrorMsg says:	java.lang.NoSuchMethodError: <init>

Example 2:

You try to instantiate an object, but an error occurs in the Constructor.

LotusScript says:	LS2J: Null JavaObject
Java StackTrace says:	java.lang.ArrayIndexOutOfBoundsException at myGraph.<init>(Compiled Code)

Example 3a:

You try to execute a method, but use the wrong number of arguments.

LotusScript says:	LS2J: Parameter mismatch calling Method <Method Name here>
Java ErrorMsg says:	LS2J error

Example 3b:

Now, you execute the method with the right arguments but there is an error in the method.

LotusScript says:	LS2J: Parameter mismatch calling Method <Method Name here>
Java StackTrace says:	java.lang.ArrayIndexOutOfBoundsException: 3 at myGraph.setOrientation(myGraph.java:262)

LS2J limitations

There are a few limitations with LS2J:

- You may not delete a Variant containing a JavaClass object.
- There are some data type limitations (see Data type mappings).
- LotusScript property names are case insensitive, but Java property names are case sensitive. If two Java properties are the same except for their case, use the GetProperty and GetValue or SetValue methods to access the correct property. Similarly, LotusScript method names are case insensitive, and Java method names are case sensitive. Java methods may also be overloaded; that is, they may differ only by parameter type. If two Java methods are the same except for their case, or except for their parameter type, use the GetMethod and Invoke methods to access the correct method. See Invoking a method in a Java object. Java method calls are limited to twelve parameters.
- LotusScript can access all Java values and classes; however, there is no mechanism for Java to access LotusScript objects directly.

LS2J classes

With LS2J, Lotus introduces the concept of a Java object reference. Similar to an OLE object reference, it is not a predefined class; rather, it represents a runtime instance of a Java object. Its properties and methods are determined at run time.

The following table lists the LS2J interface classes.

LS2J class	*Description*
JavaClass	Represents a Java class.
JavaError	Allows LotusScript programmers to find an error raised from the Java program.
JavaMethod	Contains information about Java methods in a class.
JavaMethodCollection	Is the enumerator for all the methods in a Java class.
JavaObject	Represents a Java object instance. This is the key to connecting with a Java object.
JavaProperty	Contains information about a Java property in a class.
JavaPropertyCollection	Is the enumerator for all the properties in a Java class.
JavaSession	Represents a connection instance of JVM with which LotusScript is interfacing.

JavaClass class

JavaClass is the reference to a Java class. You can create an instance of the object, or you can look at the static properties (fields) and invoke static methods of the class.

Properties

ClassName property

Methods

CreateObject method

getClassMethods method

getClassProperties method

GetMethod method

GetProperty method

Access

You can access a JavaClass object by getting:

- A specific class using the getClass method of the JavaSession class
- The JClass property of a JavaMethod object
- The JClass property of a JavaProperty object

Example: JavaClass class

This script gets the Java class "Java.lang.Integer" and creates an object of that class in LotusScript.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myObject As JavaObject
Set mySession = New JavaSession ()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Create a "java.lang.Integer" object
Set myObject = myClass.CreateObject("(I)V", 5)
```

ClassName property

This property is the name of the JavaClass object. The property is read only.

Defined in
JavaClass

Data type
String

Syntax
To get: *string* = *javaclass*.**ClassName**

Usage
This property is useful for retrieving a class name. For example, if the class has been passed to a function, this property allows you to find out the name of that class.

Example: ClassName property
See Example: JClass property.

CreateObject method

This method creates a JavaObject instance base of the JavaClass object.

Defined in
JavaClass

Syntax
Set *javaobject* = *javaclass*.**CreateObject(***Signature[, Argument1, ..., Argumentn*]**)**

Parameter
Signature

String. This is a JNI signature representing the constructor to use to initialize the object.

JNI Signature	*Description*	*Examples*	*Constructor requires*
B	byte	(B)V	A byte argument
C	char	(C)V	A char argument
D	double	(D)V	A double argument
F	float	(F)V	A float argument
I	int	(I)V	An int argument
J	long	(J)V	A long argument
S	short	(S)V	A short argument
Z	boolean	(Z)V	A Boolean argument
L<fully-qualified-class>	fully-qualified class	"(Ljava/lang/Integer;)V" A java.lang.Integer argument	
[<sigtype>	Array of <sigtype>	([I)V	An int array
L<fully-qualified-class>; L<fully-qualified-class>; I L<fully-qualified-class>;	fully-qualified class fully-qualified class int fully-qualified class	"(Ljava/lang/String;Ljava/lang/String;ILjava/lang/String;)V" 4 arguments: java.lang.String, java.lang.String, int, java.lang.String	

If the constructor has no parameters, call CreateObject with no parameters:

```
<javaclass>.CreateObject()
```

If the constructor has one or more parameters, call CreateObject with a signature parameter as follows:

```
<javaclass>.CreateObject("(...)V")
```

where ... represents the types of one the parameters in the table. Note that each signature for a fully-qualified-class must start with an L and end with a semicolon.

Argumentn

The arguments needed by the constructor, varying from 0 to 12. These arguments are optional.

Usage
This method creates a JavaObject instance base of the JavaClass object, and returns a JavaObject reference. By default, the empty constructor is used. Otherwise, the user must specify which constructor by using the signature.

Error thrown
LS2J error if there are any issues regarding the signature or the arguments.

Example: CreateObject method
This script gets the Java class "Java.lang.Integer" and creates an object of that class in LotusScript.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myObject As JavaObject
Set mySession = New JavaSession ()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Create a "java.lang.Integer" object
Set myObject = myClass.CreateObject("(I)V", 5)
```

getClassMethods method

This method returns a list of methods.

Defined in
JavaClass

Syntax
Set *Javamethodcollection = javaclass*.**getClassMethods()**

Usage
This method returns a list of all the public methods belonging to the class specified by *javaclass*.

Example: getClassMethods method
See Example: JavaMethod class.

getClassProperties method

This method returns a list of properties.

Defined in
JavaClass

Syntax
Set *Javapropertycollection = javaclass*.**getClassProperties()**

Usage
This method returns a list of all the public properties belonging to the class specified by *javaclass*.

Example: getClassProperties method
This script prints out all the available public properties belonging to the java.lang.Integer class.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myPCollection As JavaPropertyCollection
Dim msg As String
Set mySession = New JavaSession ()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all properties belonging
' to the java.lang.Integer class
Set myPCollection = myClass.getClassProperties()
msg = "The properties belonging to java.lang.Integer are:"
ForAll p in myPCollection
  msg = msg & {
} & p.PropertyName
End ForAll
MessageBox msg
```

GetMethod method

This method returns a method from a JavaClass object.

Defined in
JavaClass

Syntax
Set *Javamethod = javaclass*.**GetMethod(***Methodname, Signature***)**

Parameters
Methodname

String. Case sensitive. Name of the method you want a handle of.

Signature

String. JNI Signature representing the method needed.

JNI Signature	*Description*
B	byte
C	char
D	double
F	float
I	int
J	long
S	short
V	void
Z	Boolean
L<fully-qualified-class>	fully-qualified class
[<sigtype>	Array of <sigtype>

Usage
This method returns the method matching the name given with the specified signature.

Error thrown
NoSuchMethodException if the Java method does not exist with the signature given.

Example: GetMethod method

This script gets the "toString" method from the java.lang.Integer class that requires an Integer argument, and returns a string.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myMethod As JavaMethod
Dim Count As Integer
Set mySession = New JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get the toString method which
' takes an Integer and returns a string
Set myMethod =
myClass.GetMethod("toString","(I)Ljava/lang/String;")
print {Data type of toString
        return value is a }  _
        & TypeName(myMethod.invoke(,5))
print {result of invoking the method
        with a value of 5 is } _
        & myMethod.invoke(,5)
```

GetProperty method

This method returns a property.

Defined in

JavaClass

Syntax

Set *Javaproperty = javaclass*.**getProperty(***PropertyName***)**

Parameter

Propertyname

String. Case sensitive. Name of the property you want a handle of.

Usage

This method returns the property matching *Propertyname*.

Error thrown

"LS2J: No such Field Invalid" if the property isn't static or does not exist.

Example: GetProperty method

This script gets the "MIN_VALUE" static property from the java.lang.Integer class.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myProperty As JavaProperty
Dim Count As Integer
Set mySession = New JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get the toString method which takes
' an Integer and returns a string
Set myProperty = myClass.GetProperty("MIN_VALUE")
Print "Property data type is "  _
  & TypeName(myProperty.getValue())
Print "Value of property is " & myProperty.getValue()
```

JavaError class

JavaError is the main interface for LotusScript to get information about Java errors that occur. Besides the standard exceptions mentioned in each object, the Java program can raise an exception for many other reasons. All these errors are caught and re-raised as the LotusScript Error-JavaError. Users can put an "on error" condition to catch any Java exceptions. The JavaException is not cleared until the method ClearJavaException is called. In order to find out more, the LotusScript user uses the JavaError object.

Properties

ErrorMsg property

StackTrace property

Example: JavaError class

This script will catch an error while trying to get a specific Java class. After the error is reported by the MessageBox, the JavaError is cleared using ClearJavaError.

```
    Dim mySession As JavaSession
    Dim myClass As JavaClass
    Dim myError As JavaError
    On Error GoTo Catch
    Set mySession = new JavaSession()
    Set myClass = mySession.GetClass("Invalid")
done:
    exit sub
Catch:
    Set myError = mySession.getLastJavaError()
    MessageBox myError.errormsg,, "Error"
    mySession.ClearJavaError
    Resume done
```

ErrorMsg property

This property contains the last Java error that occurred. The property is read only.

Defined in

JavaError

Data type

String

Syntax

To get: *String = javaError*.**ErrorMsg**

Example: ErrorMsg property

See Example: JavaError class.

StackTrace property

This property contains the call stack of the error. The property is read only.

Defined in
JavaError

Data type
String

Syntax
To get: *String = javaError*.**StackTrace**

Example: StackTrace property
Using three Java Classes, this script demonstrates an error that would cause a stack trace. The result is printed under the section labeled "LotusScript code."

In ClassA.java

```
public class ClassA {
  ClassB CB = new ClassB();
  public int FunctA(){
        return CB.FunctB();
  }
}
```

In ClassB.java

```
public class ClassB {
  ClassC CC = new ClassC();
  public int FunctB(){
        return CC.FunctC();
  }
}
```

In ClassC.java

```
public class ClassC {
  int x = 10;
  int y = 0;
  public int FunctC(){
```

```
        return x/y;
    }
}
```

LotusScript code

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myMethod As JavaMethod
Dim myObject As JavaObject
Dim myError As JavaError
Set mySession = New JavaSession ()
Set myClass = mySession.GetClass("ClassA")
Set myObject = myClass.CreateObject("()V")
Set myMethod = myClass.getMethod("FunctA", "()I")
On Error GoTo errhandler:
print myMethod.invoke(o)

done:
exit sub
errhandler:
set myError = mySession.getLastJavaError()
MessageBox myError.StackTrace,, Error
Resume done

(Results)
java.lang.ArithmeticException: / by zero
  at valid5.javaconn.java.ClassC.FunctC(ClassC.java:6)
  at valid5.javaconn.java.ClassB.FunctB(ClassB.java:5)
  at valid5.javaconn.java.ClassA.FunctA(ClassA.java:5)
```

JavaMethod class

The JavaMethod class describes a public method in a JavaClass object. This class is used when the dot reference method fails (as happens with case sensitivity, method overloading, or long names).

Properties

JClass property

MethodName property

Modifier property

Signature property

Methods

Invoke method

Example: JavaMethod class

This script prints out the number of toString methods belonging to the java.lang.Integer class.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myMCollection As JavaMethodCollection
Dim Count As Integer
Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all methods belonging
' to the java.lang.Integer class
Set myMCollection = myClass.getClassMethods()
Count = 0
ForAll m in myMCollection
  If m.MethodName = "toString" Then
        Count = Count + 1
  End If
End ForAll
```

```
MessageBox "There are " & Count & " instances of the
toString _
  method in the method collection for java.lang.Integer"
```

JClass property

This property is a JavaClass object representing the JavaClass object in which the method belongs. The property is read only.

Defined in
JavaMethod

Data type
JavaClass

Syntax
Set *JavaClass = javamethod*.**JClass**

Example: JClass property (JavaMethod class)
This script displays a message box with the method name in the title and the class it belongs to in the message.

```
Sub Test (myMethod As JavaMethod)
' JClass property of JavaMethod
' is an instance of JavaClass
MessageBox "Belongs to " & myMethod.JClass.ClassName _
      & " Class",, "Method " & myMethod.MethodName
End Sub
```

MethodName property

This property contains the name of the method. This property is read only.

Defined in
JavaMethod

Data type
String

Syntax
To get: *String = javamethod*.**MethodName**

Usage

This is the name of the method in *javamethod*. A method name might not be unique, because each method name could have a different signature.

Example: MethodName property

This script prints out the position within the collection where there is a toString method.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myMethod As JavaMethod
Dim myMCollection As JavaMethodCollection
Dim msg As String

Set mySession = new JavaSession()
msg = " "
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all methods belonging
' to the java.lang.Integer class
Set myMCollection = myClass.getClassMethods()
set myMethod = myMCollection.getFirst()
do
  if myMethod.MethodName = "toString" then
        msg = msg + {
              toString } & myMethod.Signature & _
{ is located at element } _
             & myMCollection.Current & _
{ within the collection}
  End If
  set myMethod = myMCollection.getNext()
loop while myMCollection.Current <> 1
' Because getNext loops back to 1 when the end is reached
MessageBox msg
```

Modifier property

This returns the modifier value(s) for a Java method. The property is read only.

Defined in
JavaMethod

Data type
Long

Syntax
To get: *Long = javamethod*.**Modifier**

Usage
The Modifier property returns a combination of bits for the modifier(s) of the Java method (as specified by *javamethod*) as follows:

Modifier	*Bit*
public	1
static	8
final	16
synchronized	32
native	256
abstract	1,024

For example, if a method is declared in Java as "public static" the value of Modifier would be 9: the value of 1 for public added to the value of 8 for static.

Note The keywords private and protected are not available (private and protected methods are not available with LS2J).

Example: Modifier property (JavaMethod class)

```
Dim mySession As JavaSession

Dim myClass As JavaClass

Dim myMethod As JavaMethod

Set mySession = New JavaSession()

Set myClass = mySession.GetClass("java/lang/Thread")

Set myMethod = myClass.GetMethod("start", "()V")

' java.lang.Thread.start() modifiers:

'   public         = 1
```

```
'   synchronized   = 32
'   native         = 256
' Modifier value is 289
Print "start() Modifier is " & myMethod.Modifier
```

Signature property

This property is the JNI signature representing the method arguments and return value. The property is read only.

Defined in
JavaMethod

Data type
String

Syntax
To get: *String= javamethod*.**Signature**

Example: Signature property
See Example: MethodName property.

Invoke method

This method executes a method.

Defined in
JavaMethod

Syntax
Set *Variant = javamethod*.**Invoke(***[JavaObject [,Argument1...[, Argument12]]***)**

Parameters
JavaObject

JavaObject. The instance of an object, if the method is not static. Optional if the method is static.

Argumentn

Variant. Optional. The arguments needed by the method. Maximum of 12 arguments.

Return value
Variant. Result of the invoked method.

Example: Invoke method
See Example: GetMethod method.

JavaMethodCollection class

The JavaMethodCollection class enumerates all the methods of a JavaClass object. This is a true enumerator class and you can use the ForAll statement on it.

Properties
Count property

Current property

Methods
getFirst method

getNext method

getNth method

Example: JavaMethodCollection class
See Example: JavaMethod class.

Count property

This property contains the number of methods in the enumeration. The property is read only.

Defined in
JavaMethodCollection

Data type
Integer

Syntax
To get: *Integer = javamethodcollection*.**Count**

Usage
Use this to retrieve the number of methods in *javamethodcollection*. Overloaded methods are counted each as a separate method.

Example: Count property (JavaMethodCollection class)

This script prints out the number of toString methods belonging to the java.lang.Integer class.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myMethod As JavaMethod
Dim myMCollection As JavaMethodCollection
Dim Count As Integer, i As Integer
Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all methods belonging
' to the java.lang.Integer class
Set myMCollection = myClass.getClassMethods()
For i = 1 to myMCollection.Count
  Set myMethod = myMCollection.getNth(i)
  If myMethod.MethodName = "toString" then
       Count = Count + 1
  End If
Next
Print "There are " & Count & " instances of toString" + _
  "method in the Method collection for java.lang.Integer"
```

Overloaded methods are also counted.

```
Sub Initialize
  Dim mySession As JavaSession
  Dim myClass As JavaClass
  Dim myMCollection As JavaMethodCollection
  Dim i As Integer
  Dim msg As String
  Set mySession = New JavaSession()
  Set myClass = mySession.GetClass("java/lang/Object")
  Set myMCollection = myClass.GetClassMethods()
```

```
    i = 1
    msg = ""
    ForAll m In myMCollection
        msg = msg & i & " " & m.MethodName & " " &
m.Signature & {
}
        i = i + 1
    End ForAll
    MessageBox msg & "Count is " & myMCollection.Count
End Sub
```

The MessageBox displays

```
1 getClass ()Ljava/lang/Class;
2 hashCode ()I
3 equals (Ljava/lang/Object;)Z
4 toString ()Ljava/lang/String;
5 notify ()V
6 notifyAll ()V
7 wait (J)V
8 wait (JI)V
9 wait ()V
Count is 9
```

The last three methods are overloaded.

Current property

This property contains the current position in the enumeration. This property is read only.

Defined in

JavaMethodCollection

Data type

Integer

Syntax

To get: *Integer = javamethodcollection*.**Current**

Usage

This returns your exact location within the collection.

Example: Current property (JavaMethodCollection class)

This script prints out the position within the collection where there is a toString method.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myMethod As JavaMethod
Dim myMCollection As JavaMethodCollection


Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all methods belonging
' to the java.lang.Integer class
Set myMCollection = myClass.getClassMethods()
Set myMethod = myMCollection.getFirst()
Do
  If myMethod.MethodName = "toString" then
        Print "toString" & myMethod.Signature & _
" is located at the " & myMCollection.Current & _
" element within the collection"
  End If
  Set myMethod = myMCollection.getNext()
Loop While myMCollection.Current <> 1
' Because getNext loops back to 1 when the end is reached
```

getFirst method

This method returns the first JavaMethod object.

Defined in
JavaMethodCollection

Syntax
Set *javamethod = javamethodcollection*.**getFirst()**

Return value
JavaMethod. The first JavaMethod object within the enumeration.

Example: getFirst method (JavaMethodCollection class)
This script prints out the position within the collection where there is a toString method.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myMethod As JavaMethod
Dim myMCollection As JavaMethodCollection


Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all methods belonging
' to the java.lang.Integer class
Set myMCollection = myClass.getClassMethods()
Set myMethod = myMCollection.getFirst()
Do
  If myMethod.MethodName = "toString" then
        Print "toString" & myMethod.Signature & _
" is located at the " & myMCollection.Current & _
" element within the collection"
  End If
  Set myMethod = myMCollection.getNext()
```

```
Loop While myMCollection.Current <> 1
' Because getNext loops back to 1 when the end is reached
```

getNext method

This method returns the next JavaMethod object in the enumeration.

Defined in
JavaMethodCollection

Syntax
Set *javamethod = javamethodcollection*.**getNext()**

Return value
JavaMethod. The next JavaMethod object within the enumeration. If you pass the last method within the enumeration, the first one will be returned.

Example: getNext method (JavaMethodCollection class)
This script prints out the position within the collection where there is a toString method.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myMethod As JavaMethod
Dim myMCollection As JavaMethodCollection

Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all methods belonging
' to the java.lang.Integer class
Set myMCollection = myClass.getClassMethods()
Set myMethod = myMCollection.getFirst()
Do
  If myMethod.MethodName = "toString" then
        Print "toString" & myMethod.Signature & _
" is located at the " & myMCollection.Current & _
" element within the collection"
```

```
    End If
    Set myMethod = myMCollection.getNext()
Loop While myMCollection.Current <> 1
' Because getNext loops back to 1 when the end is reached
```

getNth method

This method returns the Java method in a specified position in the enumeration.

Defined in
JavaMethodCollection

Syntax
Set *Javamethod = javamethodcollection*.**getNth(***n***)**

Parameters
n

Integer. The exact position within the enumeration to get the method from.

Return value
JavaMethod. The method in the nth position in the enumeration. If there is no method at the specified position, it returns null.

Example: getNth method (JavaMethodCollection class)
This script prints out the number of toString methods belonging to the java.lang.Integer class.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myMethod As JavaMethod
Dim myMCollection As JavaMethodCollection
Dim Count As Integer, i As Integer
Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all methods belonging
' to the java.lang.Integer class
```

```
Set myMCollection = myClass.getClassMethods()
For i = 1 to myMCollection.count
  Set myMethod = myMCollection.getNth(i)
  If myMethod.MethodName = "toString" then
        Count = Count + 1
  End If
Next
Print "There are " & Count & " instances of the toString "
+ _
  "method in the Method collection for java.lang.Integer"
```

JavaObject class

The JavaObject Reference is the key to connecting with a Java object. It is returned from the CreateObject method of the JavaClass class or the GetJavaObject function. It is similar to an OLE reference and represents a Java object instance. The properties and methods are adapted automatically. It can be assigned only to a Variant.

Although Java Native Interface (JNI) allows us to look at properties (fields) and methods with different protected attributes, LotusScript adapts only the public ones.

Properties

[adaptive -- all public fields (static, instance)]

Methods

[adaptive -- all public methods (static, instance)]

Usage

The JavaObject reference is not set if LotusScript has problems adapting the specified Java object. If the program tries to use the properties or methods, it raises an “Object Variable Not Set” error.

Note

Due to LotusScript limitations, you cannot access the following properties and methods:

- Properties and methods with names over 40 characters
- Properties and methods with names that differ only in case (LotusScript is not case sensitive whereas Java is)
- Methods with the same name and number of arguments, but with a different signature

For these conditions, you must explicitly use JavaProperty class and JavaMethod class.

Example: JavaObject class

This script prints the area of a 2 X 4 rectangle. Then, with the use of setValue, it resets the width and height and prints the new size.

```
Dim mySession As JavaSession

Dim myClass As JavaClass

Dim myProperty As JavaProperty

Dim myObject As JavaObject

Set mySession = new JavaSession()

' setProperty on custom class

Set myClass =
mySession.GetClass("valid5/javaconn/java/Rectangle;")

Set myObject = myClass.CreateObject("(II)V", 2,4)

print "The area of our Rectangle (2 X 4) is " & _
myObject.getArea()

Set myProperty = myClass.GetProperty("width")

call myProperty.setValue(5,o)

Set myProperty = myClass.GetProperty("height")

call myProperty.setValue(10,o)


Print "The area of our Rectangle (5 X 10) is " & _
myObject.getArea()
```

JavaProperty class

The JavaProperty class describes a public property in a JavaClass object. This class is used for instances when the dot reference method fails (because of case sensitivity or long names).

Properties

JClass property

PropertyName property

Modifier property

Type property

Methods

getValue method

setValue method

Example: JavaProperty class

This script prints out all the properties belonging to a JavaClass object.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myPCollection As JavaPropertyCollection

Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all Properties belonging
' to the java.lang.Integer class
Set myPCollection = myClass.getClassProperties()
Print "myPCollection.count & " properties of the " & _
  myClass.ClassNAme & " class are :"
ForAll p in myPCollection
  Print p.PropertyName & " (" & myPCollection.current & _
")"
End ForAll
```

JClass property

This is a JavaClass object representing the Java class in which the JavaProperty belongs. The property is read only.

Defined in
JavaProperty

Data type
JavaClass

Syntax
Set *JavaClass = javaproperty*.**JClass**

Example: JClass property (JavaProperty class)
This script displays a message box with the PropertyName in the title and the class it belongs to in the message.

```
Sub Test (myProperty As JavaProperty)
' JClass property of JavaProperty
' is an instance of JavaClass
MessageBox "Belongs to " & myProperty.JClass.ClassName & _
  " Class",, "Property " & myProperty.PropertyName
End Sub
```

PropertyName property

This is the name of the JavaProperty. The property is read only.

Defined in
JavaProperty

Data type
String

Syntax
To get: *String = javaproperty*.**PropertyName**

Example: PropertyName property

This script displays a message box with the PropertyName in the title and the class it belongs to in the message.

```
Sub Test (myProperty As JavaProperty)
' JClass property of JavaProperty is
' an instance of JavaClass
MessageBox "Belongs to " & myProperty.JClass.ClassName & _
   " Class",, "Property " & myProperty.PropertyName
End Sub
```

Modifier property

This returns the modifier value(s) for a Java property. The property is read only.

Defined in

JavaProperty

Data type

Long

Syntax

To get: *Long = javaproperty*.**Modifier**

Usage

The Modifier property returns a combination of bits for the modifier(s) of the Java property (as specified by *javaproperty*) as follows:

Modifier	*Bit*
public	1
static	8
final	16
volatile	64
transient	128

For example, if a property is declared in Java as "public static final" the value of Modifier would be 25: the value of 1 for public added to the value of 8 for static added to the value of 16 for final.

Note The keywords private and protected are not available (private and protected properties are not available with LS2J).

Example: Modifier property (JavaProperty class)

```
Dim mySession As JavaSession

Dim myClass As JavaClass

Dim myProperty As JavaProperty

Set mySession = New JavaSession()

Set myClass = mySession.GetClass("java/lang/Short")

Set myProperty = myClass.GetProperty("MIN_VALUE")

' java.lang.Short.MIN_VALUE has modifiers:

' public     =    1

' static     =    8

' final =    16

' Modifier value is 25

Print "MIN_VALUE Modifier is " myProperty.Modifier
```

Type property

This is the LotusScript data type of the JavaProperty. The property is read only.

Java	*LotusScript*	*LotusScript data type*
byte	Byte	17
short	Integer	2
int	Long	3
long	Double	5
float	Single	4
double	Double	5
boolean	Boolean	11
char	String	8
java.lang.String	String	8
other Objects	JavaObject	32

Defined in

JavaProperty

Data type

Long (no decimals)

Syntax
To get: *Long = javaproperty*.**Type**

Example: Type property
This script prints out all the properties and their data types.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myPCollection As JavaPropertyCollection

Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all Properties belonging
' to the java.lang.Integer class
Set myPCollection = myClass.getClassProperties()
Print " myPCollection.count & " properties of the " _
  & myClass.ClassNAme & " class are :"
ForAll p in myPCollection
  Print p.PropertyName & " (" & p.Type & ")"
End ForAll
```

getValue method

This method returns the JavaProperty value.

Defined in
JavaProperty

Syntax
Set *Variant = javaProperty*.**getValue(***[JavaObject]***)**

Parameter
JavaObject

JavaObject. The instance of an object from which you want a property value, if the property is not static. Optional if the property is static.

Return value

Variant

Value of the JavaProperty.

Usage

This method is used to get the value of either a public static property or a public object property. The object is necessary if the property is not static, and disregarded if the property is static.

Example: getValue method

This script prints out the position of MAX_VALUE within the collection.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myPCollection As JavaPropertyCollection


Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all Properties belonging
' to the java.lang.Integer class
Set myPCollection = myClass.getClassProperties()
Print " myPCollection.count & " properties of the " _
  & myClass.ClassNAme & " class are :"
ForAll p in myPCollection
  If p.type <> 32 then     ' If it's not an object
       Print p.PropertyName & _
" (" & myPCollection.current & ") and value is " & _
p.getValue()
  Else
       Print p.PropertyName & _
" (" & myPCollection.current & ") and value is " & _
p.getValue().toString()
  End If
End ForAll
```

setValue method

This method sets the JavaProperty value.

Defined in
JavaProperty

Syntax
Call *javaproperty*.**setValue(***NewValue [, JavaObject]***)**

Parameters
NewValue

Variant. New Value for the JavaProperty.

JavaObject

JavaObject. Object to be set, if the property is not static. Optional if the property is static.

Error thrown
IllegalAccessException. Thrown if the value is of the wrong type, or if the property is read only.

Example: setValue method
This script prints the area of a 2 X 4 rectangle then, with setValue, resets the width and height and prints the new size.

```
Dim mySession As JavaSession

Dim myClass As JavaClass

Dim myProperty As JavaProperty

Dim myObject As JavaObject

Set mySession = new JavaSession()

' setProperty on custom class

Set myClass =
mySession.GetClass("valid5/javaconn/java/Rectangle;")

Set myObject = myClass.CreateObject("(II)V", 2,4)

Print "The area of our Rectangle (2 X 4) is " & _
myObject.getArea()

Set myProperty = myClass.GetProperty("width")

Call myProperty.setValue(5,o)

Set myProperty = myClass.GetProperty("height")
```

```
Call myProperty.setValue(10,o)


Print "The area of our Rectangle (5 X 10) is " & _
myObject.getArea()
```

JavaPropertyCollection class

The JavaPropertyCollection class enumerates all the properties of a JavaClass object. This is a true enumerator class and you can use the ForAll statement with it.

Properties

Count property

Current property

Methods

getFirst method

getNext method

getNth method

Example: JavaPropertyCollection class

This script prints out all the properties belonging to a JavaClass object.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myPCollection As JavaPropertyCollection

Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all Properties belonging
' to the java.lang.Integer class
Set myPCollection = myClass.getClassProperties()
Print " myPCollection.count & "properties of the " _
  & myClass.ClassNAme & " class are :"
ForAll p in myPCollection
  Print p.PropertyName & " (" & myPCollection.current & ")"
End ForAll
```

Count property

This property contains the number of properties in the enumeration. The property is read only.

Defined in
JavaPropertyCollection

Data type
Integer

Syntax
To get: *Integer = javapropertycollection*.**Count**

Usage
Use this property to retrieve the number of properties in *javapropertycollection*.

Example: Count property (JavaPropertyCollection class)
This script prints out all the properties belonging to a JavaClass object.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myPCollection As JavaPropertyCollection

Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all Properties belonging
' to the java.lang.Integer class
Set myPCollection = myClass.getClassProperties()
Print " myPCollection.count & "properties of the " _
  & myClass.ClassNAme & " class are :"
ForAll p in myPCollection
  Print p.PropertyName & " (" & myPCollection.current & ")"
End ForAll
```

Current property

This property contains the current position in the enumeration. The property is read only.

Defined in
JavaPropertyCollection

Data type
Integer

Syntax
To get: *Integer = javaPropertycollection*.**Current**

Usage
This property contains the exact location within the collection.

Example: Current property (JavaPropertyCollection class)
This script prints out the position of MAX_VALUE within the collection.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myProperty As JavaProperty
Dim myPCollection As JavaPropertyCollection

Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all methods belonging
' to the java.lang.Integer class
Set myPCollection = myClass.getClassProperties()
Set myProperty = myPCollection.getFirst()
Do
  If myProperty.PropertyName = "MAX_VALUE" then
        Print "MAX_VALUE is located at the " & _
myPCollection.current _
             & " position within the collection"
        Exit Do
```

```
  End If
  Set myProperty = myPCollection.getNext()
Loop While myPCollection.Current <> 1
' Because getNext loops back to 1 when the end is reached
```

getFirst method

This method returns the first JavaProperty.

Defined in
JavaPropertyCollection

Syntax
Set *javaProperty = javaPropertycollection*.**getFirst()**

Return value
JavaProperty. The first JavaProperty within the enumeration.

Example: getFirst method (JavaPropertyCollection class)
This script prints out the position of MAX_VALUE within the collection.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myProperty As JavaProperty
Dim myPCollection As JavaPropertyCollection

Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all methods belonging
' to the java.lang.Integer class
Set myPCollection = myClass.getClassProperties()
Set myProperty = myPCollection.getFirst()
Do
  If myProperty.PropertyName = "MAX_VALUE" then
       Print "MAX_VALUE is located at the " & _
myPCollection.current _
```

```
            & " position within the collection"
        Exit Do
    End If
    Set myProperty = myPCollection.getNext()
Loop While myPCollection.Current <> 1
' Because getNext loops back to 1 when the end is reached
```

getNext method

This method returns the next JavaProperty in the enumeration.

Defined in
JavaPropertyCollection

Syntax
Set *javaProperty = javaPropertycollection*.**getNext()**

Return value
JavaProperty. The next JavaProperty within the enumeration. If you pass the last property within the enumeration, the first one is returned.

Example: getNext method (JavaPropertyCollection class)
This script prints out the position of MAX_VALUE within the collection.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myProperty As JavaProperty
Dim myPCollection As JavaPropertyCollection

Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all methods belonging
' to the java.lang.Integer class
Set myPCollection = myClass.getClassProperties()
Set myProperty = myPCollection.getFirst()
Do
```

```
    If myProperty.PropertyName = "MAX_VALUE" Then
        Print "MAX_VALUE is located at the " & _
myPCollection.current _
            & " position within the collection"
        Exit Do
    End If
    Set myProperty = myPCollection.getNext()
Loop While myPCollection.Current <> 1
' Because getNext loops back to 1 when the end is reached
```

getNth method

This method returns the Java property in the specified position in the enumeration.

Defined in
JavaPropertyCollection

Syntax
Set *javaProperty = javaPropertycollection*.**getNth(***n***)**

Parameter
n

Integer. The exact position within the enumeration to get the property from.

Return value
JavaProperty. The property in the nth position in the enumeration. If there is no property at the specified position, the method returns null.

Example: getNth method (JavaPropertyCollection class)
This script prints out all the properties belonging to a JavaClass object.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myPCollection As JavaPropertyCollection
Dim i As Integer
```

```
Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Get a list of all Properties belonging
' to the java.lang.Integer class
Set myPCollection = myClass.getClassProperties()
Print " myPCollection.count & "properties of the " _
  & myClass.ClassNAme & " class are :"
For i = 1 to myPCollection.count
  Set myProperty = myPCollection.getNth(i)
  Print myProperty.PropertyName & " (" &
myPCollection.current & ")"
next i
```

JavaSession class

JavaSession is the starting point for access to the Java objects. The session attaches to the existing JVM, if there is one. If a JVM has not been started, the LotusScript client tries to create the JVM and apply all the specified arguments. You can create as many JavaSessions as you want. All the resources created are associated with a particular session. Delete the session object to reclaim the resources.

Properties

None

Methods

ClearJavaError method

GetClass method

GetLastJavaError method

Creation and access

To access the current JVM session.

Syntax

Dim *variableName* **As New JavaSession**

OR

Set *javaSession* **= New JavaSession**

Parameter

Arguments for this class are not documented because they are overwritten by Notes. These arguments are for internal use only.

Example: JavaSession class

This script gets the Java class "Java.lang.Integer" and creates an object of that class in LotusScript.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myObject As JavaObject
Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Create a "java.lang.Integer" object
Set myObject = myClass.CreateObject("(I)V", 5)
```

ClearJavaError method

This method clears the last JavaError method that occurred.

Defined in

JavaSession

Syntax

Call *javasession*.**ClearJavaError**

Usage

This method is used to clear the most recent JavaError. After you call *javasession*.ClearJavaError(), *javaerror*.errorMsg and *javaerror*.stackTrace return to their initial values: "No Java Errors or Exceptions" and "No Java stack trace," respectively.

Example: ClearJavaError method

This script catches an error while trying to get a specific Java class. After the error is reported by the message box, then all JavaErrors are cleared using ClearJavaError.

```
Dim myClass As JavaClass
Dim myObject As JavaObject
Dim myError As JavaError
```

```
    On Error GoTo Catch
    Set mySession = new JavaSession()
    Set myClass = mySession.GetClass("Invalid")
    exit Sub
Catch:
    Set myError = mySession.getLastJavaError()
    MessageBox myError.errormsg,, "Error"
    mySession.ClearJavaError
```

getClass method

This method returns a reference to a Java class.

Defined in
JavaSession

Syntax
Set *JavaClass* **=** *javasession*.**getClass(***ClassName$***)**

Parameters
ClassName$

String. The name of the class you would like to retrieve. For example, "java/lang/Integer."

Usage

This method will return a Java class reference with which a Java object can be created within LotusScript.

Note There are 2 ways you can represent a String class:

java.lang.String	example: set MyClass = Session.getClass("java.lang.String")
java/lang/String	example: set MyClass = Session.getClass("java/lang/String")

When you use the dot "." notation on the Macintosh, the Mac will return an error that the Class cannot be found. Instead, use the slash "/" notation. The slash "/" notation works on all platforms. Use the slash "/" notation in your applications for multi-platform support.

Error thrown

"JavaClassNotFound" is thrown if the Java class cannot be located.

Example: getClass method

This script gets the Java class "Java.lang.Integer" and creates an object of that class in LotusScript.

```
Dim mySession As JavaSession
Dim myClass As JavaClass
Dim myObject As JavaObject
Set mySession = new JavaSession()
' Get Java "java.lang.Integer" class
Set myClass = mySession.GetClass("java/lang/Integer;")
' Create a "java.lang.Integer" object
Set myObject = myClass.CreateObject("(I)V", 5)
```

getLastJavaError method

This method retrieves the last JavaError that occurred, and, in some cases, a call stack.

Defined in

JavaSession

Syntax

Set *JavaError = javasession*.**getLastJavaError**

Usage

This method is used to retrieve any possible JavaErrors that might have occurred.

Example: getLastJavaError method

This script catches an error while trying to get a specific Java class. After the error is reported by the MessageBox, then the JavaError is cleared using ClearJavaError.

```
    Dim myClass As JavaClass
    Dim myObject As JavaObject
    Dim myError As JavaError
    On Error GoTo Catch
    Set mySession = new JavaSession()
    Set myClass = mySession.GetClass("Invalid")
    exit Sub
Catch:
    Set myError = mySession.getLastJavaError()
    MessageBox myError.errormsg,, "Error"
    mySession.ClearJavaError
```

Data type mappings

LotusScript provides mapping for Java basic data types and Java reference types.

Basic data types

The basic data types are mapped between LotusScript and Java automatically.

LotusScript data type	*Is mapped to Java data type*
Byte	byte
Boolean	boolean
Integer	short
Long	int

continued

LotusScript data type	*Is mapped to Java data type*
String	These three Java data types map to a LotusScript string: • Java char (maps to a LotusScript String of length 1) • Java char array • java.lang.String From LotusScript to Java, the mapping depends on what the Java code expects as a type. See the example in the String Mapping example.
Single	float
Double	double long Which data type is used depends on what the Java code expects as a type.

A Variant should map to whatever data type it contains.

Note The Java byte type is signed (range -128 to +127), but the LotusScript Byte type is unsigned (range 0 to +255).

Java byte values of -128 to -1 map to LotusScript Byte values of +128 to +255. Java byte values of 0 to +127 map to the same LotusScript values, 0 to +127.

Java byte value	*LotusScript Byte value*
-128	+128
-127	+129
-126	+130
...	...
-2	+254
-1	+255
0	0
1	1
...	...
+126	+126
+127	+127

Note You can use the LotusScript data type in place of the Java data type for Get/Set properties, arguments for Java methods, and return values.

About Java precision and the long data type

The Java long data type range is:

min -2^63 == -9,223,372,036,854,775,808 == approx. -9.22337203685478E+18

max +2^63 - 1 == +9,223,372,036,854,775,807 == approx. +9.22337203685478E+18

However, because of a lack of precision in floating-point types, LS2J supports only a smaller range of approximately:

+- 9,223,372,036,854,770,000 == +- 9.22337203685477E+18

This range varies slightly by platform. LS2J throws an "Expression out of range" error if a LotusScript value outside these limits is passed to a Java long data type.

Even within the supported range, only 15 digits of precision are available; that is, a Java long data type will map to a predictable integral LotusScript value only within the range:

+- 1,000,000,000,000,000 ==+- 1.0E+15

String mapping example

LSStrings.java:

```
public class LSStrings
{
  public char    F1;
  public char [] F2;
  public String F3;
  public char    M1(char p)    { return p; }
  public char [] M2(char [] p) { return p; }
  public String M3(String p) { return p; }
}
```

LSStrings.lss:

```
Option Public
Uselsx "*javacon"
Dim mySession As JavaSession
Sub Initialize
```

```
  Dim myClass As JavaClass, myObject As JavaObject, _
s1 As String, s2 As String, s3 As String
  s1 = "A"
  s2 = "BC"
  s3 = "DE"
  Set mySession = New JavaSession ("\LSI\test\java;")
  Set myClass = mySession.GetClass("LSStrings;")
  Set myObject = myClass.CreateObject
  myObject.F1 = s1
  myObject.F2 = s2
  myObject.F3 = s3
  MsgBox myObject.F1 & myObject.F2 & myObject.F3 & _
myObject.M1(s1) _
        & myObject.M2(s2) & myObject.M3(s3)
  ' Displays "ABCDEABCDE"
End Sub
```

Java reference types

The Java reference types have limited support:

- The JavaObject data type is mapped directly and dynamically into a LotusScript ADT. You can use a LotusScript JavaObject in places where a Java object is needed: for example, in Get/Set properties, arguments for JavaMethods, and return values.
- LotusScript is only able to handle single dimension arrays of all the primitive types (byte, short, int, long, float, double, Boolean, and char). The Java char[ ] is mapped to the LotusScript dynamic string type. Notice that the Java/lang/String class is mapped to a LotusScript ADT. The following statement prints the actual text string out, assuming that myObject has a toString method, which returns a Java/lang/String object:

  ```
  print myObject.toString().toCharArray()
  ```

  LS2J dynamically adapts the Java/lang/String class then binds to the toCharArray method. The toCharArray method returns a char[], which is automatically translated into a LotusScript string.

Processing arguments

You can pass all primitive types and Java objects directly as arguments to JavaMethods. For reference types, LotusScript does not yet support the call-by-reference semantics. You can pass single dimension arrays into a Java method, but the results are not copied back into the LotusScript space. LotusScript also does not yet support passing in arrays of Java objects.

Limitations

Some important limitations include:

- You can't bring a bitmap into LotusScript because the Java byte (signed 8-bit) data type is mapped to LotusScript integer.
- You can't bring an integer greater than 32 bits into LotusScript without losing precision because the Java long (64 bit) data type is mapped to LotusScript long.
- You can use only single dimension arrays.
- There is no call-by-reference semantics for arguments of reference type.
- No Java object can be passed as an argument or a return value.

LS2J extended example

This sample code demonstrates calling Java methods within LotusScript using LS2J. Immediately following this example the code is modified slightly to demonstrate handling situations where there is a syntax discrepancy between Java and LotusScript.

Mortgage calculator

With this mortgage calculator you can get the value of a Java property and call several Java methods by using the names directly from LotusScript.

Mortgage.java

```
public class Mortgage

{

   // Java property

   public static String F = "Mortgage Calculator";

   // Java methods

   public double CalculateInterest(float rate, short yr,

      double principal)

   {
```

```
        // This bank doesn't bother with compound interest!
        return (rate / 100) * yr * principal;
    }
    public double CalculateTotal(double principal, _
double interest)
    {
        return principal + interest;
    }
    public double CalculateMonthlyPayment(float rate, _
short yr, double principal)
    {
      double interest = CalculateInterest(rate, yr, _
principal);
      double total = CalculateTotal(principal, interest);
      return total / (yr * 12);
    }
}
```

Mortgage.lss

```
Uselsx "*javacon"

Dim mySession As JavaSession

Sub Initialize
      Dim myClass As JavaClass
      Dim myObject As JavaObject
      Dim header As String
      Dim rate As Single
      Dim yr As Integer
      Dim principal As Double, interest As Double
      Dim total As Double, monthly_payment As Double
      Set mySession = New JavaSession()
      ' Set LS values
      rate       = 8.5
      yr               = 30
      principal  = 200000
```

```
      Set mySession = New JavaSession()
      ' Get Java "Mortgage" class
      Set myClass = mySession.GetClass("Mortgage;")
      ' Call Java "Mortgage" Constructor
      Set myObject = myClass.CreateObject
      ' Get Java property (can just use the name)
      header = myObject.F
      ' Call Java Methods (can just use the names)
      interest = myObject.CalculateInterest(rate, yr, _
principal)
      total = myObject.CalculateTotal(principal, interest)
      monthly_payment = _
myObject.CalculateMonthlyPayment(rate, yr, principal)
      MsgBox { } & header & {
            Interest rate:   } & rate & {%
            Years:            } & 30 & {
            Principal:       $} & principal & {
            Interest:        $} & Round(interest, 2) & {
            Total:            $} & Round(total, 2) & {
            Monthly payment: $} & Round(monthly_payment, 2)
End Sub
```

Mortgage calculator (version 2)

In this example, a few problems with syntax have been introduced:

- Two Java property names and two Java method names differ only by case.
- Two other Java methods are overloaded, differing only by parameter type.
- One Java method is over 40 characters long.

These types of syntax are allowed in Java but not in LotusScript. "Mortgage2.lss" shows how to use LS2J in this situation. At the end is an example of trapping a JavaError.

Mortgage2.java

```
public class Mortgage2
{
   // Java properties with names that differ only by case
   public static String F = "Mortgage Calculator";
   public static String f = "from your friendly neighborhood
bank";
   // Java methods
   // Two Java methods with names that differ only by case
   public double calculateinterest(float rate, short yr,
      double principal)
   {
         return rate * yr * principal;
      }
   public double CalculateInterest(float rate, short yr,
      double principal)
   {
         return calculateinterest(rate/100, yr, principal);
   }
   // Method with a long name
   public double

CalculateTotal_with_a_method_name_over_40_characters_long
      (double principal, double interest)
   {
         return principal + interest;
   }
   // Two Java overloaded methods -- differ only by parameter
types
   public double CalculateMonthlyPayment(double total, short
yr)
   {
         return total / (yr * 12);
```

```
   }
   public double CalculateMonthlyPayment(float rate, _
short yr,  double principal)
   {
       double interest = CalculateInterest(rate, yr, _
principal);
       double total =

CalculateTotal_with_a_method_name_over_40_characters_long
            (principal, interest);
       return CalculateMonthlyPayment(total, yr);
   }
}
```

Mortgage2.lss

```
Uselsx "*javacon"
Dim mySession As JavaSession
Sub Initialize
      Dim myClass As JavaClass
      Dim myObject As JavaObject

      Dim myProperty As JavaProperty
      Dim myMethod As JavaMethod
      Dim myError As JavaError
      Dim header1 As String, header2 As String
      Dim rate As Single
      Dim yr As Integer
      Dim principal As Double, interest As Double
      Dim total As Double, monthly_payment As Double
      Set mySession = New JavaSession()
      ' Set LS values
      rate       = 8.5
      yr               = 30
      principal  = 200000
```

```
      Set mySession = New JavaSession()
      Set myClass = mySession.GetClass("Mortgage2;")
      ' Call "Mortgage2" constructor
      Set myObject = myClass.CreateObject
      ' Use GetProperty()/GetValue() syntax when
      ' property names differ only by case
      Set myProperty = myClass.GetProperty("F")
      header1 = myProperty.GetValue()
      Set myProperty = myClass.GetProperty("f")
      header2 = myProperty.GetValue()
      ' Use GetMethod()/Invoke() syntax when method
      ' names differ only by case
      Set myMethod = _
myClass.GetMethod("CalculateInterest", "(FSD)D")
      interest = myMethod.Invoke(myObject, rate, yr, _
principal)
      ' or when method name is over 40 characters long
      Set myMethod = myClass.GetMethod  _

("CalculateTotal_with_a_method_name_over_40_characters_long",
_
            "(DD)D")
      total = myMethod.Invoke(myObject, principal, interest)
      ' or for overloaded Methods (differing only by parameter
type)
      Set myMethod = _
myClass.GetMethod("CalculateMonthlyPayment", "(DS)D")
      monthly_payment = _
myObject.CalculateMonthlyPayment(total, yr)
      MsgBox { } & header1 & {
            } & header2 & {
            Interest rate:   }   & rate & {%
            Years:            } & 30 & {
            Principal:       $} & principal & {
```

```
            Interest:        $} & Round(interest, 2) & {
            Total:            $} & Round(total, 2) & {
            Monthly payment: $} & Round(monthly_payment, 2)
      On Error GoTo errhandler
      ' Throws "java.lang.NoSuchMethodException"
      ' when user tries to get a method that doesn't exist
      Set myMethod = _
myClass.GetMethod("CalculateTax", _
"(D)D") done:
      Exit Sub
errhandler:
      Set myError = mySession.GetLastJavaError()
      MsgBox "JavaError was " & myError.errorMsg
      ' Clear the Error
      mySession.ClearJavaError
      Resume done
End Sub
```

Chapter 12
LotusScript Language Reference

This chapter describes the use of statements, built-in functions, subs, data types, and directives in the LotusScript language.

Abs function

Returns the absolute value of a numeric expression.

Syntax
Abs (*numExpr* **)**

Elements
numExpr
Any numeric expression.

Return value
Abs returns the absolute value of *numExpr*.

The data type of the return value is the same as the data type of *numExpr*, unless *numExpr* is a Variant. In that case, the following rules apply:

- If *numExpr* contains a string that LotusScript can convert to a number, the data type is Double.
- If *numExpr* contains a value that LotusScript cannot convert to a number, the function raises a type-mismatch error.
- If *numExpr* contains a NULL, the return value is NULL.

Usage
The absolute value of a number is its unsigned magnitude; for example, 3 and -3 both have an absolute value of 3.

Language cross-reference
@Abs function in formula language

Examples: Abs function

```
Print Abs(12)                          ' Prints 12
Print Abs(-12)                         ' Prints 12
Print Abs(13 - 25)                     ' Prints 12
Print TypeName(Abs(-12))               ' Prints INTEGER

Dim someV As Variant
someV = "123"
Print Abs(someV)                       ' Prints 123
someV = NULL
Print Abs(someV)                       ' Prints #NULL#
```

ACos function

Returns the arccosine, in radians, of a number between -1 and 1, inclusive.

Syntax

ACos (*numExpr* **)**

Elements

numExpr

A numeric expression with a value between -1 and 1, inclusive.

Return value

ACos returns the arccosine, in radians, of the value of *numExpr.*

The range of the return value is zero to pi, inclusive.

The data type of the return value is Double.

If the value of *numExpr* is not in the range -1 to 1, inclusive, the function raises an error.

Usage

The arccosine of a number is the angle, in radians, whose cosine is equal to the value of that number.

Language cross-reference

@Acos function in formula language

Examples: ACos function

```
Dim rad As Double
Dim degrees As Double

' Assign the value PI/2, the angle whose cosine is 0.
rad# = ACos(0)
```

```
' Assign the value 90, the same angle in degrees.
degrees# = rad# * (180 / PI)

Print rad#; degrees#          ' Prints 1.5707963267949  90
```

ActivateApp statement

Makes a program window the active window.

Syntax

ActivateApp *windowName*

AppActivate is acceptable in place of ActivateApp.

Elements

windowName

A string expression designating the program window to activate.

Usage

windowName is not case sensitive. It must exactly match the leftmost characters of the program title that appears in the program window title bar. For example, if the program title of a running program window is "Lotus Notes - Workspace," then a *windowName* value of "Lotus Notes" will activate that window. If more than one program title matches *windowName*, LotusScript will choose one of the program windows.

ActivateApp can activate a minimized window, but cannot restore or maximize it. Use SendKeys to restore or maximize a window. Use Shell to start a program.

Examples: ActivateApp statement

```
' Activate the Lotus Notes program window (assuming that
' Lotus Notes is already running). This would match a window
' with the title "Lotus Notes - Workspace".
ActivateApp "Lotus Notes"
```

ArrayAppend function

Appends one array to the end of another array and returns the result as a third array.

Syntax

ArrayAppend(*sourceArray1, sourceArray2* **)**

Elements

sourceArray1

Any variant containing an array.

sourceArray2

Any variant containing an array.

Return value

A variant containing an array.

Usage

During this operation *sourceArray1* and *sourceArray2* are not modified. If the two arrays are arrays of matching types, the returned array will be of that type. Otherwise, the returned array will be an array of Variants. The lower bound of the returned array is the same as the lower bound of *sourceArray1*, and the upper bound is the combined total of *sourceArray1* and *sourceArray2*.

For example:

sourceArray1(1 to 5) = [1,2,3,4,5]

sourceArray2(1 to 8) = [1,3,6,9,12,15,18,21)

returned array (1 to 13) = [1,2,3,4,5,1,3,6,9,12,15,18,21)

Error handling

ArrayAppend throws a Type mismatch error if:

- *sourceArray1* is not an array
- An array with more than one dimension is used

ArrayAppend throws a Subscript out of range error if the array bounds of the constructed array are outside acceptable array limits.

Extended examples: array and String functions

Extended examples: array and String functions

This program illustrates the functionality and usage of the following LotusScript array functions: ArrayAppend, ArrayGetIndex, ArrayReplace, FullTrim; and the following String functions: StrLeft, StrRight, StrLeftBack, StrRightBack.

The important code is in the two routines, ArrayExamples and AtComputeStrings. The rest of it consists of declarations and initialization. The generated output from the code is also listed below.

```
StringExample:

Option Public
Option Base 1

Dim arr1(8) As String
Dim arr2(8) As String
Dim arr3
Dim arr4(8) As Integer
Dim tarray1(10) As Integer
Dim tarray2(10) As Integer
Dim tarray3(10) As Integer
Dim i As Integer, x As Integer
Dim ans As String
Dim Indexresult
Dim localarray
Dim arresult

Sub Initialize

   ' arr1 will contain the following names
   arr1(1) = "Daniel"
   arr1(2) =  "Nate"
   arr1(3) =  "Joshua"
   arr1(4) =  "Sam"
   arr1(5) =  "Benjamin"
   arr1(6) =  "Julie "
   arr1(7) =  "Lauren "
   arr1(8) =  "Scrubbles"

   ' arr2 will contain "Joe1", "Joe2", etc
   ' arr4 will contain integers, with all even
   ' entries being zero

   For i = 1 To 8
      arr2(i) = "Joe" & i
      If (i Mod 2) = 0 Then
         arr4(i) = 0
      Else
         arr4(i) = i
```

```
      End If
   Next

   ' Initialize the arrays
   ' tarray1 will contain (1,2,3,4,5,6,7,8,9,10)
   x = 1
   For i =1 To 20
      If i  =< 10 Then
         tarray1(i) = i
      End If

   ' tarray2 will contain (2,4,6,8,10,12,14,16,18,20)
      If (i Mod 2) = 0 Then
         tarray2(x) = i
         x = x+1
      End If
   Next

   ' tarray3 will contain the following (8,6,4,2,25,0,0,0,0,0)
   tarray3(1) =  8
   tarray3(2) =  6
   tarray3(3) =  4
   tarray3(4) =  2
   tarray3(5) =  25

   ' Run the examples
   ArrayExamples
   AtComputeStrings

End Sub

Sub Arrayexamples

   ' Arrayappend populates arr3 with all elements of arr1
   ' and all elements of arr2,  arr3 lower bound is 1
   ' its upper bound is 16
   Print "Arrayappend results:"
   arr3 =  Arrayappend (arr1, arr2)
   Print "    arr3 contains: ", arr3(1), arr3(2), "..." , _
      arr3(15), arr3(16)
   Print "    Up/Low bounds for arr3: " & Lbound(arr3) & _
      " & " & Ubound(arr3)

   ' Araygetindex example
   value = "Benjamin"
   Indexresult = Arraygetindex(arr1,value)
   Print "Araygetindes results:"
   Print "    Arraygetindex(arr1,value) returns ";_
     Indexresult
```

```
   Indexresult = Arraygetindex(arr1,"Scrubbles")
   Print "    Arraygetindex(arr1,""Scrubbles"") returns ";
Indexresult

   ' Fulltrim of an array
   Print "Arraygetindex on fulltrimed array results:"
   localarray = Fulltrim(arr4) 'localarray = [1, 3, 5, 7]
   Indexresult = Arraygetindex(localarray, 3)
   Print  "    Arraygetindex(localarray, 3) returns ";
Indexresult

   'Fulltrim of a string
   Print "Fulltrim of string:"
   qbf_spaces = "       The       quick brown     fox" & _
     "    jumped      over the    lazy    dog.      "

   Print "   ", qbf_spaces
   qbf_trimed = Fulltrim(qbf_spaces)
   Print "   ", qbf_trimed

   ' Arrayreplace example

   Print "Arrayreplace results:"

   'Expected answer is "1 8 3 6 5 4 7 2 9 25"
   arresult = Arrayreplace( tarray1, tarray2, tarray3)

   ' Generate string that represent the contents of arresult
   msg1 = ""
   For i = 1 To 10
      msg1 = msg1 & " " & arresult(i)
   Next
   Print "    arresult = " & msg1

End Sub

Sub  AtComputeStrings()
   Dim s1 As String
   Dim s2 As String
   Dim v1 As Variant

   s1 = "The quick brown FOX jumps over the lazy dog."
   s2 = "he"

   Print " "
   Print "Results for Strleft, strright, strrightback," & _
     " Strleftback"
   ' Search left to right
   ' Strleft: expect v1 = "T"
   v1 = Strleft( s1, s2 )
   Print "    " + v1

   ' Strright:
   ' expect v1 = " quick brown FOX jumps over the lazy dog."
```

```
   v1 = Strright( s1,s2 )
   Print "    " + v1

   ' Search right to left
   ' Strleftback:
   ' expect v1 = "The quick brown FOX jumps over t"
   v1 = Strleftback( s1, s2 )
   Print "    " + v1

   ' Strrightback: expect v1 = " lazy dog."
   v1 = Strrightback( s1, s2 )
   Print "    " + v1

   ' With some optionals.....
   s1 = "The quick brown FOX jumps over the lazy dog."
   s2 = "o"
   ' the letter o  CHANGED S2, pattern searched for HERE.

   ' A case INsensitive search, it finds the second
   ' occurrence of 'o' and returns what is to the left of
that.
   ' expect v1 = "The quick brown F "
   v1 = Strleft( s1,s2, 5, 2  )
   Print "    " + v1

   ' A Case sensitive search,  Finds the third occurrence of
   ' 'o'  and returns what is to the RIGHT of that.
   ' expect  v1 = "ver the lazy dog."
   v1 = Strright( s1,s2, 0, 3 )
   Print "    " + v1

   s2 = "O"
   ' A case sensitive search. Expect v1 = "The quick brown F"
   v1 = Strleftback( s1,s2,0 )
   Print "    " + v1

   ' A case sensitive search, with a skip first occurrence,
   ' O in FOX is Skipped and  no other occurrence exists,
   ' expect v1 = ""
   v1 = Strleft( s1,s2,0,2)
   Print "    " + v1

End Sub
```

The results of this program are:

ArrayAppend results:

```
arr3 contains:   Daniel, Nate, ... Joe7, Joe8
Up/Low bounds for arr3: 1 & 16
```

ArrayGetIndex results:

```
ArrayGetIndex(arr1,value) returns  5
Arraygetindex(arr1,"Scrubbles") returns  8
```

ArrayGetIndex on fulltrimed array results:

```
Arraygetindex(localarray, 3) returns  2
```

FullTrim of string:

The quick brown fox jumped over the lazy dog.

```
The quick brown fox jumped over the lazy dog.
```

ArrayReplace results:

```
arresult =  1 8 3 6 5 4 7 2 9 25
```

Results for Strleft, strright, strrightback, Strleftback

```
T
  quick brown FOX jumps over the lazy dog.
 The quick brown FOX jumps over t
  lazy dog.
 The quick brown F
 g.
 The quick brown F
```

ArrayGetIndex function

Searches an array of strings for the value given. If the value is found within the array, the array index of that value is returned.

Syntax

ArrayGetIndex(*sourceArray, searchValue [, compMethod]* **)**

Elements

sourceArray

An array or Variant containing an array.

searchValue

A value to search for within sourceArray.

compMethod

Optional integer specifying the type of comparison to use when searching for searchValue.

Number	*Comparison Mode*
0	case sensitive, pitch sensitive
1	case insensitive, pitch sensitive
4	case sensitive, pitch insensitive
5	case insensitive, pitch insensitive

Return value

A Variant of type long that provides the index into *sourceArray* where *searchValue* can be found. If no match is found, NULL is returned.

Usage

ArrayGetIndex converts all values passed to it into strings. For example, if you pass an array of integers, this function converts the values in the array to strings *for this operation only*. These string values are then used for comparing the array values to the *searchValue*. Option Compare can be used to specify whether case/pitch sensitivity should play a role in the comparisons. If *compMethod* is not specified, the default for the module is used.

Items that cannot be converted are not compared.

Extended examples: array and String functions

ArrayReplace function

Performs a search and replace routine for multiple values within an array.

Syntax

ArrayReplace(*sourceArray*, *compareArray*, *replaceArray* **)**

Elements

sourceArray

The source array from which a copy, with possible modifications, will be produced.

compareArray

An array containing the elements to be compared to the elements in *sourceArray* (can be a scalar which is treated as a single-element array).

replaceArray

An array containing the elements to be used to replace elements from *sourceArray* (can be a scalar which is treated as a single-element array).

Return value

A Variant containing an array which is constructed by these rules (the answer array).

Usage

Each element in *sourceArray* is prepared to be copied into the answer array. The resulting array is the same size as the array contained in parameter *sourceArray*. If the source and replace arrays are arrays of matching types, the answer array will be of that type. Otherwise, the answer array will be an array of Variants.

Note ArrayReplace works only with these LotusScript scalar data types: integer, long, single, double, currency, string, boolean, and byte. If any other data type is used in the *sourceArray* or the *replaceArray*, the resulting array contains the exact same data elements as the *sourceArray*; that is, no replacement of array elements occurs.

For each element in *sourceArray*, *compareArray* is scanned. If no elements match, the element from *sourceArray* is copied into the next available index in the answer array. However, if an element of *compareArray* matches an element from *sourceArray*, the index of the *compareArray* element is used to find a value in the array *replaceArray*. This value is then copied into the answer array instead of the value from *sourceArray*.

For example:

sourceArray = [1,2,3,4,5]

compareArray = [2,4,6,8,10,12,14,16,18,20]

replaceArray = [8,6,25,0,0,11,17]

1. Element 1 from *sourceArray* is compared to the elements in *compareArray*. Since no match is found, the first element from *sourceArray* is copied into the answer array in the first element.

   answer array = [1,...]

2. Element 2 from *sourceArray* is compared to the elements in *compareArray*. The first element in *compareArray* matches the second element from *sourceArray*, so the index to the first element in *compareArray*, which is 1, is used to find a value in *replaceArray*, which is [8]. This value is then copied into the answer array.

   answer array = [1,8,...]

3. Element 3 from *sourceArray* is compared to the elements in *compareArray*. Since no match is found, the third element from *sourceArray* is copied into the answer array.

   answer array = [1,8,3,...]

4. Element 4 from *sourceArray* is compared to the elements in *compareArray*. The second element in *compareArray* matches the fourth element from *sourceArray*, so the index to the second element in *compareArray*, which is 2, is used to find a value in *replaceArray*, which is [6]. This value is then copied into the answer array.

   answer array = [1,8,3,6,...]

5. The last element from *sourceArray* is compared to the elements in *compareArray*. Since no match is found, the fifth element from *sourceArray* is copied into the answer array.

   answer array = [1,8,3,6,5]

If the index from *compareArray* cannot be used as an index into *replaceArray* (that is, the index is out of bounds), a 0 or type equivalent is copied into the answer array for that element.

Indices into the arrays are calculated from their base. Assume that *compareArray* is an array from (-10 to 0), and *replaceArray* is an array from (1 to 5). If the -10th element of *compareArray*, which is the first element in that array, is a match for a given element in *sourceArray*, then the first element of *replaceArray* is used as a replacement.

For example:

sourceArray(1 to 10) =
[the,quick,sleek,cat,jumped,over,the,fat,sleeping,dog]

compareArray(-10 to 0) =
[sleek,cat,jumped,fat,sleeping,under,ball,purple,tree,slow,over]

replaceArray(1 to 5) = [red,fox,hurdled,lazy,brown]

1. The first element in *sourceArray* is compared to the elements in *compareArray*. No match is found, so the first element from *souceArray* is copied to the answer array.

   answer array=[the,...]

2. The second element in *sourceArray* is compared to the elements in *compareArray*. No match is found, so the first element from *souceArray* is copied to the answer array.

   answer array=[the,quick,...]

3. The third element in *sourceArray* is compared to the elements in *compareArray*. A match is found at the first element of *compareArray*, but rather than trying to access the -10th index of *replaceArray*, which would be invalid, instead the equivalent index of the matching element of *compareArray* is calculated for *replaceArray*. As

a result, the first element in *replaceArray* is then copied into the answer array.

answer array=[the,quick,red...]

and so on.

Note that the 0th element of *compareArray* is a match for an element in *sourceArray*. Since this translates to 11 for *replaceArray*, which is out of bounds, a null value is used for the replacement value instead.

answer array=[the,quick,red,fox,hurdled,{null},...]

In this way "the quick sleek cat jumped over the fat sleeping dog" becomes "the quick red fox hurdled the lazy brown dog."

Each element type must match for a conversion to take place. For example, if *sourceArray* contains the value 1 of data type integer, and *compareArray* contains the value 1 of data type long, then these elements would **not** match.

Extended examples: array and String functions

ArrayUnique function

Removes duplicate elements from an Array.

Syntax

ArrayUnique(*sourceArray* **[**,*compMethod* **])**

Elements

sourceArray

Array of any type.

compMethod

Optional Integer specifying the type of comparison to use when searching for the delimiter, if the elements are strings.

Number	*Comparison Mode*
0	case sensitive, pitch sensitive
1	case insensitive, pitch sensitive
4	case sensitive, pitch insensitive
5	case insensitive, pitch insensitive

If you omit *compMethod*, the default comparison mode is the mode set by the Option Compare statement for this module. If there is no statement for the module, the default is case sensitive and pitch sensitive.

Return value

Returns an array with duplicates removed. For any elements of the array which compare equal, the first occurrence is copied into the result array.

Usage

Elements in a variant array will only compare equal if they are of the same type. The variant array can't contain classes or objects.

Array elements that contain the null value will match other null values.

Array elements that are empty will match with other elements that are empty.

Error handling

ArrayUnique throws a Run-time Type mismatch if:

- The first parameter is not an array
- A list is passed instead of an array
- The array passed in has not been properly initialized
- The array is of classes
- The array is of NotesDocuments
- The array contains an array as an element
- The array contains nothing as an element

ArrayUnique throws a run-time Wrong Number of Dimensions error if the array is not one-dimensional.

Language cross-reference

@Unique function in formula language

Examples: ArrayUnique function

```
'Declare array of variants

Dim myTestarr(4) as variant

myTestArr(0) = "abc DEF Ghi"
myTestArr(1) = "ABC def gHi"
myTestArr(2) = "abc DEF Ghi"
myTestArr(3) = "ABC def gHi"
myTestArr(4) = "abc DEF Ghi"

Sub Initialize
   Dim resultArr as variant
   Dim count as integer

   ' use Comparison Method = 0 (case sensitive, pitch
sensitive)
```

```
   resultArr = arrayunique(myTestArr,0)
   for count = lbound(resultArr) To ubound(resultArr)
      Print resultArr(count)
   next count
End Sub

'Output:
'abc DEF Ghi
'ABC def gHi
```

Asc function

Returns the locale-sensitive ASCII character code for the first character in a string.

Syntax

Asc (*stringExpr* **)**

Elements

stringExpr

Any string expression.

Return value

Asc returns the locale-sensitive ASCII character code of the first character in *stringExpr*. If LotusScript is running on a native ASCII platform, the code represents the character value in the platform's native character set. If LotusScript is running on a native EBCDIC platform, the character is converted to its ASCII equivalent for the platform's current locale and that code is returned.

The data type of the return value is Long.

If the value of *stringExpr* is NULL or the empty string (""), the function raises an error.

Examples: Asc function

```
Dim bigA As Long
Dim littleA As Long
bigA& = Asc("A")
littleA& = Asc("a")
Print bigA&; littleA&                ' Prints 65  97
```

ASin function

Returns the arcsine, in radians, of a number between -1 and 1, inclusive.

Syntax
ASin (*numExpr* **)**

Elements
numExpr
A numeric expression with a value between -1 and 1, inclusive.

Return value
ASin returns the angle, in radians, whose sine is equal to the value of *numExpr.*

The range of the return value is -pi/2 to pi/2, inclusive.

The data type of the return value is Double.

If the value of *numExpr* is not in the range -1 to 1, inclusive, the function raises an error.

Language cross-reference
@ASin function in formula language

Examples: ASin function
```
Dim rad As Double
Dim degrees As Double

' Assign the value pi/2, the angle whose sine is 1.
rad# = ASin(1)

' Assign the value 90, the same angle in degrees.
degrees# = rad# * (180 / pi)

Print rad#, degrees#          ' Prints  1.5707963267949  90
```

ATn function

Returns the arctangent, in radians, of a number.

Syntax
ATn (*numExpr* **)**

Elements
numExpr
Any numeric expression.

Return value

ATn returns the angle, in radians, whose tangent is equal to the value of *numExpr*.

The range of the return value is -pi/2 (-90 degrees) to pi/2 (90 degrees), exclusive.

The data type of the return value is Double.

Language cross-reference

@ATan function in formula language

Examples: ATn function

```
Dim rad As Double
Dim degrees As Double

' Assign the value pi/4, the angle whose tangent is 1.
rad# = ATn(1)

' Assign the value 45, the same angle in degrees.
degrees# = rad# * (180 / pi)

Print rad#; degrees#         ' Prints .785398163397449  45
```

ATn2 function

Returns the polar coordinate angle, in radians, of a point in the Cartesian plane.

Syntax

ATn2 (*numExprX* **,** *numExprY* **)**

Elements

numExprX, numExprY

Any numeric expressions. At least one of the two must be non-zero. *numExprX* and *numExprY* designate the coordinates of a point in the Cartesian plane.

Return value

ATn2 returns the angular portion of the polar coordinate representation of the point *(numExprX, numExprY)* in the Cartesian plane.

The range of the return value is -pi to pi, inclusively.

If *numExprX* is 0, then ATn2 returns one of the following values:

- -pi/2, if *numExprY* is negative
- pi/2, if *numExprY* is positive

If *numExprX* is positive, then ATn2(*numExprX, numExprY*) returns the same value as ATn(*numExprY / numExprX*).

Language cross-reference

@ATan2 function in formula language

Examples: ATn2 function

```
Dim quad1 As Double, quad2 As Double, _
   quad3 As Double, quad4 As Double

' Assign the arctangents of four points in the plane.
quad1# = ATn2(1, 1)
quad2# = ATn2(-1, 1)
quad3# = ATn2(-1, -1)
quad4# = ATn2(1, -1)

' Print the value each angle in degrees.
Print quad1# * (180 / pi)              ' Prints 45
Print quad2# * (180 / pi)              ' Prints 135
Print quad3# * (180 / pi)              ' Prints -135
Print quad4# * (180 / pi)              ' Prints -45
```

Beep statement

Generates a tone on the computer.

Syntax

Beep

Usage

The tone that LotusScript produces depends on the sound-generating hardware in your computer.

Examples: Beep statement

```
' While a user-specified interval (in seconds) elapses, beep
' and count the beeps. Then tell the user the number of beeps.

Dim howLong As Single, howManyBeeps As Integer
Function HowManyTimes (howLong As Single) As Integer
   Dim start As Single, finish As Single, counter As Integer
   start! = Timer
   finish! = start! + howLong!
   While Timer < finish!
```

```
      Beep
      counter% = counter% + 1
   Wend
   HowManyTimes% = counter%
End Function
howLong! = CSng(InputBox _
   ("For your own sake, enter a small number."))
howManyBeeps% = HowManyTimes(howLong!)
MessageBox "Number of beeps:" & Str(howManyBeeps%)
```

Bin function

Returns the binary representation of a number as a string.

Syntax

Bin[$] (*numExpr* **)**

Elements

numExpr

Any numeric expression. If *numExpr* evaluates to a number with a fractional part, LotusScript rounds it to the nearest integer before deriving its binary representation.

Return value

Bin returns a Variant of DataType 8 (String), and Bin$ returns a String.

Return values will only include the characters 0 and 1. The maximum length of the return value is 32 characters.

Usage

If the data type of *numExpr* is not Integer or Long, then LotusScript attempts to convert it to a Long. If it cannot be converted, a type mismatch error occurs.

Examples: Bin function

```
Print Bin$(3)                 ' Prints "11"

' Converts Double argument to Long.
Print Bin$(3.0)               ' Prints "11"

' Rounds Double argument, then converts to Long.
Print Bin$(3.3)               ' Prints "11"

' Computes product 2.79, rounds to 3.0, then converts to Long.
Print Bin$(3.1 * .9)          ' Prints "11"
```

Boolean data type

Specifies a variable that contains a True (-1) or False (0) value.

Usage

A Boolean value is one that contains the value of True or False only. Boolean values are stored as 16-bit (2-byte) numbers. When Boolean values are converted to numeric data types, True becomes -1 and False becomes 0. When other numeric data types are converted to the Boolean data type, 0 becomes False and any other value becomes True.

Boolean variables are initialized to False.

There is no suffix character for the Boolean data type.

When printed, a variable of Boolean data type displays as either True or False; when Write # is used, the variable is displayed as either #TRUE# or #FALSE#.

Examples: Boolean data type

```
dim x

dim y As Boolean, z As Boolean

x = 1 > 2        ' the expression 1 > 2 evaluates to False, so
x is

                      ' assigned a value of False and data
type Boolean

Print x               ' Output: False

Print TypeName(x)     ' Output: BOOLEAN

Print DataType(x)     ' Output: 11

x = True

Print x               ' Output: True

Print CInt(x)         ' Output: -1

Print x + 2      ' Output: 1

Print x + " Blue"     ' Output: True Blue

y = True

z = 0

x = y AND z

Print x               ' Output: False
```

Bracket notation

For applications developed with some Lotus products, such as 1-2-3®, you can use names in brackets rather than object reference variables to identify Lotus software objects. To determine whether your Lotus software supports this notation, see the product documentation.

Syntax

[*prodObjName*]

Elements

prodObjName

The name understood by the product to identify an object (an instance of a product class).

Usage

In some cases, Lotus products assign names to objects, and in other cases you can use the product user interface to name the objects you create. In a spreadsheet, for example, A1 identifies a particular cell, and you could use the user interface to name a chart SalesTracking.

Bracket notation lets you use these names without declaring an object variable and binding it to the object. For example, the product might allow you to use:

```
[A1].contents = Cstr(247000)
```

instead of:

```
Dim myCell as Range
Set myCell = Bind("A1")
mycell.contents = Cstr(247000)
```

In some cases, the product uses bracket notation when it records transcripts of user actions. This makes the transcripts easier to read and modify. For more information, see the product documentation.

The LotusScript compiler does not attempt to determine the class of objects that are identified with bracket notation, so any class syntax errors you make (such as the incorrect use of properties and other methods), will generate run-time errors, not compile-time errors.

You can also use empty brackets to identify the currently selected product object. Empty brackets are equivalent to leading dot notation. For example, if the current selection is a range named Sales, then

```
[ ].CopyToClipboard
```

and

```
.CopyToClipboard
```

are equivalent to

```
[Sales].CopyToClipboard
```

All three statements copy the contents of the Sales range to the clipboard.

To include square brackets as text within a string, double the brackets. For example, if the current selection is a range named Sales[East], use the following syntax:

```
[Sales[[East]]].CopyToClipboard
```

Examples: Bracket notation

```
' Using the Chart class Print method, print chart
SalesTracking
[SalesTracking].Print
```

Byte data type

Specifies a variable that contains a single, one-byte unsigned number.

Usage

A Byte value is a positive integer in the range 0 to 255, inclusive, stored as a single, 8-bit (1-byte) unsigned number.

Byte variables are initialized to 0.

There is no suffix character for the Byte data type.

A byte type can be used anywhere an integer type can be used. The baseline specification for the Byte data type is the same as the byte data type in Visual Basic.

Byte is both a value and a data type. This means a byte value can be stored in either a variable declared as the Byte data type or a variable declared as a variant. Because a value retrieved from a variant may be significant, both cases must be tested for.

Examples: Byte data type

Example 1

```
' The variables count and nextNum are explicitly declared
' as type Byte. There is no suffix character for Byte, so a
' variable of type Byte cannot be declared implicitly.
Dim count as Byte
```

```
Dim nextNum as Byte
count = 1
nextNum = count + 1
Print count; nextNum  ' Output:      1   2
```

Example 2

```
' Use Byte data type to retrieve single byte from a file
Dim b As Byte
Dim FF As Integer
FF = Freefile
Open "myfile.data" For Binary Access Read As ff
While (Not Eof(ff))
      Get #ff, ,b
Wend
Close #ff
```

Call statement

Calls a LotusScript sub or function.

Syntax 1

Call *subOrFunction* [**(** [*argList*] **)**]

Syntax 2

subOrFunction [*argList*]

Syntax 3

subOrFunction **(** *argPassedByVal* **)**

Syntax 4 (functions only)

returnVal = *function* [**(** [*argList*] **)**]

Elements

subOrFunction

The name of the sub or function being called.

argList

A list of arguments, separated by commas, for the sub or function being called.

argPassedByVal

A single argument to be passed by value to the sub or function being called.

function

The name of the function being called.

returnVal

The assignment variable containing the function's return value.

Usage

When you use the Call keyword, you must include parentheses around the argument list. If there are no arguments, the empty parentheses are optional.

When you omit the Call keyword, the following parenthesis rules apply:

- For a sub or a function, do not use parentheses around the argument list (Syntax 2) unless you are passing a single argument by value to the sub or function (Syntax 3).
- For a function within an expression, enclose the argument list (if there is one) in parentheses (Syntax 4).

Sub calls do not return a value.

LotusScript uses a function's return value if the function call appears in an expression. The call can appear anywhere in an expression where the data type of the function's return value is legal. Function calls that use the Call keyword, however, do not return a value and cannot appear in an expression.

LotusScript always uses the return value of a call to a built-in function. You must use its return value in an expression, and you cannot use the Call keyword.

Referencing a function that returns an array, list, or collection

If a function returns an array, list, or collection, a reference to the function can contain subscripts according to the following rules:

- If the function has parameters, the first parenthesized list following the reference must be the argument list. A second parenthesized list is treated as a subscript list. For example, f1(1,2)(3) is a reference to a function f1 that has two parameters and returns a container.
- If the function has no parameters and the return type is a variant or collection object, two parenthesized lists, but not one, can follow the reference. The first must be empty and the second is treated as a subscript list. For example, f1()(3) is a reference to a function f1 that contains no parameters but is a container.

- If the function has no parameters and the return type is not a variant or collection object, any parenthesized list following the reference is an error, except that a single empty list is allowed. For example, f1() is a reference to a function f1 that contains no parameters and may or may not be a container; if f1 is a container, the reference is to the entire container.

Examples: Call statement

Example 1

```
' Define a function and then invoke it in three ways.

Function MiniMult (x As Integer, y As Integer) As Integer
   MiniMult = x% * y%
End Function
Dim result As Integer

Call MiniMult(3, 4)
       ' With Call; return value (12) is not used.
Call MiniMult 3, 4
       ' Without Call; return value is not used.
result% = MiniMult(3, 4) ' With Call; return value is used.
Print result             ' Prints 12.
```

Example 2

```
' Define a sub and then invoke it in two ways.
Sub PrintProduct (a As Integer, b As Integer)
   Print a% * b%
End Sub

Call PrintProduct(34, 5)    ' With Call; prints 170.
PrintProduct 34, 5          ' Without Call; prints 170.
```

CBool function

Returns an expression converted to the Boolean data type.

Syntax

CBool (*expr*)

Elements

expr

Any numeric expression, or the string expressions True and False.

Return value

CBool returns an expression that has been converted to a Variant of subtype Boolean.

CBool(EMPTY) returns 0 (False).

If *expr* is a numeric expression, CBool returns a variant containing the value True or False, depending on the value of the numeric expression: 0 becomes False, and any other value becomes True.

If *expr* lies outside the acceptable range for the Boolean subtype, the function raises an error.

Examples: CBool function

```
' Convert and display Integer and String values converted to
Boolean

dim Int_1 as integer

dim String_1 as string

dim Bool_1, Bool_2

Int_1 = 0

print CBool(Int_1)      'prints FALSE

Bool_1 = CBool(Int_1)

Int_1 = 99

print CBool(Int_1)      'prints TRUE

String_1 = "True"

print CBool(String_1)   'prints TRUE

Bool_2 = CBool(String_1)

String_1 = "No Value"

print CBool(String_1)   'Generates type mismatch error (Error
13)

                              'String value must be "True" or
"False" for

                              'successful conversion to type
Boolean

print DataType(Bool_1)
'prints 11 (Boolean)

print DataType(Bool_2)
'prints 11 (Boolean)
```

CByte function

Returns an expression converted to the Byte data type.

Syntax
CByte (*expr*)

Elements
expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value
CByte returns an expression that has been converted to a Variant of subtype Byte.

CByte(EMPTY) returns 0.

If *expr* is a string expression, CByte returns the numeric representation of the string, rounded to the nearest integer. If LotusScript cannot convert the string to a number, the function raises an error.

If *expr* lies outside the acceptable range for the Byte subtype, the function raises an error.

Examples: CByte function
```
'Convert an expression to a Byte

Dim MyDouble, MyByte

MyDouble = 125.5678      'MyDouble is a Double

MyByte = CByte(MyDouble) 'MyByte contains 126
```

CCur function

Returns a value converted to the Currency data type.

Syntax
CCur (*expr* **)**

Elements
expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value

CCur returns the numeric value of *expr* rounded to four decimal places, as a Currency value.

CCur(EMPTY) returns 0.

If *expr* is a string expression, CCur returns the numeric representation of the string, rounded to four decimal places. If LotusScript cannot convert the string to a number, the function raises an error.

If the value of *expr* is too large to fit in the Currency data type, the function raises an error.

Examples: CCur function

```
Dim bulkPrice As Double
Dim labelPrice As String
Dim unitsSold As Integer
Dim paymentDue As Currency

bulkPrice# = 11.400556
unitsSold% = 57
paymentDue@ = CCur(bulkPrice# * unitsSold%)
Print paymentDue@               ' Prints 649.8317

labelPrice$ = "12.99"
paymentDue@ = CCur(labelPrice$) * unitsSold%
Print paymentDue@               ' Prints 740.43
```

CDat function

Converts a numeric value or string value to a date/time value.

Syntax

CDat (*expr* **)**

CVDate is acceptable in place of CDat.

Elements

expr

Any of the following kinds of expression:

- A numeric expression
- A string expression that can be converted to a number
- A string expression that can be converted to a date/time value

Return value

CDat returns a date/time value.

The data type of the return value is a Variant of DataType 7 (Date/Time).

If the integer part of *expr* is not in the range -657434 to 2958465, the function raises an error.

CDat(0) returns the date/time value December 30, 1899, 12:00:00 AM, formatted as 12:00:00 AM. CDat(EMPTY) returns the same value.

Usage

CDat converts *expr* to a date/time value in the LotusScript date/time format.

CDat uses different conversion rules depending on the form of *expr*:

- If *expr* is a numeric expression, CDat converts the integer part of its value to a date and the fractional part to a time, and returns the corresponding date/time value.

  A date/time value stored in a Variant is an eight-byte floating-point value. The integer part represents a serial day counted from Jan 1, 100 AD. Valid dates are represented by integer numbers in the range -657434, representing Jan 1, 100 AD, to 2958465, representing Dec 31, 9999 AD. The fractional part represents the time as a fraction of a day, measured from time 00:00:00 (midnight on the previous day). In this representation of date/time values, day 1 is the date December 31, 1899.

- If *expr* is a string expression that can be converted to a number, CDat converts the string to a number and then converts the number to a date/time value and returns the result, as described in the previous bullet.

- If *expr* is a string expression in the form of a date, for example "8/20/98", CDat converts the value to a date/time in the internal date/time format.

If LotusScript cannot convert the value to a date/time, the function raises an error.

Language cross-reference

@Time function in formula language

@TextToTime function in formula language

Examples: CDat function

Here are two programming examples of the CDat function.

Example 1

```
Dim dateV As Variant

' Convert a numeric value to a date/time value.
dateV = CDat(34814.3289)

' Display the formatted date and time.
Print Format$(dateV, "Medium Date"), _
      Format$(dateV, "Medium Time")
' Prints 25-Apr-95  07:53 AM

' Convert the date back to a number.
Print CDbl(dateV)                ' Prints 34814.3289

' Convert a date string to a date.
Print CDat("April 25, 1995")      ' Prints 4/25/95
```

Example 2

```
print CDat(-1_,cdate(0), cdate(1)

'Output is 12/29/1899    12:00:00 AM    12/31/1899

print CDat("23:59:59"), cdat("00:00:00"), cdat("00:00:01")

'Output is 11:59:59 PM    12:00:00 AM   12:00:01 AM
```

CDbl function

Returns a value converted to the Double data type.

Syntax

CDbl (*expr* **)**

Elements

expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value

CDbl returns the numeric value of *expr* as a Double value.

CDbl(EMPTY) returns 0.

If *expr* is a string expression, CDbl returns the numeric representation of the string, including any fractional part. If LotusScript cannot convert the string to a number, the function raises an error.

If the value of *expr* is too large to fit in the Double data type, the function raises an error.

Language cross-reference

@TextToNumber function in formula language

Examples: CDbl function

```
' Convert the sum of two Single values to Double.
Dim x As Single
Dim y As Single
Dim result As Double
x! = 11.06E23
y! = 6.02E23
result# = CDbl(x! + y!)
Print result#              ' Prints 1.70800003057064E+24
```

ChDir statement

Sets the current directory.

Syntax

ChDir *path*

Elements

path

A string expression representing the path of an existing directory.

Usage

ChDir sets the current directory to *path*. The current directory is the directory that LotusScript uses when you specify a file name without a path.

If the value of *path* does not begin with a drive letter, ChDir sets the current directory for the current drive.

If the value of *path* includes a drive letter, ChDir sets the current directory for that drive, but does not reset the current drive. The path will not be used as the current directory until the current drive is reset. To change the current drive, use ChDrive.

To return the current drive, use CurDrive. To return the current directory, use CurDir.

The format and maximum length of *path* follow the conventions of the platform on which LotusScript is running.

Examples: ChDir statement

```
' Set the current drive to d.
ChDrive "d"

' Set current directory on the c drive to \test.
ChDir "c:\test"

' Set current directory on current drive (d) to \test.
ChDir "\test"

Print CurDir()                  ' Prints d:\test
```

ChDrive statement

Sets the current drive.

Syntax

ChDrive *drive*

Elements

drive

A string expression representing an existing drive.

Usage

ChDrive sets the current drive to the value of *drive.* The current drive is the drive that LotusScript uses whenever you specify a file name or a path that does not include a drive.

If the value of *drive* is the empty string (""), ChDrive does not change the current drive.

If the value of *drive* is a string of more than one character, ChDrive uses only the first character. ChDrive does not require a colon (`:`) after the drive letter.

The *drive* must be in the range A to *lastdrive*, inclusive, where *lastdrive* is the maximum drive letter specified in CONFIG.SYS.

To change the current directory, use ChDir.

To return the current drive, use CurDrive. To return the current directory, use CurDir.

Examples: ChDrive statement

```
' Set the current drive to D.
ChDrive "D"
```

Chr function

Returns the character represented by a value interpreted as a locale-sensitive character code.

Syntax
Chr[**$**] **(** *numExpr* **)**

Elements
numExpr

A numeric expression of data type Long in the range 0-255. If LotusScript is running on a native ASCII platform, the value is interpreted as a character code in the platform-native character set. If LotusScript is running on an EBCDIC platform, the value is interpreted as the character code for the ASCII equivalent in the platform's current locale.In either case, only single-byte ASCII values are valid.

Return value
Chr returns the character corresponding to the value of *numExpr*.

Chr returns the ANSI platform-specific character corresponding to the value of *numExpr*.

Chr returns a Variant of DataType 8 (String). Chr$ returns a String.

Usage
If the value of *numExpr* contains a fraction, LotusScript rounds the value before using it.

Examples: Chr function
```
Dim myAlph As String
Dim letterCode As Long
' Iterate through the character codes for "a" through "z".
' Build an alphabet string by concatenating the letters.
For letterCode& = Asc("a") To Asc("z")
   myAlph$ = myAlph$ & Chr$(letterCode&)
Next
Print myAlph$         ' Prints abcdefghijklmnopqrstuvwxyz
```

CInt function

Returns a value converted to the Integer data type.

Syntax
CInt (*expr* **)**

Elements

expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value

CInt returns the value of *expr* rounded to the nearest integer, as an Integer value.

CInt(EMPTY) returns 0.

If *expr* is a string expression, CInt returns the numeric representation of the string, rounded to the nearest integer. If LotusScript cannot convert the string to a number, the function returns an error.

If the value of *expr* is too large to fit in the Integer data type, the function raises an error.

Language cross-reference

@Integer function in formula language

@TextToNumber function in formula language

Examples: CInt function

```
' Convert a Currency value to Integer.
Dim x As Currency
x@ = 13.43
Print CInt(x@)                  ' Prints 13
```

Class statement

Defines a class with its member variables and procedures.

Syntax

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

classBody

End Class

Elements

Public | **Private**

Optional. Public specifies that the class is visible outside the module where the class is defined, as long as this module is loaded. Private specifies that the class is visible only in this module.

A class is Private by default.

className
The name of the class.

baseClass
Optional. The name of another class from which this class is derived.

classBody
Declarations and definitions of class members. Class members can include member variables; member procedures (functions, subs, and properties); a constructor sub, named New; and a destructor sub, named Delete. Constants cannot be class members.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

Rules for defining classes:

- Define a class only in module scope. Do not define a class within a procedure or within another class.
- Do not use the word Object as a class name.

Rules for declaring member variables:

- Omit the Dim keyword from the variable declaration of member variables.
- A separate declaration is required for each member variable. You can't declare two or more member variables in a single declaration using a comma-separated list.
- You can use the Public or Private keywords for variable declarations. A member variable is private by default; it can be accessed only within the class.
- Member variables cannot be declared Static.
- A class can include an instance of itself as a member, but the variable declaration cannot include the New keyword. That is, the variable declaration cannot create an object of the class.
- Do not use the following LotusScript keywords as member variable names: Public, Private, Static, Sub, Function, Property, Get, Set, New, Delete, and Rem.

Rules for declaring member procedures:

- You can use the keywords Public or Private for procedure declarations. A member procedure is Public by default; it can be accessed outside of the class.
- Member procedures cannot be declared Static.
- All LotusScript keywords are legal as member procedure names. Use the names New and Delete only to name the class constructor and destructor subs, respectively.

Rules for referring to class members:

- Refer to class members using the notation *objName.memberName,* where *memberName* identifies a class member defined in the class of the object reference variable *objName*.
- You can use the keyword **Me** to refer to the object itself when you are inside a member procedure. In the example, Me.textColor refers to the value currently assigned to the textColor member of this instance of the class.
- If you name a class member with a LotusScript keyword, you must refer to the member within member subprograms using the Me keyword.
- Derived class methods can override methods of the base class. The signature of the overriding member must match the signature of the overridden member. Within the procedure of a derived class, you refer to a base class member of the same name using the notation *baseClassName..memberName*.
- Use the With statement to work with members of a specific class using the notation *.memberName*.

Rules for working with objects (class instances):

- To create an object, use the New keyword in a Dim or Set statement for an object reference variable.
- LotusScript sets the initial value of an object reference variable to NOTHING. Use the Is operator to test an object reference variable for the NOTHING value.
- Any Variant variable can take an object reference as its value. Use the IsObject function to test whether the contents of a Variant variable are an object reference.
- Use the Delete statement to delete an object. LotusScript sets the value of variables that refer to the object to NOTHING.

A class definition can include a definition for the constructor sub, named New. If the definition exists, LotusScript calls this sub each time it creates an object of that class.

A class definition can include a definition for the destructor sub, named Delete. If the definition exists, LotusScript calls this sub whenever it deletes an object of that class.

Examples: Class statement

```
' Define a class.
Class textObject
   ' Declare member variables.
   backGroundColor As Integer
   textColor As Integer
   contentString As String

   ' Define constructor sub.
   Sub New (bColor As Integer, tColor As Integer, _
      cString As String)
      backGroundColor% = bColor%
      textColor% = tColor%
      contentString$ = cString$
   End Sub

   ' Define destructor sub.
   Sub Delete
      Print "Deleting text object."
   End Sub

   ' Define a sub to invert background and text colors.
   Sub InvertColors
      Dim x As Integer, y As Integer
      x% = backGroundColor%
      y% = textColor%
      Me.backGroundColor% = y%
      Me.textColor% = x%
   End Sub
End Class

' Create a new object of class textObject.
Dim y As textObject
Set y = New textObject(0, 255, "This is my text")
' Invert the object's background and text colors.
y.InvertColors
' Delete the object.
Delete y
' Output:
' Deleting text object.
```

CLng function

Returns a value converted to the Long data type.

Syntax

CLng (*expr* **)**

Elements

expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value

CLng returns the value of *expr* rounded to the nearest integer, as a Long value.

CLng(EMPTY) returns 0.

If *expr* is a string expression, CLng returns the numeric representation of the string, rounded to the nearest integer. If LotusScript cannot convert the string to a number, the function raises an error.

If the value of *expr* is too large to fit in the Long data type, the function raises an error.

Language cross-reference

@Integer function in formula language

@TextToNumber function in formula language

Examples: CLng function

```
' Convert Double and String values to Long, rounding up or
down as indicated.
Dim x As Double, y as String

x# = 13.400556

Print CLng(x#)   'Prints 13

x# = 13.67

Print CLng(x#)   'Prints 14

y="1.345"

Print CLng(y)         'Prints 1

y="1.678"

Print CLng(y)         'Prints 2

y="string"

Print CLng(y)         'returns a type mismatch error
```

Close statement

Closes one or more open files, after writing all internally buffered data to the files.

Syntax

Close [[**#**] *fileNumber* [**,** [**#**] *fileNumber*] ...]

Elements

fileNumber

Optional. The number that LotusScript assigned to the file when it was opened.

If you omit *fileNumber,* Close closes all open files.

Usage

The pound sign (#) preceding *fileNumber* is optional and has no effect on the statement.

Before closing the open files, Close writes all internally buffered data to the files.

If LotusScript encounters a run-time error that is not handled by an On Error statement, LotusScript closes all open files; otherwise, the files remain open.

If the value of *fileNumber* is contains a fraction, LotusScript rounds the value before using it.

Examples: Close statement

```
Open "c:\rab.asc" For Input Access Read Shared As 1 Len = 128
Close #1
```

CodeLock function

Acquires the lock specified by ID.

Syntax

CodeLock (*lockID***)**

Elements

lockID

ID of lock to be locked (assigned by LotusScript through CreateLock)

Return values

CodeLock will return TRUE, if the lock is acquired.

Usage

Acquires the lock specified by ID. If the lock is already held by another agent, the thread stalls until the lock becomes available. Bear in mind that CodeLock and CodeUnLock should always be done in pairs; failure to adhere to this practice will lead to unexpected results.

Extended examples: lock functions

Extended examples: lock functions

This set of examples demonstrates using the Lock functions to assist in accumulating web site "hits" (counting the number of visits to a web site). The first example demonstrates what happens if several people hit the same web site simultaneously: the users will read the exact same number and the increment will be off.

Example 1:

```
Sub Initialize
      Dim Sess As New NotesSession
      Dim Doc As NotesDocument
      Dim Count As NotesItem

      Set Doc = Sess.SavedData

      Set count = Doc.GetFirstItem("WebHits")
      If count Is Nothing Then
            Set count = New NotesItem(Doc, "WebHits", 0)
      End If

      count.Values = count.Values(0) + 1

      Call Doc.Save(True,False)
End Sub
```

The second example demonstrates how CodeLock can avoid the problem presented in Example 1. You create and make sure you have a secure lock before you read and make changes to the count, and when you are done, you release the lock.

Example 2:

```
Sub Initialize
      Dim Sess As New NotesSession
      Dim Doc As NotesDocument
      Dim Count As NotesItem
      Dim Status As Integer
      Dim LockID As Integer
      Dim others As Integer

      '  Creating a Lock ID or getting the Lock ID
      '  For the event of "WebSiteHits"
      LockID = Createlock("WebSiteHits")

      '  Infinite loop that can only be exited
      '  when this agent has a successfull
      '  lock.  An unsuccessfull lock means
      '  that this agent is presently being
      '  run by someone else.
      Do While True
            If Codelock(LockID) Then
                  Exit Do       '  We finally have a lock,
exiting Loop
            End If
      Loop

      Set Doc = Sess.SavedData

      Set count = Doc.GetFirstItem("WebHits")
      If count Is Nothing Then
            Set count = New NotesItem(Doc, "WebHits", 0)
      End If
```

```
        count.Values = count.Values(0) + 1

        Call Doc.Save(True,False)

        '  Once completed, release and
        '  destroy this lock so another
        '  run of this agent can continue.
        Status = CodeUnlock(LockID)
        Status = DestroyLock(LockID)

End Sub
```

CodeLockCheck function

Returns the number of agents waiting for the the specified lock, plus 1.

Syntax

CodeLockCheck (*lockID***)**

Elements

lockID

ID of lock to be checked (assigned by LotusScript through CreateLock)

Return values

A Long value indicating the sum of the agents that have the lock and are waiting for the lock.

Usage

A sample return value of 4 would mean that one agent has the specified lock and three other agents are waiting for it. Zero indicates the lock is not locked.

CodeUnlock function

Releases the lock, making it available for the next agent requesting it.

Syntax

CodeUnlock (*lockID* **)**

Elements

lockID

ID of lock to be unlocked (assigned by LotusScript through CreateLock)

Return values

CodeUnLock returns TRUE if the lock was successfully released.

Usage

Releases the lock specified by ID. Bear in mind that CodeLock and CodeUnLock should always be done in pairs; failure to adhere to this practice will lead to unexpected results.

Extended examples: lock functions

Command function

Returns the command-line arguments used to start the Lotus software application that started LotusScript.

Syntax

Command[$]

Return value

The return value does not include the program name.

Command returns a Variant of DataType 8 (String). Command$ returns a String.

If the command that started the product specified no arguments, the function returns the empty string (“”).

Usage

You can call the Command function as either Command or Command(). You can call the Command$ function as either Command$ or Command$().

To run a Lotus software application macro in a script, use Evaluate. To start a program from a script, use Shell.

In Lotus Notes, the Command function always returns an empty string.

In OS/2, macros in some products must be converted before they are OS/2 ready.

Examples: Command function

```
If Command$() = "" Then
   Print "No command-line arguments"
Else
   Print "Command-line arguments are: " + Command$()
End If
```

Const statement

Defines a constant.

Syntax

[Public | Private] Const constName = expr [, constName = expr]...

Elements

Public | **Private**

Optional. Public specifies that the constant is visible outside the module where the constant is defined, as long as that module is loaded. Private specifies that the constant is visible only within the module where the constant is defined.

A constant is Private by default.

If you declare a constant within a procedure, you cannot use Public or Private.

constName

The name of the constant.

expr

An expression. The value of the expression is the value of the constant.

The expression can contain any of the following.

- Literal values (numbers and strings)
- Other constants
- Arithmetic and logical operators
- Built-in functions, if their arguments are constant and if LotusScript can evaluate them at compile time. The following functions are evaluated at compile time if their arguments are expressions including only literals and constants.

Functions that can be evaluated as LotusScript constants

Abs	InStrB	RightB
ACos	Int	Round
ASin	LCase	RTrim
ATn	Left	Sgn
ATn2	LeftB	Sin
Bin	Len	Sqr
Cos	LenB	Str
DataType	Log	Tan
DateNumber	LTrim	TimeNumber
Exp	Mid	Trim
Fix	MidB	TypeName
Fraction	Oct	UCase
Hex	Right	Val
InStr		

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

A constant is a named variable whose value cannot be changed. You can declare a constant in a module or a procedure, but you cannot declare a constant in a type or class definition.

You can specify the data type of a constant by appending a data type suffix character to *constName*. Alternatively, if the constant is numeric and *expr* is a numeric literal, you can specify the data type by appending a data type suffix character to *expr*.

If you do not append a data type suffix character to *constName* or *expr*, LotusScript determines the data type of the constant by the value assigned to it.

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

These rules are illustrated in the examples following.

Whether you specify a suffix character in the Const statement or LotusScript determines the data type based on the constant's value, you can use the constant in a script with or without a data type suffix character. If you use the constant with a suffix character, the suffix character must match the data type of the constant.

The data type of a constant is not affected by Def*type* statements.

Examples: Const statement

Example 1

```
Const x = 123.45    ' Define a Double constant.
Const y = 123       ' Define an Integer constant.
Const z = 123456    ' Define a Long constant. The value is too
                    ' large to define an Integer constant.
```

Example 2

```
' Define a String constant, firstName.
Const firstName$ = "Andrea"
' Define a Single constant, appInterest.
Const appInterest! = 0.125
' Define a Currency constant, appLoan.
Const appLoan@ = 4350.20

' Display a message about the amount of interest owed.
MessageBox firstName$ & " owes "  _
   & Format(appLoan@ * appInterest!, "Currency")
```

Cos function

Returns the cosine of an angle.

Syntax

Cos (*angle* **)**

Elements

angle

A numeric expression, specifying an angle expressed in radians.

Return value

Cos returns the cosine of *angle,* a value between -1 and 1, inclusive.

The data type of the return value is Double.

Language cross-reference

@Cos function in formula language

Examples: Cos function

```
Dim degrees As Integer
Dim rad As Double
' Convert the angle 45 degrees to radians.
degrees% = 45
rad# = degrees% * (PI / 180)
' Print the cosine of that angle.
Print Cos(rad#)               ' Prints .707106781186548
```

CreateLock function

Finds the lock ID associated with Name. If none exists, the Lock ID is created.

Syntax

CreateLock(*lockName***)**

Elements

lockName

String identifier for this particular lock.

Return values

CreateLock will return the lock ID as a simple integer. It will return an error if the platform does not support locks or if there is insufficient shared memory.

Usage

Note that the variable the lock ID is stored in is simply an integer. If the variable goes out of scope the ID will be lost. It can be recovered by calling CreateLock again with the same name. Locks are unique across the current shared memory name space. Locks are freed automatically when the thread exits or may be freed by DestroyLock.

Note When a lock ID is lost DestroyLock cannot be used on the lock and system resources are taken up by the lock until the ID is recovered and the lock destroyed or the agent or thread is exited.

Extended examples: lock functions

CreateObject function

Creates an OLE Automation object of the specified class.

Note CreateObject is not supported under OS/2 or UNIX. It is supported on the Macintosh as long as OLE support is installed.

Syntax

CreateObject (*className* **)**

Elements

className

A string of the form *appName.appClass*, designating the kind of object to create (for example, "WordPro.Application").

The *appName* is an application that supports OLE Automation.

The *appClass* is the class of the object to create. Products that support OLE Automation provide one or more classes. See the product documentation for details.

Return value

CreateObject returns a reference to an OLE Automation object.

Usage

Use the Set statement to assign the object reference returned by CreateObject to a Variant variable.

If the application is not already running, CreateObject starts it before creating the OLE Automation object. References to the object remain valid only while the application is running. If the application terminates while you are using the object reference, LotusScript raises a run-time error.

LotusScript supports the OLE vartypes listed in the table below. Only an OLE method or property can return a vartype designated as "OLE only."

OLE vartype	*Description*
VT_EMPTY	(No data)
VT_NULL	(No data)
VT_I2	2-byte signed integer
VT_I4	4-byte signed integer
VT_R4	4-byte real
VT_R8	8-byte real

continued

OLE vartype	*Description*
VT_CY	Currency
VT_DATE	Date
VT_BSTR	String
VT_DISPATCH	IDispatch, OLE only
VT_ERROR	Error, OLE only
VT_BOOL	Boolean
VT_DECIMAL	Decimal, converted to Double
VT_VARIANT	(A reference to data of any other type)
VT_UNKNOWN	IUnknown, OLE only
VT_ARRAY	(An array of data of any other type)

LotusScript supports iterating over OLE collections with a ForAll statement.

LotusScript supports passing arguments to OLE properties. For example:

```
' Set v.prop to 4; v.prop takes two arguments.
v.prop(arg1, arg2) = 4
```

LotusScript does not support identifying arguments for OLE methods or properties by name rather than by the order in which they appear, nor does LotusScript support using an OLE name by itself (without an explicit property) to identify a default property.

Results are unspecified for arguments to OLE methods and properties of type boolean, byte, and date that are passed by reference. LotusScript does not support these data types.

The word CreateObject is not a LotusScript keyword.

Examples: CreateObject function

This example creates a Notes session and displays some information from it.

```
' Create a Notes session and display the current user's name.
Dim session As Variant
Set session = CreateObject("Notes.NotesSession")
Messagebox session.UserName
```

The following script works on the Mac with Microsoft Word installed

```
Sub Initialize
      Set MyApp = CreateObject ( "Word.Application")
      MyApp.Visible = True
End Sub
```

CSng function

Returns a value converted to the Single data type.

Syntax

CSng (*expr* **)**

Elements

expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value

CSng returns the numeric value of *expr* as a Single value.

CSng(EMPTY) returns 0.

If *expr* is a string expression, CSng returns the numeric representation of the string, including any fractional part. If LotusScript cannot convert the string to a number, the function raises an error.

If the value of *expr* is too large to fit in the Single data type, the function raises an error.

Language cross-reference

@TextToNumber function in formula language

Examples: CSng function

```
' Convert a Double value by rounding to nearest Single.
Dim x As Double
x# = 1.70800003057064E+24
Print CSng(x#)              ' Prints 1.708E+24
```

CStr function

Returns a value converted to the String data type.

Syntax

CStr (*expr* **)**

expr

Any numeric expression, date, or string expression that LotusScript can convert to a string.

Return value
CStr returns the value of *expr* as a String value.

CStr(EMPTY) returns the empty string (“”).

Language cross-reference
@Text function in formula language

Examples: CStr function
```
Dim x As Integer
Dim y As Integer
x% = 1
y% = 2

' Use the addition operator +
Print x% + y%                      ' Prints 3

' Use the string concatenation operator +
Print CStr(x%) + CStr(y%)          ' Prints 12
```

CurDir function

Returns the current directory on a specified drive.

Syntax
CurDir[**$**] [**(** *drive* **)**]

Elements
drive

Optional. A string expression specifying an existing drive. If you omit *drive,* CurDir uses the current drive.

Return value
CurDir returns the current directory on *drive.*

CurDir returns a Variant of DataType 8 (String). CurDir$ returns a String.

Usage
If the value of *drive* is a string of more than one character, CurDir uses only the first character. CurDir does not require a colon after the drive letter.

To set the current directory on a specified drive, use ChDir. To set the current drive, use ChDrive. To return the current drive, use CurDrive.

You can call this function with no arguments as either CurDir or CurDir( ).

Language cross-reference
@FileDir function in formula language

Examples: CurDir function

```
ChDir "c:\test"
Print CurDir$()   ' Prints "c:\test"
```

CurDrive function

Returns a string identifying the current drive.

Syntax

CurDrive[**$**]

Return value

CurDrive returns the current drive letter followed by a colon.

CurDrive returns a Variant of DataType 8 (String). CurDrive$ return a String.

To set the current directory on a specified drive, use ChDir. To set the current drive, use ChDrive. To return the current directory on a drive, use CurDir.

You can call the CurDrive function as either CurDrive or CurDrive(). You can call the CurDrive$ function as either CurDrive$ or CurDrive$().

Examples: CurDrive function

```
Dim tempDrive As String
tempDrive$ = CurDrive$()
If tempDrive$ <> "c:" Then
   ChDrive "c"
End If
ChDir "\test"
Print CurDir$()   ' Prints "c:\test"
```

Currency data type

Specifies a variable that contains an 8-byte integer, scaled to four decimal places to suitably represent a monetary value.

Usage

The Currency suffix character for implicit type declaration is @.

Use the Currency data type for calculations with money.

Currency variables are initialized to 0.

The range of Currency values is -922,337,203,685,477.5807 to 922,337,203,685,477.5807, inclusive.

On UNIX platforms, the values must fall within the range -922,337,203,685,477.5666 to 922,337,203,685,477.5666, inclusive.

Use the Currency data type for fixed point calculations in which four-decimal-place accuracy is meaningful.

LotusScript aligns Currency data on a 4-byte boundary. In user-defined types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Currency data type

```
' Explicitly declare two Currency variables.
Dim sales As Currency
Dim expenses As Currency
sales@ = 20.9999
expenses@ = 10.5555
' Implicitly declare a Currency variable.
earnings@ = sales@ - expenses@
' Currency is calculated to four decimal places.
Print earnings@              ' Prints 10.4444
```

CVar function

Returns a value converted to the Variant data type.

Syntax

CVar (*expr* **)**

Elements

expr

Any expression.

Return value

CVar returns the value of *expr.*

The data type of the return value is Variant.

Examples: CVar function

```
' The Abs function requires a numeric or Variant argument.
' Convert a string value to Variant and use it in Abs.

Dim gNum As String
gNum$ = "-1"
Print Abs(CVar(gNum$))   ' Prints 1 (absolute value of -1)
Print Abs (gNum$)        ' Generates an error
```

DataType function

Returns the data type of the value of an expression.

Syntax

DataType (*expr* **)**

VarType is acceptable in place of DataType.

Elements

expr

Any expression.

Return value

DataType returns a number representing the data type of *expr.*

The following table describes the possible return values. The first column is the return value. The last column is "Yes" if the return value applies to variants only.

Return	*Value type*	*Constant*	*Variants only*
0	EMPTY	V_EMPTY	Yes
1	NULL	V_NULL	Yes
2	Integer	V_INTEGER	
3	Long	V_LONG	
4	Single	V_SINGLE	
5	Double	V_DOUBLE	
6	Currency	V_CURRENCY	
7	Date/Time	V_DATE	Yes
8	String	V_STRING	
9	OLE object or NOTHING	V_DISPATCH	Yes
10	OLE error	V_ERROR	Yes
11	Boolean	V_BOOLEAN	
12	Variant list or array	V_VARIANT	
13	IUNKNOWN (OLE value)	V_IUNKNOWN	Yes
17	Byte	V_BYTE	
34	User-defined object	V_LSOBJ	
35	Product object	V_PRODOBJ	
2,048	List		
8,192	Fixed array		
8,704	Dynamic array		

Usage

The file lsconst.lss defines the constants described in the preceding table. If you want to refer to the return values as symbolic constants instead of numbers, use the %Include directive to include this file in your script.

If the argument to DataType is a list or an array, the return value is the sum of the value that represents a list or an array plus the value that represents the data type of elements of the list or array. For example, a fixed array of Integers is 8194 (that is, 8192 + 2); a list of Variants is 2060 (that is, 2048 + 12).

The return value 13 signifies an unknown value type, corresponding to the OLE value IUNKNOWN. To test for this value, use the IsUnknown function.

Language cross-reference

@IsNumber function in formula language

@IsTime function in formula language

@IsText function in formula language

Examples: DataType function

```
Dim item(5) As Variant    ' Declare a Variant fixed array.
Dim itemWeight As Single
Dim itemName As String

itemWeight! = 2.7182
itemName$ = "Jute twine"
item(1) = itemWeight!
item(2) = itemName$
Print DataType(item(1))   ' Prints 4
Print DataType(item(2))   ' Prints 8
Print DataType(item(3))   ' Prints 0 (initalized to EMPTY)
Dim cells As Range        ' Suppose Range is a
                          ' product-defined class.
Print DataType(cells)     ' Prints 35
Set cells2 = cells
Print DataType(cells2)    ' Prints 35
Dim areas(3) As Range     ' An array of Range product objects
Print DataType(areas)     ' Prints 8227 (8192 + 35)
Set cal = CreateObject("dispcalc.ccalc")
Print DataType(cal)       ' Prints 9
Dim stats(3) As Integer   ' An array of Integers
Print DataType(stats%)    ' Prints 8194 (8192 + 2)
Dim misc List As Variant  ' A list of Variants
Print DataType(misc)      ' Prints 2060 (2048 + 12)
```

About data types

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

Data type	*Suffix*	*Value range*	*Size*
Byte	none	0 to 255 Initial value: 0	1 byte
Boolean	none	0 (False) or -1 (True) initial value: 0	2 bytes
Integer	%	-32,768 to 32,767 Initial value: 0	2 bytes
Long	&	-2,147,483,648 to 2,147,483,647 Initial value: 0	4 bytes
Single	!	-3.402823E+38 to 3.402823E+38 Initial value: 0	4 bytes
Double	#	-1.7976931348623158+308 to 1.7976931348623158+308 Initial value: 0	8 bytes
Currency	@	-922,337,203,685,477.5807 to 922,337,203,685,477.5807 Initial value: 0	8 bytes
String	$	(String length ranges from 0 to 32K characters) Initial value: "" (empty string)	(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	An aggregate set of elements having the same data type. An array can comprise up to 8 dimensions whose subscript bounds can range from -32768 to 32767. Initial value: Each element in a fixed array has an initial value appropriate to its data type.	Up to 64K bytes
List	A one-dimensional aggregate 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 any scalar value, array, list, or object reference. Initial value: EMPTY	16 bytes

continued

Data type or structure	*Description*	*Size*
User-defined data type	An aggregate set of elements of possibly disparate data types. Comparable to a record in Pascal or a struct in C. Initial value: Member variables have initial values appropriate to their data types.	Up to 64K bytes
Class	An aggregate set of elements of possibly disparate data types together with procedures that operate on them. Initial value: When you create an instance of a class, LotusScript initializes its member variables to values appropriate to their data types, and generates an object reference to it.	
Object reference	A pointer to an OLE Automation object or an instance of a product class or user-defined class. Initial value: NOTHING.	4 bytes

In each of the preceding tables, the specified storage size is platform independent.

Date function

Returns the current system date as a date/time value.

Syntax
Date[$]

Return value
Date returns the integer part of the value returned by the Now function. Date returns that value as a Variant of DataType 7 (Date/Time). Date$ returns that value as a String.

Usage
The Date function is equivalent to the Today function.

You can call the Date function as either Date or Date( ). You can call the Date$ function as either Date$ or Date$().

Examples: Date function
```
Print Date$         ' Prints "04/25/95" if the current
                    ' system date is April 25, 1995.
```

Date statement

Sets the system date.

Syntax

Date[$] = *dateExpr*

Elements

dateExpr

Any expression whose value is a valid date/time value: either a String in a valid date/time format, or else a Variant containing either a date/time value or a string value in date/time format.

If *dateExpr* is a string in which the date part contains only numbers and valid date separators, the operating system's international Short Date format determines the order in which the numbers are interpreted as month, day, and year values. The date part of the string must have one of the following forms:

mm-dd-yy or dd-mm-yy
mm-dd-yyyy or dd-mm-yyyy
mm/dd/yy or dd/mm/yy
mm/dd/yyyy or dd/mm/yyyy

Usage

If you specify a 2-digit year designation (yy) in Notes or Domino, LotusScript interprets 50 through 99 as the years 1950 through 1999 and 00 through 49 as the years 2000 through 2049. For example, 88 and 1988 are equivalent year designations and 12 and 2012 are equivalent year designations.

If you specify a 2-digit year designation in SmartSuite, LotusScript interprets the years differently. For information on how each SmartSuite product interprets 2-digit year designations, see the online Help entry entitled Year 2000. This entry appears on the Help menu of each SmartSuite product.

Examples: Date statement

```
' Depending on the international Short Date format,
' set the system date to September 7, 2003 or to 9 July, 2003.
Date$ = "09-07-03"
```

DateNumber function

Returns a date value for a given set of year, month, and day numbers.

Syntax

DateNumber (*year* **,** *month* **,** *day* **)**

DateSerial is acceptable in place of DateNumber.

Elements

year

A numeric expression designating a year.

If you specify a 2-digit year designation (yy) in Notes or Domino, LotusScript interprets 50 through 99 as the years 1950 through 1999 and 00 through 49 as the years 2000 through 2049. For example, 88 and 1988 are equivalent year designations and 12 and 2012 are equivalent year designations.

If you specify a 2-digit year designation in SmartSuite, LotusScript interprets the years differently. For information on how each SmartSuite product interprets 2-digit year designations, see the online Help entry entitled Year 2000. This entry appears on the Help menu of each SmartSuite product.

month

A numeric expression designating a month of the year (a value from 1 through 12).

If you assign *month* a negative value, DateNumber calculates a prior date by measuring backward from December of the preceding year. (For example, 1995, -2, 10 is evaluated as October 10, 1994.)

day

A numeric expression designating a day of the month (a value from 1 through 31).

If you assign *day* a negative value, then DateNumber calculates a prior date by measuring backward from the last day of the month immediately preceding the specified month. (For example, 1995, 5, -3 is evaluated as April 27, 1995, by subtracting 3 from 30, the last day of April, the month before the 5th month.)

Return value

DateNumber returns the date value for the given *year, month,* and *day.*

The data type of the return value is a Variant of DateType 7 (Date/Time).

Language cross-reference

@Date function in formula language

@Time function in formula language

Examples: DateNumber function

```
Print DateNumber(1999, 10, 8)          ' Prints 10/8/99

' The following two functions calculate a past date
' using negative arguments.
' Print the date 5 months and 10 days before 2/4/99.
Print DateNumber(99, 2 - 5, 4 - 10)    ' Prints 8/25/98
' Print the date 3 months and 6 days before 1/1/99.
Print DateNumber(99, -3, -6)           ' Prints 8/25/98
```

DateValue function

Returns the date value represented by a string expression.

Syntax

DateValue (*stringExpr* **)**

Elements

stringExpr

A string expression representing a date/time. *stringExpr* must be a String in a valid date/time format or else a Variant containing either a date/time value or a string value in date/time format. If you omit the year in *stringExpr,* DateValue uses the year in the current system date.

If *stringExpr* is a string whose date part contains only numbers and valid date separators, the operating system's international Short Date format determines the order in which the numbers are interpreted as month, day, and year values.

If you specify a 2-digit year designation (yy) in Notes or Domino, LotusScript interprets 50 through 99 as the years 1950 through 1999 and 00 through 49 as the years 2000 through 2049. For example, 88 and 1988 are equivalent year designations and 12 and 2012 are equivalent year designations.

If you specify a 2-digit year designation in SmartSuite, LotusScript interprets the years differently. For information on how each SmartSuite product interprets 2-digit year designations, see the online help entry entitled Year 2000. This entry appears on the Help menu of each SmartSuite product.

Return value

DateValue returns the date value represented by *stringExpr*.

The data type of the return value is a Variant of DataType 7 (Date/Time).

Usage

If the *stringExpr* argument specifies a time of day, DateValue validates the time, but omits it from the return value.

Language cross-reference

@TextToTime function in formula language

Examples: DateValue function

```
Dim birthDateV As Variant
' Calculate the date value for October 8, 1996.
birthDateV = DateValue("October 8, 1996")
' Print this value as a date string.
Print CDat(birthDateV)   ' Prints 10/8/96
' Print the age this person reaches, in years,
' on this year's birthday.
Print Year(Today) - Year(birthDateV)
```

Day function

Returns the day of the month (an integer from 1 to 31) for a date/time argument.

Syntax

Day (*dateExpr* **)**

Elements

dateExpr

Any of the following kinds of expression:

- A valid date/time string of String or Variant data type.

  In a 2-digit year designation (yy) in Notes or Domino, LotusScript interprets 50 through 99 as the years 1950 through 1999 and 00 through 49 as the years 2000 through 2049. For example, 88 and 1988 are equivalent year designations and 12 and 2012 are equivalent year designations.

  In a 2-digit year designation in SmartSuite, LotusScript interprets the years differently. For information on how each SmartSuite product interprets 2-digit year designations, see the online Help entry entitled Year 2000. This entry appears on the Help menu of each SmartSuite product.

- A numeric expression whose value is a Variant of DataType 7 (Date/Time)
- A number within the valid date range: the range -657434 (representing Jan 1, 100 AD) to 2958465 (Dec 31, 9999 AD)
- NULL

Return value

Day returns an integer between 1 and 31.

The data type of the return value is a Variant of DataType 2 (Integer).

Day(NULL) returns NULL.

Language cross-reference

@Day function in formula language

Examples: Day function

```
Dim x As Variant, dd As Integer
x = DateNumber(1992, 4, 7)
dd% = Day(x)
Print dd%           ' Prints 7
```

Declare statement (external C calls)

Declares a LotusScript function or sub that calls an external C function, allowing calls to a function that is defined in a shared library of C routines.

Note the Declare statement (external C calls) is not supported under OS/2.

Syntax

Declare [**Public** | **Private**] { **Function** | **Sub** } *LSname* **Lib** *libName* [**Alias** *aliasName*]
([*argList*]) [**As** *returnType*]

Elements

Public | **Private**

Optional. Public indicates that the declared C function is visible outside this module, for as long as the module is loaded. Private indicates that the declared C function is visible only within this module.

A declared C function is Private by default.

Function | **Sub**

Specifies that the C function is to be called as either a function or a sub.

LSname

The function or sub name used within LotusScript. If you omit the Alias clause, this name must match the name declared in the shared library.

If the statement is declaring a Function (using the keyword Function), then you can append a data type suffix character to *LSname,* to declare the type of the function's return value.

libName

A literal string, or a string constant, specifying the shared library file name. The file name extension is optional. You can optionally include a complete path specification. LotusScript automatically converts *libName* to uppercase. If you need to preserve case sensitivity, use the *aliasName* described below.

aliasName

Optional. A literal string containing one of the following:

- A case-sensitive C function name as declared in the shared library
- A pound sign (#) followed by an ordinal number representing the position of the function in the library; for example, "#1"

This argument is useful when the C function name is not a valid LotusScript name, or when you need to preserve case sensitivity (for example, when calling an exported library function in a 32-bit version of Windows).

argList

Optional. An argument list for the external function. Parentheses enclosing the list are required, even if the C function takes no arguments.

argList has the form:

argument [**,** *argument*] ...

where *argument* is:

[**ByVal**] *name* **As** [**LMBCS** | **Unicode**] [*dataType* | **Any**]

The optional LMBCS and Unicode keywords may be used with the String data type only, to specify the character set. See the usage information and examples that follow.

Use the keyword Any to pass an argument to a C function without specifying a data type, suppressing type checking.

returnType

The data type of the function's return value. The clause **As** *returnType* is not allowed for a sub, since a sub doesn't return a value.

For a function, either specify **As** *returnType*, or append a data type suffix character to *LSname,* to declare the data type of the function's return value. Do not specify both a *returnType* and a data type suffix character.

You can't use Any as a *returnType.*

You can't use Variant, Currency, or fixed-length String as a *returnType*.

If you omit **As** *returnType* and the function name has no data type suffix character appended, the function returns a value of the data type specified by a Def*type* statement that applies to the function name. A C function can't return a Variant; so a DefVar statement can't apply to the function name.

returnType has the form:

[LMBCS | **Unicode]** *dataType*

The *dataType* must match the C function return type exactly; no conversion is performed on the return value.

The optional LMBCS and Unicode keywords may be used with the String data type only, to specify the character set. See the usage information and examples that follow.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

You can only declare external functions at the module level. If a function is not typed with a return type or a data type suffix character, LotusScript generates an error.

The "_" is reserved for Notes specific DLLs. This is a change put in as of Notes 4.5.1. If you attempt to load a DLL in Notes 4.51 or greater using LotusScript and the name of the DLL is preceded by an underscore you will receive the error "Error in loading DLL".

Passing arguments

By default, LotusScript passes arguments to external functions by reference. Arguments can be passed by value using the ByVal keyword, but only if LotusScript can convert the value passed to the data type of the corresponding C function argument.

Arrays, type variables, and user-defined objects must be passed by reference.

You can't pass lists as arguments to C functions.

You can't use a fixed-length String as an argument.

Product objects can be passed by reference (passing a reference to the instance handle) or by value (passing the instance handle itself). They can be passed by value only by using the keyword ByVal. Parentheses can't be used on the actual argument.

An argument can be typed as Any to avoid data type restrictions. Arguments of type Any are always passed by reference, regardless of the type of data they contain. You can pass a Variant containing an array or list to a C function argument declared as Any.

Using LMBCS or Unicode strings

Use the optional keywords LMBCS and Unicode with a String *argument* or *returnType* to specify the character set.

Unicode designates a Unicode string of two-byte characters (words) using the platform-native byte order.

LMBCS designates a LMBCS optimization group 1 string (multibyte characters).

If neither LMBCS nor Unicode is specified, the string variable uses the platform-native character set.

Calling exported library functions in 32-bit versions of Windows

If you're using a 32-bit version of Windows, the names of exported library functions are case sensitive; however, LotusScript automatically converts them to uppercase in the Declare statement. To successfully call an exported library function, use the Alias clause to specify the function name with correct capitalization (LotusScript leaves the alias alone).

Examples: Declare statement (external C calls)

Example 1

```
Dim strOut As String
' Declare the external function StrUpr, defined in StrLib.
Declare Function StrUpr Lib "StrLib" (ByVal inVal As String) _
   As String
' Call StrUpr
strOut$ = StrUpr("abc")
```

Example 2

```
' Declare an exported library function (SendDLL) with an alias
' to preserve case sensitivity.
Declare Function SendDLL Lib "C:\myxports.dll" _
   Alias "_SendExportedRoutine" (i1 As Long, i2 As Long)
' Call SendDLL
SendDLL(5, 10)
```

Example 3

```
' Pass  the string argument amIStr to the function StrFun as
' a Unicode string. The function's return value is also
' a Unicode string.
Declare Function StrFun Lib "lib.dll" _
   (amIStr As Unicode String) As Unicode String
```

Example 4

```
' Pass the string argument amLStr to the function StrFun as
' a LMBCS string. The function's return value is a LotusScript
' platform-native string.
Declare Function StrFun Lib "lib.dll" _
   (amLStr As LMBCS String) As String
```

Declare statement (forward reference)

Declares a forward reference to a procedure (a function, sub, or property), allowing calls to a procedure that has not yet been defined.

Syntax

Declare [**Static**] [**Public** | **Private**] *procType procName* [**(** [*argList*] **)**] [**As** *returnType*]

Elements

Static

Optional. Specifies that the values of the procedure's local variables are saved between calls to the procedure.

If this keyword is present, it must also be present in the definition of the procedure.

Public | **Private**

Optional. Public indicates that the declared procedure is visible outside this module, for as long as the module is loaded. If this keyword is present, it must also be present in the definition of the procedure.

Private indicates that the declared procedure is visible only within this module. If this keyword is present, it must also be present in the definition of the procedure.

procType

One of the following four keyword phrases, to identify the kind of procedure:

Function
Sub
Property Get
Property Set

procName

The name of a function, sub, or property. If *procType* is Function (a function is being declared), then *procName* can have a data type suffix character appended to declare the type of the function's return value.

argList

A comma-separated list of argument declarations for the procedure. The procedure must be a function or a sub (*procType* must be Function or Sub). The argument declarations must match the argument declarations in the function or sub definition exactly.

The syntax for each argument declaration is:

[**ByVal**] *argument* [**( )** | **List**] [**As** *type*]

ByVal means that *argument* is passed by value: that is, the value assigned to *argument* is a local copy of a value in memory, rather than a pointer to that value.

*argument***()** is an array variable. *argument* **List** identifies *argument* as a list variable. Otherwise, *argument* can be a variable of any of the other data types that LotusScript supports.

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 *argument* doesn't end in a data type suffix character (and isn't covered by an existing Def*type* statement), its data type is Variant.

Enclose the entire list of argument declarations in parentheses.

returnType

The data type of the function's return value. This is optional for a function, and not allowed for a sub or a property, because they don't return values. *returnType* must match the return type specified in the function definition; no conversion is performed on the return value.

If you omit **As** *returnType,* the function name's data type suffix character appended to *procName* (the function name) determines the return value's type. Do not specify both a *returnType* and a data type suffix character.

If you omit **As** *returnType* and *procName* has no data type suffix character appended, the function returns a value either of data type Variant or of the data type specified by a Def*type* statement.

Usage

The IDE implicitly generates forward declarations of procedures; directly entering them in the IDE is unnecessary and causes syntax errors. You can %Include a file containing forward declarations of procedures contained in the file. You can directly enter in the IDE forward declarations of class properties and methods.

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

You can make a forward declaration only at the module level or within a class.

The procedure, if it exists, must be defined in the same scope as the forward declaration. LotusScript does not generate an error if a procedure has a forward declaration but is not defined. (An error will be generated if you try to call a procedure that has been declared but not defined.)

A procedure declared within a class definition cannot be declared as Static.

The use of Static, Public, and Private keywords in a Property Get forward declaration must match their use in the corresponding Property Set forward declaration, if one exists.

Examples: Declare statement (forward reference)

```
' The forward declaration of the function Times allows the
' use of Times within the definition of the sub PrintFit.
' The function definition of Times appears later in the
script.

' Forward declare the function Times.
Declare Function Times (a As Single, b As Single) As Single

' Define the sub PrintFit. It calls Times.
Sub PrintFit (lead As String, x As Single)
   Print lead$, Times (x!, x!)
End Sub

' Define Times.
Function Times (a As Single, b As Single) As Single
   Times = (a! - 1.0) * (b! + 1.0)
End Function

' Call the sub PrintFit.
PrintFit "First approximation is:", 13
' Prints "First approximation is:     168"
```

Def*type* statements

Set the default data type for variables, functions, and properties whose names begin with one of a specified group of letters.

Syntax

DefBool *range* [, *range*] ...

DefByte *range* [, *range*] ...

DefCur *range* [, *range*] ...

DefDbl *range* [, *range*] ...

DefInt *range* [, *range*] ...

DefLng *range* [, *range*] ...

DefSng *range* [, *range*] ...

DefStr *range* [, *range*] ...

DefVar *range* [, *range*] ...

Elements

range

A single letter, or two letters separated by a hyphen. Spaces or tabs around the hyphen are ignored. A two-letter range specifies the group of letters including the given letters and any letters between. These must be letters with ASCII codes less than 128.

Letters in *range* are case insensitive. For example, the group of letters J, j, K, k, L, and l can be designated by any one of these *range* specifications: J-L, L-J, j-L, L-j, J-l, l-J, j-l, or l-j.

Usage

The following table lists the Def*type* statements, the data type that each one refers to, and the data type suffix character for that data type.

Statement	*Data type*	*Suffix character*
DefBool	Boolean	(none)
DefByte	Byte	(none)
DefCur	Currency	@
DefDbl	Double	#
DefInt	Integer	%
DefLng	Long	&
DefSng	Single	!
DefStr	String	$
DefVar	Variant	(none)

Def*type* statements can only appear at the module level, but they affect all declarations contained within the module at module level and within its procedures. They do not affect the declarations of data members of types and classes. They do affect declarations of function members and property members of classes.

All Def*type* statements in a module must appear before any declaration, explicit or implicit, in the module. Exception: the declaration of a constant (by the Const statement) is not affected by Def*type* statements.

No range in any Def*type* statement can overlap any other range in the same Def*type* statement or in any other Def*type* statement in the same module.

The range A-Z is special. It includes all international characters, not only the letters with ASCII codes less than 128. It is the only range specification that includes international characters. For example, to change the default data type of all variables, functions, and properties to Single (the standard data type for several versions of BASIC), specify DefSng A-Z.

Declarations that are explicit as to data type (such as Dim X As Integer, Dim Y$, or Define MyFunction As Double) take precedence over Def*type* declarations.

Examples: Deftype statements

```
DefInt a-z

' x is declared explicitly, with no type.
Dim x
Print TypeName(x)                ' Output: INTEGER
' Ñ is declared explicitly, with no type.
Dim Ñ
Print TypeName(Ñ)                ' Output: INTEGER
' y is declared explicitly, with the String type.
' The specified type overrules the DefInt statement.
Dim y As String
Print TypeName(y)                ' Output: STRING
' b is declared implicitly, with the String type.
' The suffix character $ overrules the DefInt statement.
b$ = "Rebar"
Print TypeName(b$)               ' Output: STRING
' sNum is declared implicitly, which makes it Integer by
' default because DefInt a-z is in effect.
sNum = 17.6
Print TypeName(sNum), sNum   ' Output: INTEGER  18
                             ' because LotusScript rounds when
                             ' converting to type Integer.
```

Delete statement

Executes an object's Delete sub, if the sub exists, and then deletes the object.

Syntax

Delete *objRef*

Elements

objRef

An object reference variable or Variant containing an object reference.

Usage

The Delete statement calls the Delete sub in the object's class definition (if one exists), and then sets all references to the object to NOTHING.

If the object's class is a derived class, LotusScript executes the base class's Delete sub (if one exists) after executing the class's Delete sub.

For product objects, the interpretation of a Delete statement is up to the product. In some cases, for example, the Delete statement deletes the object reference, but not the object itself. A product may provide its own script mechanism for deleting the object. In Lotus Notes Release 4, for example, you can use the Delete statement to delete an object reference to a Notes database, but you use the NotesDatabase class Remove method to delete the database itself (an .nsf file).

Examples: Delete statement

```
' Define the class Customer.
Class Customer
   Public Name As String
   Public Address As String
   Public Balance As Currency

   ' Define a constructor sub for the class.
   Sub New (Na As String, Addr As String, Bal As Currency)
      Me.Name$ = Na$
      Me.Address$ = Addr$
      Me.Balance@ = Bal@
   End Sub

   ' Define a destructor sub for the class.
   Sub Delete
      Print "Deleting customer record for: "; Me.Name$
   End Sub
End Class

' Create an object of the Customer class.
Dim X As New Customer ("Acme Corporation", _
   "55 Smith Avenue, Cambridge, MA", 14.92)
Print X.Balance@
' Output:
' 14.92

' Delete the object, first running the destructor sub.
Delete X
' Output:
' Deleting customer record for: Acme Corporation


' Then the object is deleted.
```

DestroyLock function

Removes the current link to the lock specified. If the number of links is zero, the lock is destroyed.

Syntax

DestroyLock (*lockID* **As Integer)**

Elements

lockID

ID of lock to be destroyed (assigned by LotusScript through CreateLock)

Return values

DestroyLock returns TRUE if the lock was successfully destroyed.

Usage

Any agent that uses locks should be sure to use the DestroyLock function when they are done using a lock. If the lock is not destroyed, it will continue to use system resources as no one can use that lock again until the agent exits.

Extended examples: lock functions

Dim statement

Declares variables.

Syntax

{ Dim | Static | Public | Private } variableDeclaration [, variableDeclaration]...

Elements

Dim | Static | Public | Private

Variable declarations begin with one of the words Dim, Static, Private, or Public.

Dim indicates that a variable is nonstatic and private by default.

- Static indicates that the variable's value is saved between calls to the procedure where the variable is declared.
- Public indicates that the variable is visible outside the scope (module or class) where the variable is defined, for as long as this module remains loaded.
- Private indicates that the variable is visible only within the current scope.

You can use the Static keyword in procedure scope, but not in module or class scope. You can use the Public and Private keywords in module or class scope, but not in procedure scope.

variableDeclaration

The declaration has one of the following forms, depending on the kind of variable being declared:

- Scalar variable: *variableName*[*dtSuffix*] [**As** *type*]
- Object reference variable: *variableName* **As** [**New**] *type* [*argList*]
- List variable: *variableName*[*dtSuffix*] **List** [**As** *type*]
- Array variable: *variableName*[*dtSuffix*] **(** [*bounds*] **)** [**As** *type*]

You can declare any number of variables in a single statement, separated by commas.

variableName

The name of the variable being declared.

dtSuffix

Optional. A character that specifies the data type of *variableName*. The data type suffix characters and the data types that they represent are: @ for Currency, # for Double, % for Integer, & for Long, ! for Single, and $ for String.

type

Optional for scalar variables, lists, and arrays. A valid LotusScript data type, user-defined data type, user-defined class, or product class. This specifies the type of *variableName*.

If *type* is the name of a class, *variableName* is an object reference for that type: its value can only be a reference to an instance of that class or to an instance of a derived class of that class, or the value NOTHING.

New

Optional. Valid only if *type* is the name of a user-defined or product class. New creates a new object of the class named by *type*, and assigns a reference to that object in *variableName*.

Note that in some cases, Lotus products provide other mechanisms for creating product objects in scripts, such as product functions or product object methods. See your Lotus software documentation for details.

argList

Optional. This is valid only if the New keyword is specified.

For user-defined classes, *argList* is the comma-separated list of arguments required by the class constructor sub New, defined in the class named by *type*. For product classes, consult the product documentation.

bounds

Optional. *bounds* is a comma-separated list of bounds for the dimensions of a fixed array. Each bound is specified in the form

[*lowerBound* **To**] *upperBound*

where *lowerBound* is a number designating the minimum subscript allowed for the dimension, and *upperBound* is a number designating the maximum. If no *lowerBound* is specified, the lower bound for the array dimension defaults to zero (0), unless the default lower bound has been changed to 1 using the Option Base statement. For example, with a default lower bound of 0, the following statement allocates storage for 4 strings instead of the assumed 3 strings:

```
Dim strArray(3) as String
```

If you don't define any *bounds*, the array is defined to be a dynamic array.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

Explicit declarations and implicit declarations

You can declare a variable name either explicitly or implicitly. The Dim statement declares a name explicitly. A name is declared implicitly if it is used (referred to) when it has not been explicitly declared, or when it is not declared as a Public name in another module being used by the module where the name is referred to. You can prohibit implicit declarations by including the statement Option Declare in your script.

Specifying the data type

Either *dtSuffix* or As *type* can be specified in *variableDeclaration*, but not both. If neither is specified, the data type of *variableName* is Variant.

The data type suffix character, if it is specified, is not part of the variable name. When the name is used (referred to) in the script, it can be optionally suffixed by the appropriate data type suffix character.

Declaring arrays

For a fixed array, Dim specifies the type of the array, the number of dimensions of the array, and the subscript bounds for each dimension. Dim allocates storage for the array elements and initializes the array elements to the appropriate value for that data type (see "Initializing variables," later in this section).

For a dynamic array, Dim only specifies the type of the array. The number of dimensions of the array and the subscript bounds for each dimension are not defined; and no storage is allocated for the array elements. The declaration of a dynamic array must be completed by a later ReDim statement.

Arrays can have up to 8 dimensions.

Array subscript bounds must fall in the range -32,768 to 32,767, inclusive.

Declaring lists

A list is initially empty when it is declared: it has no elements, and no storage is allocated for it. An element is added to a list when the list name with a particular list tag first appears on the left-hand side of an assignment statement (a Let statement or a Set statement).

If the character set is single byte, Option Compare determines whether list names are case sensitive. For example, if Option Compare Case is in effect, the names "ListA" and "Lista" are different; if Option Compare NoCase is in effect, these names are the same. If the character set is double byte, list names are always case and pitch sensitive.

Declaring object reference variables

If *type* is the name of a class and the keyword New is not specified, the initial value of the declared object reference variable is NOTHING. To assign another value to an object reference variable, use the Set statement later in the script.

Dim *variableName* As New *className* generates executable code. When you save a compiled module, module-level executable code is not saved, so be careful about using such a statement at the module level. Your Lotus software may prohibit you from placing executable statements at the module level.

You may prefer to declare the object reference variable at the module level with Dim *variableName* As *className*, which is not executable code, then use a Set statement (which is executable code) in a procedure to bind the object reference variable to an object.

The New keyword is not valid in an array declaration or a list declaration.

Initializing variables

Declaring a variable also initializes it to a default value.

- Scalar variables are initialized according to their data type:
  - Numeric data types (Boolean, Byte, Integer, Long, Single, Double, Currency): Zero (0)
  - Variants: EMPTY
  - Fixed-length strings: A string filled with the NULL character Chr(0)
  - Variable-length strings: The empty string (“”)
- Object reference variables are initialized to NOTHING, unless New is specified in the variable declaration.
- Each member of a user-defined data type variable is initialized according to its own data type.
- Each element of an array variable is initialized according to the array’s data type.
- A list variable has no elements when it is declared, so there is nothing to initialize.

Visibility of declarations

The default visibility for a declaration at the module level is Private, unless Option Public has been specified.

The default visibility for a variable declaration within a class is Private.

Public and Private can only be used to declare variables in module or class scope. Variables declared within a procedure are automatically Private; members of user-defined data types are automatically Public. Once created, these cannot be changed.

Examples: Dim statement

Example 1

```
' Declare a one-dimensional Integer array and a Single
' variable.
Dim philaMint(5) As Integer
Dim x As Single
x! = 10.0
philaMint%(0) = 3                ' Assigns an Integer value
philaMint%(1) = x                ' Converts Single 10.0 to
Integer 10
Print DataType(philaMint%(0)); DataType(philaMint%(1))
' Output:
' 2  2
' Both values are Integers.
```

Example 2

```
Dim xB As New Button("Merge", 60, 125)
```

xB is declared as an object reference variable to hold references to objects of the class named Button. A new Button object is created. For this example, suppose that the constructor sub for the class Button takes three arguments: a name for a button, and x- and y-position coordinates for the location of the button. The new button created is named "Merge," and positioned at (60, 125). A reference to this button is assigned to xB.

Example 3

```
' Declare iVer and kVer as Integer variables. Note that
' the phrase As Integer must be repeated to declare both
' variables as Integer.
Dim iVer As Integer, kVer As Integer
' Declare nVer as an Integer variable.
' The declared type of mVer is Variant, the default
' data type, because no data type is declared for mVer:
' there is no As type phrase for mVer, and no data type
' suffix attached to mVer.
Dim mVer, nVer As Integer
Print TypeName(mVer), TypeName(nVer%)         ' Prints EMPTY
INTEGER
```

Example 4

```
' Declare marCell and perDue as Integer variables.
' The phrase As Integer declares marCell as an Integer
' variable. The data type suffix % declares perDue as an
' Integer variable.
Dim marCell As Integer, perDue%
Print TypeName(marCell), TypeName(perDue%)    ' Prints INTEGER
INTEGER
```

Example 5

```
Dim marCell% As Integer
' Error, because the Dim statement attempts to declare
' the Integer variable marCell using both the data type
' suffix character for Integer, and the data type name
' Integer. The declaration should include one or the
' other, but not both.
```

Example 6

```
' A data type suffix character is optional in references to a
' declared variable.
' Declare marCell as an Integer variable.
Dim marCell As Integer
' Use the data type suffix character in a reference to
' marCell.
```

```
marCell% = 1
' Refer to marCell without using the suffix character.
Print marCell                        ' Prints 1
```

Example 7

```
' Declare marCell as an Integer variable.
Dim marCell As Integer
' Assign integer value to marCell.
marCell% = 1
Print marCell$
' Error, because the Print statement refers to marCell
' using the data type suffix character $ for a String
' variable, but marCell was declared as an Integer.
```

Example 8

```
Dim db As New NotesDatabase ("Server003", "discuss.nsf")
```

This Dim *objRef* As New *prodClass*(*argList*) statement declares an object reference to, and creates an instance of, the Notes/Domino NotesDatabase class. The Dim statement for creating a NotesDomino object requires two string arguments: a server name and a database path name.

Dir function

Returns file or directory names from a specified directory, or returns a drive volume label.

Syntax

Dir[**$**] [**(** *fileSpec* [**,** *attributeMask*] **)**]

Elements

fileSpec

A string expression that specifies a path and the file names you want returned. The argument is required only for the first call to Dir$ for any path.

Standard wildcard characters can be used in *fileSpec* to designate all files satisfying the wildcard criterion. Asterisk (*) for either the file name or the extension designates all files with any characters in that position. Question mark (?) in any character position in either part of the name designates any single character in that position.

attributeMask

An integer expression whose value specifies what names should be returned. If this argument is omitted, the names of normal files that match *fileSpec* are returned. If you supply an *attributeMask* argument, you must supply a *fileSpec* argument.

Dir$ always returns the names of normal files. To include other files in the returned list of file names, specify the sum of those values in the following table that correspond to the desired kinds of files:

Mask	*File attribute*	*Constant*
0	Normal file	ATTR_NORMAL
2	Hidden file	ATTR_HIDDEN
4	System file	ATTR_SYSTEM
8	Volume label	ATTR_VOLUME. If any other attribute is specified, ATTR_VOLUME is ignored.
16	Directory	ATTR_DIRECTORY

Return value

Dir returns a Variant of DataType 8 (String), and Dir$ returns a String.

Note On any platform except Windows (16, 9x, NT, 2000) if you ask for just the volume label you will get an empty string.

Usage

The constants in the table are defined in the file lsconst.lss. Including this file in your script allows you to use constant names instead of their numeric values.

To determine whether a particular file exists, use an exact file name for the *file_spec* argument to Dir or Dir$. The return value is either the file name or, if the file does not exist, the empty string ("").

The first call to Dir or Dir$ returns the name of the first file in the specified directory that fits the file name specifications in the *fileSpec* argument. Then:

- Subsequent calls to Dir or Dir$ without an argument retrieve additional file names that match *fileSpec*. You can call the Dir function with no arguments as either Dir or Dir( ). You can call the Dir$ function with no arguments as either Dir$ or Dir$().
- If there are no more file names in the specified directory that match the specification, Dir returns a Variant of DataType 8 (String); Dir$ returns the empty string ("").

If Dir or Dir$ is called without an argument after the empty string has been returned, LotusScript generates an error.

The Dir or Dir$ function may not be called recursively.

Examples: Dir function

```
' List the contents of the c:\ directory, one entry per line.
Dim pathName As String, fileName As String
pathName$ = "c:\*.*"
fileName$ = Dir$(pathName$, 0)

Do While fileName$ <> ""
   Print fileName$
   fileName$ = Dir$()
Loop
```

Do statement

Executes a block of statements repeatedly while a given condition is true, or until it becomes true.

Syntax 1

Do [**While** | **Until** *condition*]

[*statements*]

Loop

Syntax 2

Do

[*statements*]

Loop [**While** | **Until** *condition*]

Elements

condition

Any numeric expression. 0 is interpreted as FALSE, and any other value is interpreted as TRUE.

Usage

In Syntax 1, *condition* is tested before entry into the loop, and before each subsequent repetition. The loop repeats as long as *condition* evaluates to TRUE (if you specify **While**), or until *condition* evaluates to TRUE (if you specify **Until**).

In Syntax 2, *condition* is tested after the body of the loop executes once, and after each subsequent repetition. The loop repeats as long as *condition* evaluates to TRUE (if you specify **While**), or until *condition* evaluates to TRUE (if you specify **Until**).

Terminating the loop

You can exit the loop with an Exit Do statement or a GoTo statement. Exit Do transfers control to the statement that follows the Do...Loop block; GoTo transfers control to the statement at the specified label.

If you do not specify a While or Until *condition*, the loop will run forever or until an Exit Do or a GoTo statement is executed within the loop. For example, this loop executes forever:

```
Do
   ' ...
Loop
```

Language cross-reference

@DoWhile function in formula language

@While function in formula language

Examples: Do statement

```
' Each loop below executes four times,
' exiting when the loop variable reaches 5.
Dim i As Integer, j As Integer
i% = 1
j% = 1
Do While i% < 5   ' Test i's value before executing loop.
   i% = i% + 1
   Print i% ;
Loop
' Output:
' 2  3  4  5
Do
   j% = j% + 1
   Print j% ;
Loop Until j% >= 5   ' Test j's value after executing loop.
' Output:
' 2  3  4  5
```

Dot notation

Use dot notation to refer to members of user-defined types, user-defined classes, and product classes.

Syntax 1

typeVarName.*memberName*

Syntax 2

objRefName.*memberName* [(*argList*)]

Elements

typeVarName

A variable of a user-defined data type.

memberName

A member of a user-defined type, a user-defined class, or a product class. Class members may include methods, properties, and variables.

objRefName

An object reference variable.

argList

Optional. A list of one or more arguments; some class methods and properties require an argument list.

Usage

Use dot notation to refer to the members of user-defined data types, user-defined classes, and product classes.

When referring to the currently selected product object, you may omit *objRefName*. In some cases, you can use bracket notation, substituting [*prodObjName*] for *objRefName*. For more information, see your Lotus software documentation.

Note that dot notation is interpreted differently when it appears within a With statement. See that topic for details.

Examples: Dot notation

In Notes/Domino, you use the NotesDatabase class to access a database. This example sets the value of the Title property and uses the GrantAccess method to adjust the database's access control list (ACL).

```
Dim db As New NotesDatabase("Server003", "discuss.nsf")
db.Title = "HQEVB Group Discussion"
Call db.GrantAccess("HQEVB Group", ACLLEVEL_AUTHOR)
```

Double data type

Specifies a variable that contains a double-precision floating-point value maintained as an 8-byte floating point value.

Usage

The Double suffix character for implicit type declaration is #.

Double variables are initialized to 0.

The range of Double values is -1.7976931348623158E+308 to 1.7976931348623158E+308, inclusive.

On UNIX platforms, the range is -1.7976931348623156E+308 to 1.797693134862315E+308, inclusive.

The smallest non-zero Double value (disregarding sign) is 2.2250738585072014E-308.

LotusScript aligns Double data on an 8-byte boundary. In user-defined types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Double data type

```
' Explicitly declare a Double variable.
Dim rate As Double
rate# = .85

' Implicitly declare a Double variable.
interest# = rate#

Print interest#           ' Prints .85
```

End statement

Terminates execution of the currently executing script.

Syntax

End [*returnCode*]

Elements

returnCode

Optional. An integer expression. The script returns the value of this expression to the Lotus software application that executed the script.

Usage

Some Lotus products do not expect a return value when an End statement executes. See the product's documentation. If the product does not expect a return value, you do not need to use *returnCode*. The product will ignore it if you do.

Language cross-reference

@Return function in formula language

Examples: End statement

```
' The End statement terminates execution of the script
' that is running when the function is called.
Function Func1 ()
   Print 1
   End                 ' Terminates program execution
   Print 2             ' Never executed
End Function           ' Ends the function definition
Func1
' Output:
' 1
```

Environ function

Returns information about an environment variable from the operating system.

Syntax 1

Environ[$] ({ *environName* | *n* } **)**

Elements

environName

A string of uppercase characters indicating the name of an environment variable.

n

A numeric value from 1 to 255, inclusive, indicating the position of an environment variable in the environment string table.

Return value

Environ returns a Variant, and Environ$ returns a String.

If you specify the environment variable by name with *environName*, LotusScript returns the value of the specified environment variable. If that environment variable is not found, LotusScript returns the empty string (""). If *environName* is the empty string or evaluates to NULL or EMPTY, LotusScript generates an error.

If you specify the environment variable by position with *n*, LotusScript returns the entire environment string, including the name of the environment variable. If *n* is larger than the number of strings in the environment string table, LotusScript returns the empty string ("").

If *n* is less than 1, greater than 255, an EMPTY Variant, or NULL, LotusScript generates an error.

Language cross-reference

@Environment function in formula language

ENVIRONMENT keyword in formula language

Examples: Environ function

The following example is specific to Windows. Microsoft Windows 3.1 stores temporary files in the directory defined by the Temp environment variable. This example makes the temp directory the current directory, and writes the string you enter to a file (MYAPP.TMP) in that directory. To determine the location of your temp directory, see the Set Temp command in your AUTOEXEC.BAT.

```
Dim TempDir As String, tempFile As Integer
Dim tempFileName As String, tempStuff As String
tempStuff$ = InputBox("Enter some temporary information")
TempDir$ = Environ("Temp")
ChDir TempDir$
tempFile% = FreeFile()
tempFileName$ = "myapp.tmp"
Open tempFileName$ For Output As tempFile%
Print #tempFile%, tempStuff$
Close tempFile%
```

EOF function

Returns an integer value that indicates whether the end of a file has been reached.

Syntax

EOF (*fileNumber* **)**

fileNumber

The ID number assigned to the file when it was opened.

Return value

The return value depends on the type of file that you are using. The following table shows the EOF return values for binary, random, and sequential file types.

File type	*EOF returns TRUE if:*	*EOF returns FALSE if:*
Binary	The last executed Get statement cannot read the amount of data (the number of bytes) requested.	It successfully reads the amount of data requested.
Random	The last executed Get statement cannot read an entire record.	It successfully reads an entire record.
Sequential	The end of the file has been reached.	The end of the file has not been reached.

Usage

The end of file is determined by the operating system (from the file length stored in the file system). A Ctrl+Z character (ASCII 26) is not considered an end-of-file marker for any type of file: sequential, random, or binary.

Examples: EOF function

```
' Open a file, print it, and close the file.
Dim text As String, fileNum As Integer
fileNum% = FreeFile()

Open "c:\config.sys" For Input As fileNum%
Do Until EOF(1)
   Line Input #1, text$
   Print text$
Loop
Close fileNum%
```

Erase statement

Deletes an array, list, or list element.

Syntax

Erase { *arrayName* | *listName* | *listName* **(** *tag* **)** }
[**,** { *arrayName* | *listName* | *listName* **(** *tag* **)** }]...

Elements

arrayName

An array or a Variant variable containing an array. *arrayName* can end with empty parentheses.

listName

A list or a Variant variable containing a list. *listName* can end with empty parentheses.

tag

The list tag of a list element to be erased from the specified list.

Usage

The following table shows how the Erase statement affects arrays and lists.

Item	*Effect of Erase statement*
Fixed array	Its elements are reinitialized.
Dynamic array	LotusScript removes all elements from storage and recovers the storage. The array retains its type, but has no elements. You must use ReDim to redeclare the array before referring to its elements again. If you used ReDim before it was erased, the array maintains the same number of dimensions.
List	LotusScript removes all elements from storage and recovers the storage. The list retains its type, but has no elements.
List element	The element no longer exists in the list.

Examples: Erase statement

```
' Use Erase to reinitialize the Integer elements of the
' array baseLine to zero.
Option Base 1
Dim baseLine(3) As Integer ' Declare the fixed array baseLine.
baseLine%(1) = 1           ' Assign values to baseLine.
baseLine%(2) = 2
baseLine%(3) = 6
Erase baseLine%            ' Erase baseLine.
Print baseLine%(1)         ' Prints 0.
```

Erl function

Returns the line number in the current script procedure where the current error occurred.

Syntax

Erl

Return value

Erl returns an Integer. It returns FALSE (0) if there is no current error, which signifies that the most recent error has been handled.

Usage

You can call the function as either Erl or Erl().

The line number returned by Erl is for the procedure handling the error. If a calling procedure contains an On Error statement and the called procedure does not, an error in the called procedure is reported at the line number of the Call statement or function reference in the calling procedure.

Examples: Erl function

```
'all lotuscript language error codes

%include "lserr.lss"

' all Notes Backend Class error codes

%include "lsxbeerr.lss"

Sub Initialize

      Dim c

      on error goto errhandler

      ' throw the "Type Mismatch" error

      Error ( ERRTYPEMISMATCH )

      exit sub

ErrHandler:

      Print "Got error " & Error$ & " on line " & cstr(Erl)

      resume next

End Sub
```

Err function

Returns the current error number.

Syntax

Err

Return value

Err returns an Integer. If there is no current error, Err returns FALSE (0).

Usage

The error number is set when an error occurs, or by the Err statement. Generally, the function Err is used within an error-handling routine.

You can call the function as either Err or Err().

Language cross-reference

@IsError function in formula language

Examples: Err function

This example uses the Err function, Err statement, Error function, and Error statement. The user is asked to enter a number between 1 and 100. If the user's entry cannot be converted to a 4-byte single, an error occurs. The example defines two additional errors for numeric entries not in the range 1 - 100.

```
Public x As Single
Const TOO_SMALL = 1001, TOO_BIG = 1002
Sub GetNum
   Dim Num As String
   On Error GoTo Errhandle
   Num$ = InputBox$("Enter a value between 1 and 100:")
   x! = CSng(Num$)    ' Convert the string to a 4-byte single.
   ' Check the validity of the entry.
   If x! < 1 Then
      Error TOO_SMALL, "The number is too small or negative."
   ElseIf x! > 100 Then
      Error TOO_BIG, "The number is too big."
   End If
   ' If the script gets here, the user made a valid entry.
   MessageBox "Good job! " & Num$ & " is a valid entry."
   Exit Sub
   ' The user did not make a valid entry.
   ' Display the error number and error message.
Errhandle:
   ' Use the Err function to return the error number and
   ' the Error$ function to return the error message.
   MessageBox "Error" & Str(Err) & ": " & Error$
   Exit Sub
End Sub
GetNum                    ' Call the GetNum sub.
```

Err statement

Sets the current error number.

Syntax

Err = *errNumber*

Elements

errNumber

A numeric expression whose value is an error number.

Usage

The Err statement sets the current error number to an error number you specify. This may be any number in the range 0 to 32767 inclusive.

Examples: Err statement

This example uses the Err function, Err statement, Error function, and Error statement. The user is asked to enter a number between 1 and 100. If the user's entry cannot be converted to a 4-byte single, an error occurs. The example defines two additional errors for numeric entries not in the range 1 - 100.

```
Public x As Single
Const TOO_SMALL = 1001, TOO_BIG = 1002
Sub GetNum
   Dim Num As String
   On Error GoTo Errhandle
   Num$ = InputBox$("Enter a value between 1 and 100:")
   x! = CSng(Num$)    ' Convert the string to a 4-byte single.
   ' Check the validity of the entry.
   If x! < 1 Then
      Error TOO_SMALL, "The number is too small or negative."
   ElseIf x! > 100 Then
      Error TOO_BIG, "The number is too big."
   End If
   ' If the script gets here, the user made a valid entry.
   MessageBox "Good job! " & Num$ & " is a valid entry."
   Exit Sub
   ' The user did not make a valid entry.
   ' Display the error number and error message.
Errhandle:
   ' Use the Err function to return the error number and
   ' the Error$ function to return the error message.
   MessageBox "Error" & Str(Err) & ": " & Error$
   Exit Sub
End Sub
GetNum                  ' Call the GetNum sub.
```

Error function

Returns an error message for either a specified error number or the current error.

Syntax

Error[**$**] [**(** *errNumber* **)**]

Elements

errNumber

A numeric expression whose value is an error number. If no *errNumber* is specified, LotusScript returns the message for the current (most recent) error.

Return value

Error returns a Variant, and Error$ returns a String. If no *errNumber* is specified, and there is no current error, the function returns the empty string ("").

You can call the Error function with no arguments as either Error or Error( ). You can call the Error$ function with no arguments as either Error$ or Error$( ).

Language cross-reference

@Error function in formula language

@IsError function in formula language

Examples: Error function

This example uses the Err function, Err statement, Error function, and Error statement. The user is asked to enter a number between 1 and 100. If the user's entry cannot be converted to a 4-byte single, an error occurs. The example defines two additional errors for numeric entries not in the range 1 - 100.

```
Public x As Single
Const TOO_SMALL = 1001, TOO_BIG = 1002
Sub GetNum
   Dim Num As String
   On Error GoTo Errhandle
   Num$= InputBox$("Enter a value between 1 and 100:")
   x! = CSng(Num$)    ' Convert the string to a 4-byte single.
   ' Check the validity of the entry.
   If x! < 1 Then
      Error TOO_SMALL, "The number is too small or negative."
   ElseIf x! > 100 Then
      Error TOO_BIG, "The number is too big."
   End If
```

```
   ' If the script gets here, the user made a valid entry.
   MessageBox "Good job! " & Num$ & " is a valid entry."
   Exit Sub
   ' The user did not make a valid entry.
   ' Display the error number and error message.
Errhandle:
   ' Use the Err function to return the error number and
   ' the Error$ function to return the error message.
   MessageBox "Error" & Str(Err) & ": " & Error$
   Exit Sub
End Sub
GetNum                    ' Call the GetNum sub.
```

Error statement

Signals an error number and its corresponding message.

Syntax

Error *errNumber* [**,** *msgExpr*]

Elements

errNumber

A numeric expression whose value is a LotusScript-defined error number or a user-defined error number.

msgExpr

Optional.

A string expression containing an error message. This string replaces any existing message associated with the error number.

Usage

If *errNumber* is a LotusScript-defined error number, this Error statement simulates a LotusScript error. If it is not, this statement creates a user-defined error. When the Error statement is executed, LotusScript behaves as if a run-time error has occurred. If no error handling is in effect (set up by an On Error statement) for the specified error, execution ends and an error message is generated.

The error message generated is *msgExpr* if it is specified. If *msgExpr* is omitted, the error message is the LotusScript error message for the specified error number, if that number designates a LotusScript error. Otherwise the message "User-defined error" is generated.

User-defined errors must be in the range of 1000-1999. See LSERR.LSS for a list of LotusScript errors.

Examples: Error statement

This example uses the Err function, Err statement, Error function, and Error statement. The On Error statement specifies which error the error-handling routine ErrTooHigh handles. The Error statement tests the routine. The user is asked to enter a number between 1 and 100. If the user's entry cannot be converted to a 4-byte single, an error occurs. The example defines two additional errors for numeric entries not in the range 1 to 100.

```
Public x As Single
Const TOO_SMALL = 1001, TOO_BIG = 1002
Sub GetNum
   Dim Num As String
   On Error GoTo Errhandle
   Num$= InputBox$("Enter a value between 1 and 100:")
   x! = CSng(Num$)     ' Convert the string to a 4-byte
single.
   ' Check the validity of the entry.
   If x! < 1 Then
      Error TOO_SMALL, "The number is too small or negative."
   ElseIf x! > 100 Then
      Error TOO_BIG, "The number is too big."
   End If
   ' If the script gets here, the user made a valid entry.
   MessageBox "Good job! " & Num$ & " is a valid entry."
   Exit Sub
   ' The user did not make a valid entry.
   ' Display the error number and error message.
Errhandle:
   ' Use the Err function to return the error number and
   ' the Error$ function to return the error message.
   MessageBox "Error" & Str(Err) & ": " & Error$
   Exit Sub
End Sub
GetNum                        ' Call the GetNum sub.
```

Evaluate function and statement

Execute a Lotus software application macro.

Syntax

Evaluate (*macro* [**,** *object*] **)**

Elements

macro

A string expression specifying the text of a Lotus software application macro, in the syntax that the product recognizes. Refer to the Lotus software documentation for the correct syntax of the macro.

If the macro text is in a constant or string literal, the Lotus software application needs to do only initial processing of the macro once, at compile time, while variable strings incur that processing each time the macro is evaluated.

object

Optional. The name of a product object. Refer to the product documentation to determine if the macro requires an object, and what the object is.

Return value

If the Lotus software application macro being executed returns a value, the Evaluate function returns a Variant containing that value. Otherwise, the function does not return a value.

Examples: Evaluate function and statement

```
' For each document in a Notes database, use a Notes macro to
' compute the average for a list of numeric entries in the
' NumberList field. Evaluate returns a Variant, and Notes
' macros return an array. In this case, the array contains
only
' 1 element (element 0). For more info, see the Notes
' documentation.

Sub Click(Source As Button)
   ' The macro text must be known at compile time.
   Const NotesMacro$ = _
"@Sum(NumberList) / @Elements(NumberList)"
   Dim result As Variant, j As Integer
   Dim db As New NotesDatabase("", "MYSALES.NSF")
   Dim dc As NotesDocumentCollection
   Dim doc As NotesDocument
   Set dc = db.AllDocuments
   For j% =  1 To dc.Count
      Set doc = dc.GetNthDocument(j%)
```

```
      result = Evaluate(NotesMacro$, doc)
      MessageBox("Average is " & result(0))
   Next
End Sub
```

Execute function and statement

Compiles and executes a text expression as a temporary module.

Statement Syntax
Execute *text*

Function Syntax
Execute (*text* **)**

Elements
text

A string expression specifying the text to be compiled and executed.

Return value
The Execute function returns one of the following values:

- The return code of an End statement, if one was executed.
- Zero (0), if no End statement was executed, or if the executed End statement had no return value.

Usage
LotusScript considers *text* a separate script, compiling and executing it as a temporary module that's unloaded as soon as execution finishes.

Variables declared in the calling script (where the Execute statement appears) are only accessible in the temporary module if they are declared Public. Both these Public variables, and variables declared Public in external modules used by the calling script, will be accessible automatically. To reference a local variable in the temporary module, use the CStr function to convert its value to a string, and then include the result in *text*.

Variables declared in the temporary module are not accessible outside of that script.

To delimit text that spans several lines or includes double-quote characters, use vertical bars (| |) or braces ({ }).

Any compilation error in the temporary module will be reported as a run-time error in the scope containing the Execute statement. Any run-time error in the temporary module will be reported as a run-time error within the scope of that module, not the scope containing the Execute statement. To handle run-time errors within the temporary module, use the On Error statement.

The Execute statement is not legal at the module level; you can use it only in procedures.

Note In Lotus Notes, if you modify a global variable in an Execute statement, the variable must be defined in the (Declarations) event for (Global), not the (Declarations) event for the object containing the script.

Examples: Execute function and statement

Example 1 (Execute statement)

```
' The Execute statement performs a calculation entered by the
' user and displays the result. If the user enters an invalid
' calculation, a compilation error occurs, and the DoCalc sub
' displays an appropriate message. The Option Declare
' statement disallows the implicit declaration of variables in
' the calculation. The user can enter 700 * 32, for example,
' or "My name is " & "Fred", or Today - 365, but an entry such
' as x / y generates an error.

Sub DoCalc
   ' To handle any compilation error in the Execute statement
   On Error GoTo BadCalc
   Execute |Option Declare
           Dim x ' x is a Variant to accept any calculation.
           x =  | & InputBox ("Enter your calculation") & |
           MessageBox "The result is " & x|
   Exit Sub
' Report an error and exit.
BadCalc:
   MessageBox "Not a valid calculation"
   Exit Sub
End Sub
DoCalc                                  ' Call the sub.
```

Example 2 (Execute function)

```
' You can use the Execute function to return an integer such
' as a status code. In this example, the Execute function
' performs the calculation entered by the user. If the result
' is less than 0 or greater than 1 (100%), Execute returns a
' status code, and the ComputeInterest sub displays an
' appropriate message.
```

```
Sub ComputeInterest
   Dim script As String, calc As String, retcode As Integer
   calc$ = InputBox("Compute loan interest (charge/loan)")
   script$ = _
      |Option Declare
      Sub Initialize
         Dim pct As Single
         pct! = | &  calc$ & |
         If pct! < 0 Then
            End -2                  ' -2 is a status code.
         ElseIf pct! > 1 Then
            End -3                  ' -3 is a status code.
         End If
         MessageBox("Interest is " & Format(pct!,"percent"))
      End Sub|
   retcode% = Execute (script$)
   If retcode% = -2 Then
      MessageBox("You computed a negative interest rate!")
   ElseIf retcode% = -3 Then
      MessageBox("You computed an excessive interest rate!")
   End If
End Sub
ComputeInterest                     ' Call the sub.
```

Exit statement

Terminates execution of the current block statement.

Syntax

Exit *blockType*

Elements

blockType

A keyword designating the type of the block statement for which execution is to be terminated. It must be one of the following keywords:

Do
For
ForAll
Function
Sub
Property

Usage

When LotusScript encounters this statement, it returns control to the scope containing the block statement for which execution is to be terminated.

An Exit statement of a particular type is legal only within an enclosing block statement. LotusScript returns control from the innermost block statement or procedure of that type.

However, the innermost block statement containing the Exit statement need not be of that type. For example, a function definition can include a For...Next block statement, and an Exit Function statement can appear within this statement. If LotusScript encounters the Exit Function statement during execution, control is returned immediately from the function, in which case the For...Next block statement is not executed to completion.

The following table shows the rules for transfer of control after the Exit statement.

Exit block type	*Execution continues*
Exit Do	At the first statement following the end of the Do block statement.
Exit For	At the first statement following the end of the For block statement.
Exit ForAll	At the first statement following the end of the ForAll block statement.
Exit Function	In the calling script, as it would from a normal return from the procedure.
Exit Sub	In the calling script, as it would from a normal return from the procedure.
Exit Property	In the calling script, as it would from a normal return from the procedure.

If you exit a function or a Property Get without assigning a value to the function or property variable, that function or property returns the initialized value of the variable. Depending on the data type of the function or property's return value, this value can be either 0, EMPTY, or the empty string ("").

Language cross-reference

@Return function in formula language

Examples: Exit statement

```
' The user is asked to enter a 5-character string. If the
' length of the entry is not 5, the result of Exit Function is
' to return the empty string and issue a message telling you
' the entry is invalid.

Function AssignCode As String
```

```
   Dim code As String
   code$ = InputBox("Enter a 5-character code")
   If Len(code$) <> 5 Then Exit Function
   AssignCode = code$              ' It is a valid code.
End Function
If AssignCode() <> "" Then
   MessageBox "You entered a valid code."
Else
   MessageBox "The code you entered is not valid."
End If
```

Exp function

Returns the exponential (base *e*) of a number.

Syntax

Exp (*numExpr* **)**

Elements

numExpr

Any numeric expression, designating the power to which you wish to raise the value *e*.

If the value of *numExpr* exceeds 709.78, LotusScript returns an overflow error.

Return value

Exp returns the exponential (base *e*) of *numExpr.*

The data type of the return value is Double.

Usage

The value of *e* is approximately 2.71828182845905.

Exp is the inverse function of Log.

Language cross-reference

@Exp function in formula language

Examples: Exp function

```
Print Exp(2)            ' Prints  7.38905609893065
```

FileAttr function

Returns the access type, or the operating system file handle, for an open file.

Syntax

FileAttr (*fileNumber*, *attribute* **)**

Elements

fileNumber

The number associated with the file when you opened it.

attribute

A number (either 1 or 2) specifying the type of information you want. Instead of 1 or 2, you can specify the constant ATTR_MODE or ATTR_HANDLE, respectively. These constants are defined in the file lsconst.lss. Including this file in your script allows you to use constants instead of their numeric values.

Return value

If *attribute* is ATTR_HANDLE, then FileAttr returns the operating system file handle for the file.

If *attribute* is ATTR_MODE, then FileAttr returns an integer representing the access for the file, as shown in the following table.

Return value	*Access*	*Constant*
1	Input	ATTR_INPUT
2	Output	ATTR_OUTPUT
4	Random	ATTR_RANDOM
8	Append	ATTR_APPEND
32	Binary	ATTR_BINARY

Examples: FileAttr function

```
' The following example creates a file and displays its
' attributes.

%Include "lsconst.lss"

Dim mode As String, msg As String
Dim hdl As Integer, fileNum As Integer
fileNum% = FreeFile()

Open "data.txt" For Append As fileNum%
hdl% = FileAttr(fileNum%, ATTR_HANDLE)

Select Case FileAttr(fileNum%, ATTR_MODE)
   Case 1 : mode$ = "Input"
```

```
   Case 2 : mode$ = "Output"
   Case 4 : mode$ = "Random"
   Case 8 : mode$ = "Append"
   Case 32 : mode$ = "Binary"
End Select

Close fileNum%
Print "DOS File Handle = "; hdl%; "Mode = "; mode$
```

FileCopy statement

Makes a copy of a file.

Syntax

FileCopy *source* **,** *destination*

Elements

source

A string expression containing the name of the file you want to copy. The expression can optionally include a path.

destination

A string expression containing the name to be given to the copy. The expression can optionally include a path.

Usage

The file being copied must not be open.

The *source* and *destination* strings cannot include wildcard characters.

If *destination* names a file that already exists, the copy replaces the existing file with that name. To prevent this, you can use the Dir function to determine whether a file with the name *destination* already exists. Or, use the SetFileAttr statement to set the read-only attribute for the file.

Examples: FileCopy statement

This example is specific to Windows:

```
' Copy C:\WINDOWS\APP.BAT to the root directory of drive C:
and
' name the copy APPLOAD.BAT.
FileCopy "C:\WINDOWS\APP.BAT", "C:\APPLOAD.BAT"
```

FileDateTime function

Returns a string showing the date and time that a file was created or last modified.

Syntax

FileDateTime (*fileName* **)**

Elements

fileName

A string expression; you can include a path. *fileName* cannot include wildcard characters.

Return value

The returned date and time appear in the default format based on the operating system's international settings. If the file doesn't exist, FileDateTime returns an error.

Examples: FileDateTime function

```
' This script creates a file called data.txt
' and prints its creation date and time.

%Include "lsconst.lss"

Dim fileName As String, fileNum As Integer
fileNum% = FreeFile()
fileName$ = "data.txt"

Open fileName$ For Output As fileNum%   ' Create data.txt
file.
Close fileNum%
Print fileName$; " Created on "; FileDateTime(fileName$)
```

FileLen function

Returns the length of a file in bytes.

Syntax

FileLen (*fileName* **)**

Elements

fileName

A string expression; you can optionally include a path. The *fileName* cannot contain wildcard characters.

Return value

FileLen returns a Long value.

Examples: FileLen function

```
' Assign the length (in bytes) of the file c:\config.sys
' to the variable verLen, and print the result.
Dim verLen As Long
verLen& = FileLen("c:\config.sys")
Print verLen&
```

Fix function

Returns the integer part of a number.

Syntax

Fix (*numExpr* **)**

Elements

numExpr

Any numeric expression.

Return value

Fix returns the value of its argument with the fractional part removed. The data type of the return value is determined by the data type of *numExpr*. The following table shows special cases.

numExpr	*Return value*
NULL	NULL
Variant containing a string interpretable as a number	Double
Variant containing a date/time value	The date part of the value

Usage

The Fix function rounds toward 0:

- For a positive argument, Fix returns the nearest integer less than or equal to the argument (if the argument is between 0 and 1, Fix returns 0).
- For a negative argument, Fix returns the nearest integer larger than or equal to the argument (if the argument is between 0 and -1, Fix returns 0).

The Fix function and the Int function behave differently. The return value from Int is always less than or equal to its argument.

Tip It is always true that Fix(*numExpr*) + fraction(*numExpr*) = *numExpr*.

Examples: Fix function

```
Dim xF As Integer, yF As Integer
Dim xT As Integer, yT As Integer
xF% = Fix(-98.8)
yF% = Fix(98.2)
xT% = Int(-98.8)
yT% = Int(98.2)
Print xF%; yF%
' Output:
' -98  98
Print xT%; yT%
' Output:
' -99  98
```

This example shows the relationship between Fix() and Fraction().

```
' Print PI

Print PI                ' Prints 3.14159265358979

' Print the integer part of PI

Print Fix(PI)           ' Prints 3

' Print the fractional part of PI

Print Fraction(PI)    ' Prints .141592653589793
```

For statement

Executes a block of statements a specified number of times.

Syntax

For *countVar* **=** *first* **To** *last* [**Step** *increment*]

[*statements*]

Next [*countVar*]

Elements

countVar

A variable used to count repetitions of the block of statements. The data type of *countVar* should be numeric.

first

A numeric expression. Its value is the initial value of *countVar*.

last

A numeric expression. Its value is the final value of *countVar*.

increment

The value (a numeric expression) by which the *countVar* is incremented after each execution of the statement block. The default value of *increment* is 1. Note that *increment* can be negative.

Usage

After exit from a loop, the *countVar* for the loop has its most recent value.

Executing the loop the first time

Before the block of statements is executed for the first time, *first* is compared to *last*. If *increment* is positive and *first* is greater than *last*, or if *increment* is negative and *first* is less than *last*, the body of the loop isn't executed. Execution continues with the first statement following the For loop's terminator (Next).

Otherwise *countVar* is set to *first* and the body of the loop is executed.

Executing the loop more than once

After each execution of the loop, *increment* is added to *countVar*. Then *countVar* is compared to *last*. When the value of *countVar* is greater than *last* for a positive *increment*, or less than *last* for a negative *increment*, the loop is complete and execution continues with the first statement following the For loop's terminator (Next). Otherwise the loop is executed again.

Exiting the loop early

You can exit a For loop early with an Exit For statement or a GoTo statement. When LotusScript encounters an Exit For, execution continues with the first statement following the For loop's terminator (Next). When LotusScript encounters a GoTo statement, execution continues with the statement at the specified label.

Nested For loops

You can include a For loop within a For loop, as in the following example:

```
Dim x As Integer
Dim y As Integer
For x% = 1 To 3
   For y% = 1 To 2
      Print x% ;
   Next         ' Next y
Next            ' Next x
' Output: 1 1 2 2 3 3
```

If you don't include *countVar* as part of a For loop terminator (Next), LotusScript matches For loop delimiters from the most deeply nested to the outermost.

LotusScript lets you combine For loop terminators when they are contiguous, as in the following example:

```
Dim x As Integer
Dim y As Integer
For x% = 1 To 3
   For y% = 1 To 2
      Print x% ;
Next y%, x% 'Terminate the inner loop and then the outer loop.
' Output: 1 1 2 2 3 3
```

Language cross-reference

@For function in formula language

Examples: For statement

```
' Compute factorials for numbers from 1 to 10
Dim m As Long
Dim j As Integer
m& = 1
For j% = 1 To 10
   m& = m& * j%
   Print m&
Next
' Output:
' 1  2  6  24  120  720  5040  40320  362880  3628800
```

ForAll statement

Executes a block of statements repeatedly for each element of an array, a list, or a collection. A collection is an instance of a product collection class or an OLE collection class.

Note ForAll works on Product collections; it does not support Notes collections.

Syntax

ForAll *refVar* **In** *container*

[*statements*]

End ForAll

Elements

refVar

A reference variable for the array, list, or collection element. In the body of the ForAll loop, you use *refVar* to refer to each element of the array, list, or collection named by *container*. *refVar* can't have a data type suffix character appended.

container

The array, list, or collection whose elements you wish to process.

Usage

On entry to the loop, *refVar* refers to the first element of the array, list, or collection. On each successive iteration, *refVar* refers to the next element of the array, list, or collection. Upon completion of the loop, execution continues with the first statement following the loop's End ForAll statement.

Note If you're using ForAll on an array of arrays, do not ReDim the iterator (this generates the "Illegal ReDim" error).

Exiting the loop early

You can force the loop to be exited early with the Exit ForAll statement or the GoTo statement. When LotusScript encounters an Exit ForAll statement, execution immediately continues with the first statement following the loop's terminator (**End ForAll**). When LotusScript encounters a GoTo statement, execution immediately continues with the statement at the specified label.

Using *refVar*

Since *refVar* is an alias for the actual array, list, or collection element, you can change the value of the element to which it refers by assigning a new value to *refVar*. For example:

```
ForAll x In y
   x = x + 1
End ForAll
```

This adds 1 to the value of each element in the array, list, or collection named y.

If *container* is a list, you can pass *refVar* to the ListTag function to get the name (the list tag) of the list element that *refVar* currently refers to. For example:

```
Print ListTag(refVar)
```

Because *refVar* is implicitly defined by the ForAll statement, you should not include it in your variable declarations. The scope of *refVar* is the loop, so you can't refer to it from outside of the loop.

If container is an array or list, *refVar* has the data type of the array or list being processed. If this data type cannot be determined by LotusScript at compile time or if *container* is a collection, *refVar* is a Variant. In that case, the data type of the array or list cannot be a user-defined data type, because Variants cannot be assigned values of a user-defined data type.

You can reuse a *refVar* in a subsequent ForAll loop, provided that the data type of the container matches that of the container in the ForAll loop where *refVar* was first defined.

You can't use the ReDim statement on the reference variable. For example, suppose that zArr is an array of arrays, and a ForAll statement begins:

```
ForAll inzArr In zArr
```

Then the statement ReDim inzArr(2) generates an error.

Language cross-reference

@Transform function in formula language

Examples: ForAll statement

Example 1

```
Dim myStats List As Variant
myStats("Name") = "Ian"
myStats("Age") = 29
ForAll x In myStats
   Print ListTag(x); " = "; x
End ForAll
' Output:
' Name = Ian
' Age =  29
```

Example 2

```
Dim minima(5) As Integer
minima%(0) = 5
minima%(1) = 10
minima%(2) = 15
' Set all elements of array minima to 0.
ForAll x In minima%
   x = 0
End ForAll
```

Example 3

In Freelance Graphics, an Application object contains a DocumentCollection object. The DocumentCollection object contains a collection of Document objects. Each Document object contains a PageCollection object. Each PageCollection object contains a number of Page objects. Each Page object contains an ObjectCollection object. ObjectCollection is a heterogenous collection that may include TextBox objects.

In addition to For loops, you can use ForAll loops or indexing to access individual members of a collection class. This example uses three nested ForAll loops to iterate through the collections. Within individual TextBlock

objects, the script uses indexing to set list entries at levels 2 through 5 in each TextBox object to Italic.

```
Dim level As Integer
ForAll doc In [Freelance].Documents
     ForAll pg In Doc.Pages
        ForAll obj In Pg.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
             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
```

The Application class Documents property returns an instance of the DocumentCollection class. Each element in the collection is a document, an instance of the Document class.

The Document class Pages property returns an instance of the PageCollection class. Each element in the collection is a page, an instance of the Page class.

The Page Objects property returns an instance of the ObjectCollection class. Some of the elements in this collection may be text blocks, instances of the TextBox class.

Format function

Formats a number, a date/time, or a string according to a supplied format.

Syntax

Format[$] (*expr* [**,** *fmt*] **)**

Elements

expr

Any expression. The expression is evaluated as a numeric expression if *fmt* is a numeric format, as a date/time if *fmt* is a date/time format, and as a string if *fmt* is a string format.

fmt

Optional. A format string: either a string consisting of the name of a format pre-defined in LotusScript, or else a string of format characters. If this format string is not supplied, Format[$] behaves like Str[$], omitting the leading space character for positive numbers.

Return value

Format returns a Variant containing a string, and Format$ returns a String.

If *expr* is a string and *fmt* is a numeric format string, LotusScript attempts to convert the string to a number. If successful, LotusScript then formats the result.

If the string can't be converted to a number, LotusScript attempts to interpret it as a date/time, and attempts to convert it to a numeric value. If successful, LotusScript then formats the result.

If *expr* can't be converted to the data type of the format string, Format returns *expr* without formatting it.

Formatting codes

Numeric formats

If *expr* is numeric, you can use one of the named numeric formats shown in the following section, or create a custom numeric format using the numeric formatting codes shown in the subsequent section.

Named numeric formats

Format name	*Display of the value of expr is ...*
General Number	As stored, without thousands separators
Currency	As defined in the operating system's international settings. For example, you can format currency values with thousands separators, negative values in parentheses, and two digits to the right of the decimal separator. In OS/2, the function does not append the currency symbol to the number.
Fixed	With at least one digit to the left of the decimal separator, and with two digits to the right of the decimal separator.
Standard	With thousands separators, with at least one digit to the left of the decimal separator, and with two digits to the right of the decimal separator.

continued

Format name	*Display of the value of expr is ...*
Percent	*expr* multiplied by 100, with at least one digit to the left of the decimal separator. Two digits are displayed to the right of the decimal separator, and a percent sign (%) follows the number.
Scientific	In standard scientific notation: with one digit to the left of the decimal separator and two digits to the right of the decimal separator, followed by the letter *E* or *e* and a number representing the exponent.
Yes/No	No if the number is 0, and Yes otherwise.
True/False	False if the number is 0, and True otherwise.
On/Off	Off if the number is 0, and On otherwise.

Custom numeric formatting codes

The following table describes the characters you can use in *fmt* to create custom formats for numeric values.

Formatting code	*Meaning*
"" (Empty string)	Display the number with no formatting
0 (zero)	Digit forced display. A digit is displayed for each zero in *fmt*, with leading or trailing zeros to fill unused spaces. All digits to the left of the decimal separator are displayed. If the number includes more decimal places than *fmt*, it is rounded appropriately.
# (pound sign)	Digit conditional display. The same display as 0 (digit forced display), except that no leading or trailing zeros are displayed.
. (period)	Decimal separator. The position of the decimal separator in *fmt*. Unless your formatting code includes a 0 immediately to the left of the decimal separator, numbers between -1 and 1 begin with the decimal separator, The actual decimal separator used in the returned formatted value is the decimal separator specified in the operating system's international settings.
% (percent sign)	Percentage placeholder. Multiplies the number by 100 and inserts the percent sign (%) in the position where it appears in *fmt*. If you include more than one percentage placeholder, the number is multiplied by 100 for each %. For example, %% means multiplication by 10000.

continued

Formatting code	*Meaning*
, (comma)	Thousands separator. To separate groups of three digits, counting left from the decimal separator, within numbers that include at least four digits to the left of the decimal separator, enclose the comma between a pair of the digit symbols 0 or #. The actual thousands separator used in the returned formatted value is the thousands separator specified in the operating system's international settings.
	A special case is when the comma is placed immediately to the left of the decimal separator (or the position of the implied decimal separator). This causes the number to be divided by 1000. For example, this returns "100": x = **`Format$(100000,"##0,.")`** If 100000 is replaced in this example by a number less than 1000 in absolute value, then this function returns "0."
E- E+ e- e+	Scientific notation. The number of digit symbols (0 or #) to the left of the decimal separator specifies how many digits are displayed to the left of the decimal separator, and the resulting magnitude of the exponent.
	Use E+ or e+ to display the sign of all exponents (the symbol + or -). Use E- or e- to display the sign of negative exponents only (the symbol -).
	All exponent digits are displayed, regardless of how many digit symbols follow the E-, E+, e-, or e+. If there are no digit symbols (the symbol 0 or #), an exponent of zero is not displayed; otherwise at least one exponent digit is displayed. Use 0 to format a minimum number of exponent digits, up to a maximum of three.
$ (dollar sign)	Currency symbol. Designates a currency value. The actual currency symbol used in the returned formatted value is the currency symbol specified in the operating system's international settings.
- + ( ) space	Literal characters. These are displayed as they appear in the format string.
\ (backslash)	Literal character prefix. The character following the backslash is displayed as is; for example, \# displays #. To display a backslash itself, precede it with another backslash; that is, \\ displays \.

continued

Formatting code	*Meaning*
"ABC"	Literal string enclosed in double quotation marks. To specify the double quotation mark character in the *fmt* argument, you must use Chr(34). The characters enclosed in quotation marks are displayed as they appear in the format string.
; (semicolon)	Format section separator. Separates the positive, negative, zero, and NULL sections in *fmt*. If you omit the negative or zero format sections, but include the semicolons representing them, they are formatted like the positive section.

A custom format string for numeric values can have from one to four sections, separated by semicolons. In a format string with more than one section, each section applies to different values of *expr*. The number of sections determines the values to which each individual section applies. The following table describes how each section of a one-part or multi-part format string is used.

Number of sections	*Description*
One	The format applies to all numbers.
Two	The first section formats positive numbers and 0. The second section formats negative numbers.
Three	The first section formats positive numbers. The second section formats negative numbers. The third section formats 0.
Four	The first section formats positive numbers. The second section formats negative numbers. The third section formats 0. The fourth section formats NULL.

Date/time formats

Since date/time values are stored as floating point numbers, date/time values can be formatted with numeric formats. They can also be formatted with date/time formats. You can either use one of the named date/time formats shown in the following section, or create a custom date/time format using the date/time formatting codes shown in the subsequent section.

Named date/time formats

Format name	*Display of the date/time value is ...*
General Date	In a standard format. Converts a floating-point number to a date/time. If the number includes no fractional part, this displays only a date. If the number includes no integer part, this displays only a time.
Long Date	A Long Date as defined in the operating system's international settings.
Medium Date	dd-mmm-yy (yy/mmm/dd in Japan)
Short Date	A Short Date as defined in the operating system's international settings.
Long Time	A Long Time as defined in the operating system's international settings. Long Time always includes hours, minutes, and seconds.
Medium Time	Hours (0 - 12) and minutes using the time separator and AM/PM notation (AMPM notation in Japan)
Short Time	Hours (0 - 23) and minutes using only the time separator.

Custom date/time formatting codes

The following table describes the characters you can use in *fmt* to create custom formats for date/time values.

Formatting code	*Meaning*
: (colon)	Time separator. Separates hours, minutes, and seconds in formatted time values. The actual time separator used in the returned formatted value is the time separator specified for the given country in the operating system's international settings.
/ (slash)	Date separator. Separates day, month, and year in formatted date values. The actual date separator used in the returned formatted value is the date separator specified in the operating system's international settings.
c	Displays a date as ddddd, and a time as ttttt (see below). If the value includes no fractional part, only a date is displayed. If the value includes no integer part, only a time is displayed.
y	Day of the year as a number (1 - 366).
d	Day of the month as a number without a leading zero (1 - 31).
dd	Day of the month as a number with a leading zero (01 - 31).
ddd	Weekday as a three-letter abbreviation (Sun - Sat).
dddd	Weekday spelled out (Sunday - Saturday).

continued

Formatting code	*Meaning*
ddddd	Serial date number as a complete date (day, month, and year) formatted as an international Short Date string. If there is no Short Date string provided in the operating system, the date format defaults to mm/dd/yy.
dddddd	Serial date number as a complete date (day, month, and year) formatted as an international Long Date string. If there is no Long Date string provided in the operating system, the date format defaults to mmmm dd, yyyy.
w	Weekday as a number (1 - 7). Sunday is 1.
ww	Week of the year as a number (1 - 53).
m	Month of the year as a number without a leading zero (1 - 12). If the character is preceded by h in *fmt*, it displays the minute of the hour as a number without a leading zero (0 - 59).
mm	Month of the year as a number with a leading zero (01 - 12). If the character is preceded by h in *fmt*, it displays the minute of the hour as a number with a leading zero (00 - 59).
mmm	Month name as a 3-letter abbreviation (Jan - Dec).
mmmm	Month name spelled out (January - December).
q	Quarter of the year as a number (1 - 4).
yy	The last two digits of the year (00 - 99). If you specify yy in Notes or Domino, LotusScript interprets 50 through 99 as the years 1950 through 1999 and 00 through 49 as the years 2000 through 2049. Note that SmartSuite interprets yy differently.
yyyy	The full (four-digit) year (0100 - 9999).
h	Hour of the day as a number without a leading zero (0 - 23).
hh	Hour of the day as a number with a leading zero (00 - 23).
n	Minute of the hour as a number without a leading zero (0 - 59).
nn	Minute of the hour as a number with a leading zero (00 - 59).
s	Second of the minute as a number without a leading zero (0 - 59).
ss	Second of the minute as a number with a leading zero (00 - 59).
ttttt	Time serial number as a complete time (including hour, minute, and second), formatted using the time separator provided in the operating system's international settings. A leading zero is displayed if the international leading zero setting is TRUE and the time is before 10:00 AM or PM. The default time format is h:mm:ss.

continued

Formatting code	*Meaning*
AM/PM am/pm	Uses hour values from 1 to 12, displaying AM or am for hours before noon, and PM or pm for hours after noon.
A/P a/p	Uses hour values from 1 to 12, displaying A or a for hours before noon, and P or p for hours after noon.
AMPM	Uses hour values from 1 to 12. Displays the contents of the 1159 string (s1159) in WIN.INI for hours before noon, and the contents of the 2359 string (s2359) for hours after noon. AMPM is case-insensitive, but the case of the string displayed matches the string as it exists in the operating system's international settings. The default format is AM/PM.

The following table shows some custom date/time formats applied to one date and time: 6:43:04 in the evening of April 12, 1995.

fmt	*Display*
m/d/yy	4/12/95
d-mmm-yy	12-Apr-95
d-mmmm	12-April
mmmm-yy	April-95
y	102.00
hh:mm AM/PM	06:43 PM
h:mm:ss a/p	6:43:04 p
h:mm	18:43
h:mm:ss	18:43:04
m/d/yy h:mm	4/12/95 18:43

String formatting codes

To format a string using Format or Format$, use the formatting codes in the following table to create a custom string format. There are no named string formats.

Custom string formats can have one section, or two sections separated by a semicolon (;). If the format has one section, the format applies to all strings. If the format has two sections, then the first applies to nonempty strings, and the second applies to the value NULL and the empty string ("").

The following table describes the characters you can use in *fmt* to create a custom string format.

Formatting code	*Meaning*
@ (at sign)	Character forced display.
	If the string being formatted includes a character in this position, display it. If not, display a space. @ is filled from right to left unless *fmt* contains an exclamation point (!).
& (ampersand)	Character optional display. If the string being formatted includes a character in this position, display it. If not, display nothing. & is filled from right to left unless *fmt* contains an exclamation point (!).
< (less-than sign)	All characters in the formatted string are displayed in lowercase.
> (greater-than sign)	All characters in the formatted string are displayed in uppercase.
! (exclamation point)	Forces @ and & to fill from left to right, rather than from right to left.

Formatting dates and times in Asian languages

The Format function supports additional formatting characters for dates and times in versions of LotusScript for Japan, China, the Taiwan region, and Korea.

Only single-byte characters are recognized as formatting characters. Double-byte characters are treated as literal characters. Some of the formatting characters for LotusScript in China and the Taiwan region are case-sensitive (see the following paragraphs); all of the other Asian language date/time formatting characters are case-insensitive.

When a date/time formatting code used in the Format function in LotusScript for Japan is also a date/time formatting code in WIN.INI, LotusScript for Japan interprets the code appropriately. For example, the formatting expression "Long Date" has the same meaning in LotusScript for Japan as in English-language LotusScript. (The meaning is to use the WIN.INI Long Date format.)

These formats only have meanings in Asian versions of Lotus products.

Date/time format codes

The first table shows the formatting codes for Japan.

Formatting code	*Meaning*
aaa	Weekday in abbreviated format (one double-byte character)
aaaa	Weekday in full format
e	Year in era (“0” suppressed)
ee	Year in era (“0” not suppressed)
g	Era name (single-byte one-character abbreviation)
gg	Era name (double-byte one-character abbreviation)
ggg	Full era name

This table shows the formatting codes for People’s Republic of China.

Formatting code	*Meaning*
aaaa	Weekday in full format (three double-byte characters)
O	Month (double-byte)
o	Month (single-byte)
A	Day (double-byte)
a	Day (single-byte)
E	Short year (double-byte)
e	Short year (single-byte)
EE	Long year (double-byte)
ee	Year (single-byte)

This table shows the formatting codes for the Taiwan region.

Formatting code	*Meaning*
aaaa	Weekday in full format (three double-byte characters)
O	Month (double-byte)
o	Month (single-byte)
A	Day (double-byte)
a	Day (single-byte)
E	Year in era (double-byte)
e	Year in era (single-byte)
EE	Year in era with era abbreviation (double-byte)
ee	Year in era with era abbreviation (single-byte)

continued

Formatting code	*Meaning*
EEE	Year in era with era name (double-byte)
eee	Year in era with era name (single-byte)
EEEE	Christian year with Christian era name (double-byte)
eeee	Christian year with Christian era name (single-byte)

This table shows the formatting codes for Korea.

Formatting code	*Meaning*
aaa	Weekday in abbreviated format (one double-byte character)
aaaa	Weekday in full format (three double-byte characters)

Examples: Format function

Here are several examples of the Format function.

```
' Currency
' Get monthly revenue and expenses from the user, converting
' strings to currency. Compute and display the balance,
' formatted as currency.

Dim rev As Currency, expense As Currency, bal As Currency

rev@ = CCur(InputBox("How much did we make this month?"))

expense@ = CCur(InputBox("How much did we spend?"))

bal@ = rev@ - expense@

MessageBox "Our balance this month is " _

   & Format(bal@, "Currency")

' Percent

Dim total As Integer, attend As Integer, percent As Double

total% = CInt(InputBox("How many people registered?"))

attend% = CInt(InputBox("How many people actually attended?"))

percent# = attend% / total%

MessageBox "The attendance was " _          ' Use "Percent"
format

     & Format(percent#, "Percent")

MessageBox "The attendance was " _          ' Use custom
format codes

     & Format(percent#, "0%")        ' can carry % to any
number of places:
```

```
                                                    '  0.0%, 0.00%,
0.000%, and so on
' Example of custom formatting using sections
Dim x As Integer
x = 1
Print Format(x, "0.0;0%")      ' Output: 1.0
x = -1
Print Format(x, "0.0;0%")      ' Output: 100%
x = 0
Print Format(x, "0.0;0%;zippo")' Output: zippo
Print chr$(34) & Format(x, "zippo") & chr$(34)     ' Output:
"zippo"
' Yes/No, True/False, & On/Off
Dim value As Integer
value = 0
Print Format$(value, "Yes/No")        ' Output "No"
Print Format$(value, "On/Off")        ' Output "Off"
Print Format$(value, "True/False")    ' Output "False"
value = 2
Print Format$(value, "Yes/No")        ' Output "Yes"
Print Format$(value, "On/Off")        ' Output "On"
Print Format$(value, "True/False")    ' Output "True"
' Date and Time formats
x = 36525
Print Format(x, "General Date")       ' Output: 12/31/1999
Print Format(x, "Long Date")          ' Output: Friday,
December 31, 1999
Print Format(x, "Medium Date")        ' Output: 31-Dec-99
Print Format(x, "Short Date")         ' Output: 12/31/99
y = 123.45
Print Format(y, "Long Time")          ' Output: 10:48:00 AM
Print Format(y, "Medium Time")        ' Output: 10:48 AM
```

```
Print Format(y, "Short Time")         ' Output: 10:48

' Custom String formats

Dim x As String, y As String, z As String

' Format directory string

x = "dtemp"

Print Format(x, "!copy to directory @:\\@")

' Output: copy to directory d:\temp

'Use sections and NULL data

Print Format(y, "!copy to directory @:\\@;don't copy - no
valid data")

'Output: don't copy - no valid data

' Manipulate character positions

dim x as string, y as string, z as string

x = "hello world"

y = "hello"

z = "world"

Print Format(x, "!@")                 ' Output: hello world

print format(y, "&&&&&&&&&&" & z)

' Output: "helloworld" (& ignores non-characters)

print format(y, "!&&&&&&&&&&" & z)

' Output: "helloworld" (! matters not with &)

print format(y, "@@@@@@@@@@" & z)     ' Output: "
helloworld"

print format(y, "!@@@@@@@@@@" & z)    ' Output: "hello
world"

' Format all characters to lower then upper case

z = "Hello World"

Print Format(z, "<")     ' Output: hello world

Print Format(z, ">")     ' Output: HELLO WORLD
```

Fraction function

Returns the fractional part of a number.

Syntax
Fraction (*numExpr* **)**

Elements
numExpr
Any numeric expression.

Return value
The data type of the return value is the same as the data type of *numExpr*.

Usage
The following table shows special cases of the return value:

numExpr	*Return value*
A date/time value	The time portion
An integer	0
NULL	NULL

Tip It is always true that Fix(*numExpr*) + fraction(*numExpr*) = *numExpr*.

Examples: Fraction function
```
' Print the fractional part of PI
Print Fraction(PI)     ' Prints .141592653589793
```

This example shows the relationship between Fix() and Fraction().

```
' Print PI

Print PI                ' Prints 3.14159265358979

' Print the integer part of PI

Print Fix(PI)           ' Prints 3

' Print the fractional part of PI

Print Fraction(PI)    ' Prints .141592653589793
```

FreeFile function

Returns an unused file number.

Syntax
FreeFile

Return value
FreeFile returns an Integer value.

Usage
Use FreeFile when you need a file number (to open a file), but you don't know what file numbers are currently available.

If no more file numbers are available, an error is generated.

LotusScript limits the number of open files to 255. Depending on your operating system environment and the Lotus software you are running, the actual number of files that you can open may be 15 or less. See your product documentation for details.

You can call the function as either FreeFile or FreeFile().

Examples: FreeFile function
```
Dim fileNum As Integer
Dim cdr As String
cdr$ = CurDrive() + "\AUTOEXEC.BAT"
' Assign the lowest available file number to fileNum.
fileNum% = FreeFile()
Print FreeFile()              ' Prints 1 (1 is unused)
Open cdr$ For Input Access Read As fileNum%

' Use file number 1
Print FreeFile()              ' Prints 2 (1 is in use)
Close fileNum%
Print FreeFile()              ' Prints 1 (1 is unused again)
```

FullTrim function

Takes an array and eliminates "empty" entries, or takes a string and eliminates duplicate, trailing and leading whitespace.

Syntax

FullTrim(*v* **)**

Element

v

Any array, string, or variant containing a string.

Return value

A variant containing an array or string. If you pass in a string, you get back a string. If you pass in an array, you get back an array.

Usage

Empty for strings is the Empty string.

Empty for numbers is the value 0.

Empty for variants containing the above are the same, as well as NULL and Empty.

The FullTrim trims strings by eliminating any duplicate whitespaces (SPACE, TAB, NEWLINE) from the center of the string and all whitespace at the beginning and end of the strings.

The number of elements in the returned array may vary as empty elements are removed. If all the elements are removed, an array with one empty element is returned.

Extended example: array and String functions

Function statement

Defines a function.

Syntax

[**Static**] [**Public** | **Private**] **Function** *functionName* [([*paramList*])] [**As** *returnType*]

[statements]

End Function

Elements

Static

Optional. Specifies that the values of the function's local variables are saved between calls to the function.

Public | **Private**

Optional. Public specifies that the function is visible outside the scope (module or class) where the function is defined, as long as that remains loaded. Private specifies that the function is visible only within the current scope.

A function in module scope is Private by default; a function in class scope is Public by default.

functionName

The name of the function. This name can have a data type suffix character appended, to declare the type of the function's return value.

paramList

Optional. A comma-separated list of declarations indicating the parameters to be passed to this function in function calls.

The syntax for each parameter declaration is:

[**ByVal**] *parameter* [**( )** | **List**] [**As** *type*]

ByVal means that *parameter* is passed by value: that is, the value assigned to *parameter* is a local copy of a value in memory, rather than a pointer to that value.

*parameter***()** is an array variable. *parameter* **List** identifies *parameter* as a list variable. Otherwise, *parameter* can be a variable of any of the other data types that LotusScript supports.

As *dataType* specifies the variable's data type. You can omit this clause and append a data type suffix character to *parameter* to declare the variable as one of the scalar data types. If you omit this clause and *parameter* has no data type suffix character appended (and isn't covered by an existing Def*type* statement), its data type is Variant.

Enclose the entire list of parameter declarations in parentheses.

returnType

Optional. The data type of the value returned by the function.

returnType can be any of the scalar data types, or Variant, or a class name.

If **As** *returnType* is not specified, the function name's data type suffix character determines the return value's type. Do not specify both a

returnType and a data type suffix character; LotusScript treats that as an error.

If you omit *returnType* and the function name has no data type suffix character appended, the function returns a value either of data type Variant or of the data type specified by a Def*type* statement.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

Arrays, lists, type instances, and objects can't be passed by value as arguments. They must be passed by reference.

To return a value from a function, assign a value to *functionName* within the body of the function definition (see the example).

If you assign an array to *functionName*, you cannot refer to an element of *functionName* within the body of the function; such a reference will be taken as a recursive call of the function. To refer to an element of *functionName*, assign *functionName* to a variant variable and index the element there.

A function returns a value; a sub does not. To use the value returned by a function, put the function call anywhere in an expression where a value of the data type returned by the function is legal.

You don't have to use the value returned by a function defined by the Function statement. (The value returned by a built-in function must be used.) To call a function without using the return value, use the Call statement.

A function definition cannot contain another function or sub definition, or a property definition.

A function member of a class cannot be declared Static.

You can exit a function using an Exit Function statement.

Note If you're using a 32-bit version of Windows, an integer has four bytes; use the short integer (two bytes) to correspond to the LotusScript Integer when passing data to LotusScript. This note applies to Windows platforms only.

Examples: Function statement

Use a sub and a function to compute the cost of buying a house as follows:

- Ask the user for the price of the house, and call the ComputeMortgageCosts sub with price as the argument.
- The ComputeMortgageCosts sub gathers down payment (at least 10% of cost), annual interest rate, and the term of the mortgage from the user, then calls the Payment function with three arguments. Annual interest and term (years) are passed by value rather than reference so the Payment function can adjust them to compute monthly rate and monthly payment without changing the values of these variables in the ComputeMortgageCosts sub.
- If the user enters positive values, Payment returns the monthly payment. Otherwise, it returns 0. ComputeMortgageCosts then constructs an appropriate message.

```
Dim price As Single, message As String

Function Payment (princpl As Single, _
                  ByVal intrst As Single, _
                  ByVal term As Integer) As Single
   intrst! = intrst! / 12
   term% = term% * 12
   ' If any of the arguments are invalid, exit the function
   ' (payment will return the value 0).
   If princpl! <= 0 Or intrst! <= 0 Or term% < 1 Then _
      Exit Function
   ' The standard formula for computing the amount of the
   ' periodic payment of a loan:
   Payment = princpl! * intrst! / (1 - (intrst! + 1) ^ _
      (-term%))
End Function

Sub ComputeMortgageCosts (price As Single)
   Dim totalCost As Single, downpmt As Single
   Dim mortgage As Single, intrst As Single
   Dim monthlypmt As Single, years As Integer
EnterInfo:
   downpmt! = CSng(InputBox("How much is the down payment?"))
   ' The downpayment must be at least 10% of the price.
   If downpmt! < (0.1 * price!) Then
     MessageBox "Your down payment must be at least " _
         & Format(price! * .1, "Currency")
      GoTo EnterInfo
   Else
      mortgage! = price! - downpmt!
   End If
   intrst! = CSng(InputBox("What is the interest rate?"))
```

```
   years% = CInt(InputBox("How many years?"))
   ' Call the Payment function, which returns the
   ' monthly payment.
   monthlypmt! = Payment(mortgage!, intrst!, years%)
   totalCost! = downpmt! + (monthlypmt! * years% * 12)
   If monthlypmt! > 0 Then ' Create a multiline message.
      message$ = _
       |Price | & Format(price!, "Currency") & |
       Down Payment: | & Format(downpmt!, "Currency") & |
       Mortgage: | & Format(mortgage!, "Currency") & |
       Interest: | & Format(intrst!, "Percent") & |
       Term: | & Str(years%) & |
       years Monthly Payment: | & Format(monthlypmt!, _
       "Currency") & |
       Total Cost: | & Format(monthlypmt! * years% * 12, _
        "Currency")
   Else
      message$ = "You did not enter valid input."
   End If
End Sub

' Start here.
price! = CSng(InputBox("How much does the house cost?"))
' Call the Compute MortgageCosts sub.
ComputeMortgageCosts (price!)
' Display the message.
MessageBox message$
```

Get statement

Reads data from a binary file or a random file into a variable.

Syntax

Get #*fileNumber* **,** [*recordNumber*] **,** *variableName*

Elements

fileNumber

The number assigned to the file when it was opened with the Open statement. Note that the pound sign (#), *fileNumber,* and *variableName* are all required.

recordNumber

Optional. The file position (the byte position in a binary file, or the record number in a random file) where data retrieval begins. If you omit *recordNumber*, LotusScript retrieves data beginning from the current file position.

variableName

The variable to be used for storing the retrieved data. *variableName* cannot be an array. However, a fixed-length array defined within a data type is allowed (this array could also contain other arrays as elements).

Usage

The first byte or record in a file is always file position 1. After each read operation, the file position is advanced:

- For a binary file, by the size of the variable
- For a random file, by the size of a record

The Get statement reads data into *variableName* depending on the variable's data type. The following table shows how the Get statement behaves for different data types.

variableName data type	*Get statement's behavior*
Variant	The Get statement interprets the first two bytes as the DataType of the data to be read. If the DataType is EMPTY or NULL, the Get statement stops reading data and sets *variableName* to EMPTY or NULL. If the DataType is numeric, the Get statement reads the appropriate number of bytes used to store data of that Data Type: Byte: 1 byte Boolean: 2 bytes Integer: 2 bytes Long: 4 bytes Single: 4 bytes Double: 8 bytes Currency: 8 bytes Date/time: 8 bytes
Fixed-length string	The Get statement reads the specified number of characters. For example, if a variable is declared as String*10, the Get statement reads exactly 10 characters.

continued

variableName data type	*Get statement's behavior*
Variable-length string	The Get statement behaves differently, depending on the type of file you're using. Random file: The first two bytes read indicate the string's length. The Get statement reads exactly that number of characters. If *variableName* is larger than a random file record, data is read from the file until *variableName* is filled. After *variableName* is filled, the file position is advanced to the next record. Binary file: The number of bytes read from the file is equal to the length of the string currently assigned to *variableName*. If *variableName* has not been initialized, no data is read from the file.
A variable of a user-defined type	The number of bytes required to read the data is the sum of the number of bytes required to read all members of the used-defined data type, which cannot contain a dynamic array, a list, or an object.

Note Even though strings in LotusScript 4 can be longer than 64K, there are still restrictions with the length of the string you can read or write using the GET and PUT statements. The only combination of filetypes that will work with long strings is with a binary file and a variable-length string. Fixed length strings, strings in variants, and random files will not work with strings greater than 64K in length because they have a two-byte header which contains the length of the string. Two bytes cannot represent more than 64K.

Examples: Get statement

```
Type PersonRecord
   empNumber As Integer
   empName As String * 20
End Type

Dim fileNum% As Integer
Dim fileName$ As String
Dim rec As PersonRecord
fileNum% = FreeFile()
fileName$ = "data.txt"

' Open a random file with record length equal to the
' size of the records in rec.
Open fileName$ For Random As fileNum% Len = Len(rec)
```

```
' Write a record at position 1.
rec.empNumber% = 123
rec.empName$ = "John Smith"
Put #fileNum%, 1, rec

' Write a record at position 2.
rec.empNumber% = 456
rec.empName$ = "Jane Doe"
Put #fileNum%, 2, rec

' Write a record at position 3.
rec.empNumber% = 789
rec.empName$ = "Jack Jones"
Put #fileNum%, , rec
' Rewind to the beginning of the file and print all records.
Seek fileNum%, 1
Do While Not EOF(fileNum%)
   Get #fileNum%, , rec
   Print rec.empNumber%; rec.empName$
   ' Get function automatically advances to the next record.
Loop
Close fileNum%

' Prints three records:
'  123 John Smith
'  456 Jane Doe
'  789 Jack Jones
```

GetFileAttr function

Retrieves file-system attributes of a file or directory.

Syntax

GetFileAttr (*fileName* **)**

GetAttr is acceptable in place of GetFileAttr.

Elements

fileName

The name of a file or directory. File and directory names can optionally include paths.

Return value

The return value is the sum of the Integer values in the following list for those attributes that apply to *fileName*:

Value	*Attribute*	*Constant*
0	Normal file	ATTR_NORMAL
1	Read-only file	ATTR_READONLY
2	Hidden file	ATTR_HIDDEN
4	System file	ATTR_SYSTEM
16	Directory	ATTR_DIRECTORY
32	File that has changed since it was last backed up (archived)	ATTR_ARCHIVE

Usage

The constants in the preceding list are defined in the file lsconst.lss. Including this file in your script allows you to use constant names instead of their numeric values.

Examples: GetFileAttr function

This example creates a file, calls SetFileAttr to set its attributes to Read-Only, System, and Hidden, and then calls GetFileAttr to determine the file attributes.

```
%Include "lsconst.lss"

Dim fileNum As Integer, attr As Integer
Dim fileName As String, msg As String
fileNum% = FreeFile()
fileName$ = "data.txt"

Open fileName$ For Output As fileNum%
Close fileNum%
SetFileAttr fileName$, ATTR_READONLY + ATTR_SYSTEM + _
ATTR_HIDDEN
attr% = GetFileAttr(fileName$)
If (attr% And ATTR_READONLY) Then
  msg$ = msg$ & " Read-Only "
Else
  msg$ = msg$ & " Normal "
End If
If (attr% And ATTR_HIDDEN)    Then msg$ = msg$ & " Hidden "
If (attr% And ATTR_SYSTEM)    Then msg$ = msg$ & " System "
If (attr% And ATTR_DIRECTORY) Then msg$ = msg$ & " Directory "
Print msg$

SetFileAttr fileName$, ATTR_NORMAL  ' Reset to normal.
Kill fileName$
```

GetObject function

Opens an OLE Automation object contained in an application file, or returns the currently active OLE Automation object of the specified class.

Note GetObject is not supported under OS/2 or UNIX. It is supported on the Macintosh as long as OLE support is installed.

Syntax

GetObject ([*pathName*] [**,** *className*] **)**

Elements

pathName

Either a string containing the full path and file name of an application file or an empty string. The application must support OLE Automation. If *pathName* is the empty string ("") or is missing, you must specify a *className*.

className

A string of the form *appName.appClass* that identifies the application in which the class is defined and the class of the object to retrieve (for example, "WordPro.Application").

appName is the name of an application that supports OLE Automation. *appClass* is the name of the class of which you want to retrieve an instance.

Return value

GetObject returns an OLE Automation object reference.

Usage

Use the Set statement to assign the object reference returned by GetObject to a Variant variable.

If the application specified by *appName* is not already running, GetObject starts it before retrieving the OLE Automation object. References to the object remain valid only while the application is running. If the application terminates while you are using the object reference, LotusScript generates a run-time error.

If *pathName* is the empty string ("") or is missing, GetObject retrieves the currently active object of the specified class. If no object of that class is active, an error occurs.

If *className* is omitted, GetObject determines the application to run and the object to retrieve based on the *pathName*. This form of GetObject is useful only when the application file contains a single object.

Each product that supports OLE Automation provides one or more classes. See the product's documentation for details.

LotusScript supports the following return types for OLE properties and methods. Only an OLE method or property can return a type designated as "OLE only."

OLE return type	*Description*
VT_EMPTY	(No data)
VT_NULL	(No data)
VT_I2	2-byte signed integer
VT_I4	4-byte signed integer
VT_R4	4-byte real
VT_R8	8-byte real
VT_CY	Currency
VT_DATE	Date
VT_BSTR	String
VT_DISPATCH	IDispatch, OLE only
VT_ERROR	Error, OLE only
VT_BOOL	Boolean
VT_DECIMAL	Decimal, converted to Double
VT_VARIANT	(A reference to data of any other type)
VT_UNKNOWN	IUnknown, OLE only
VT_ARRAY	(An array of data of any other type)

Note If the application specified by appName is registered as a single-instance object, call CreateObject to get that object as GetObject will cause an error. This is different from Visual Basic; in Visual Basic, if GetObject is called with an empty string as first parameter, it behaves the same as CreateObject.

You can use a ForAll loop to iterate over the members of OLE collections.

LotusScript supports passing arguments to OLE properties. For example:

```
' Set v.prop to 4; v.prop takes two arguments.
v.prop(arg1, arg2) = 4
```

LotusScript does not support identifying arguments for OLE methods or properties by name rather than by the order in which they appear, nor does LotusScript support using an OLE name by itself (without an explicit property) to identify a default property.

Results are unspecified for arguments to OLE methods and properties of type boolean, byte, and date that are passed by reference. LotusScript does not support these data types.

The word GetObject is not a LotusScript keyword.

Examples: GetObject function

```
Dim myDoc As Variant

'Get the WordPro.Document object from a file.

Set mydoc =
getobject("d:\wordpro\docs\test.lwp","WordPro.Document")

' Call the Print method defined for WordPro.Document object.

myDoc.Print
```

The following script works on the Mac with Microsoft Word installed.

```
Sub Initialize

      Dim myDoc As Variant

      Dim filename As String

      filename = "MKashG4HD:MSTestDoc"

      'Get the Word.Document object from a file.

      Set myDoc = GetObject(filename,"Word.Document")

      ' Make the object visible

      myDoc.Application.Visible = True

End Sub
```

GetThreadInfo function

Returns system information about the thread.

Syntax

GetThreadInfo (Dim *InfoID* **as Integer)**

Elements

InfoID

Information to be returned

Return values

Data

A variant containing the information to be returned.

Usage

Pass any of the LSI_ constants from the table below to GetThreadInfo to have it return the current value of that constant.

Code	*Meaning*
LSI_THREAD_LINE	Current Line Number
LSI_THREAD_PROC	Name of current procedure
LSI_THREAD_MODULE	Name of current module
LSI_THREAD_VERSION	LotusScript version number
LSI_THREAD_LANGUAGE	(Human) language setting
LSI_THREAD_COUNTRY	Country or region setting
LSI_THREAD_TICKS	Get current clock ticks
LSI_THREAD_TICKS_PER_SEC	Get clock ticks per second (supported only on platforms that support parallel processing primitives)
LSI_THREAD_PROCESS_ID	Get current process ID (supported only on platforms that support parallel processing primitives)
LSI_THREAD_TASK_ID	Get current task ID (supported only on platforms that support parallel processing primitives)
LSI_THREAD_CALLPROC	Get the name of the calling procedure
LSI_THREAD_CALLMOD	Get the name of the calling module

The values of the constants are defined in LSPRVAL.LSS, which is automatically included through LSCONST.LSS.

Examples: GetThreadInfo function

```
%INCLUDE "lsconst.lss" 'include constants file

retval = GetThreadInfo(LSI_THREAD_VERSION)

print retval          'prints 5.0.0.4

retval = GetThreadInfo(LSI_THREAD_LANGUAGE)

print retval          'prints "en"

retval = GetThreadInfo(LSI_THREAD_COUNTRY)

print retval          'prints 1

retval = GetThreadInfo(LSI_THREAD_TICKS)

print retval          'prints 76135109
```

GoSub statement

Transfers control in a procedure to a labeled statement, with an optional return of control.

Syntax

GoS**ub** *label*

Elements

label

The label of a statement to which you want to transfer control.

Usage

You can't use the GoSub statement at the module level; you can only use it in a procedure. The GoSub statement, its label, and the Return statement must all reside in the same procedure.

When LotusScript encounters a GoSub statement, execution branches to the specified labeled statement and continues until either of two things happen:

- LotusScript encounters a Return statement, at which point execution continues from the statement immediately following the GoSub statement.
- LotusScript encounters a statement such as Exit or GoTo, which passes control to some other part of the script.

Examples: GoSub statement

```
' In response to user input, LotusScript transfers control
' to one of three labels, constructing an appropriate
' message, and continues execution at the statement
' following the GoSub statement.

Sub GetName
   Dim yourName As String, Message As String
   yourName$ = InputBox$("What is your name?")
   If yourName$ = "" Then    ' The user enters nothing.
      GoSub EmptyString
   ' Do a case-insensitive comparison.
   ElseIf LCase(yourName$) = "john doe" Then
      GoSub JohnDoe
   Else
      Message$ = "Thanks, " & yourName$ _
         & ", for letting us know who you are."
   End If
   ' The Return statements return control to the next line.
   MessageBox Message$
   Exit Sub
```

```
EmptyString:
   yourName$ = "John Doe"
   Message$ = "Okay! As far as we're concerned, " _
         & "your name is " & yourName$ & _
         ", and you're on the run!"
   Return

JohnDoe:
   Message$ = "We're on your trail, " & yourName$ _
         & ". We know you are wanted dead or alive!"
   Return
End Sub
GetName    ' Call the GetName sub.
```

GoTo statement

Transfers control within a procedure to a labeled statement.

Syntax

GoTo *label*

Elements

label

A label of a statement to which you want to transfer control.

Usage

You can't use the GoTo statement at the module level; you can only use it in a procedure. You can't use GoTo to transfer control into or out of a procedure or a With...End With block.

Use the GoTo statement to transfer control to any labeled statement that does not violate either of the preceding rules.

Examples: GoTo statement

This example illustrates On Error...GoTo, On...GoTo, Resume...GoTo, and GoTo.

The user enters a value. If the value is 1, 2, or 3, the On...GoTo statement transfers control to label1, label2, or label3. If the value is another number in range for On...GoTo (the range is 0-255), control moves on the next statement. If the user enters a number that is out of range for On...GoTo or that the CInt function cannot convert to an integer, an error condition occurs, and the OnError...GoTo statement transfers control to the OutOfRange label.

Depending on the user's entry, the OneTwoThree sub displays an appropriate message. If the entry is valid, an Exit Sub statement exits the Sub. If the entry is not valid, a GoTo statement transfers control to the EnterNum label, and the user is given another chance to make a valid entry.

```
Sub OneTwoThree
   Dim num As Integer
   On Error GoTo OutOfRange
EnterNum:
   num% = CInt(InputBox("Enter 1, 2, or 3"))
   On num% GoTo label1, label2, label3
  ' The user did not enter 1, 2, or 3, but a run-time error
  ' did not occur (the user entered a number in the
  ' range 0-255).
   MessageBox "You did not enter a correct value! Try again!"
   GoTo EnterNum
label1:
   MessageBox "You entered 1."
   Exit Sub
label2:
   MessageBox "You entered 2."
   Exit Sub
label3:
   MessageBox "You entered 3."
   Exit Sub
   ' An error condition has occurred.
OutOfRange:
   MessageBox "The value you entered is negative, " _
      & "greater than 255, or is not a number. Try again!"
   GoTo EnterNum
End Sub
OneTwoThree   ' Call the OneTwoThree sub.
```

Hex function

Return the hexadecimal representation of a number as a string.

Syntax

Hex[**$**] **(** *numExpr* **)**

Elements

numExpr

Any numeric expression. If *numExpr* evaluates to a number with a fractional part, LotusScript rounds it to the nearest integer before deriving its hexadecimal representation.

Return value

Hex returns a Variant of DataType 8 (String), and Hex$ returns a String.

Return values will only include the characters 0 - 9 and A - F, inclusive. The maximum length of the return value is eight characters.

Usage

If the data type of *numExpr* is not Integer or Long, LotusScript attempts to convert it to a Long. If it cannot be converted, an error occurs.

Examples: Hex function

```
Print Hex$(15)             ' Prints "F"

' Converts Double argument to Long.
Print Hex$(15.0)           ' Prints "F"

' Rounds Double argument, then converts to Long.
Print Hex$(15.3)           ' Prints "F"

' Computes product 14.841, rounds to 15.0, then converts to
15.
Print Hex$(15.3 * .97)     ' Prints "F"
```

Hour function

Returns the hour of the day for a date/time argument as an integer from 0 to 23.

Syntax

Hour (*dateExpr* **)**

Elements

dateExpr

Any of the following:

- A valid date/time string of String or Variant data type.

  For Notes or Domino, LotusScript interprets a 2-digit year designation in a date/time string so that 50 through 99 represent the years 1950 through 1999 and 00 through 49 represent the years 2000 through 2049.

  For SmartSuite, LotusScript interprets the years differently. For more information, see the SmartSuite online Help entry entitled Year 2000.

- A number within the valid date range: the range -657434 (Jan 1, 100 AD) to 2958465 (Dec 31, 9999 AD), inclusive.
- NULL

Return value

Hour returns a Variant containing a value of DataType 2 (Integer). If the *dateExpr* is a Variant containing the value NULL, then Hour returns NULL.

Language cross-reference

@Hour function in formula language

Examples: Hour function

```
' Construct a message that displays the current time and
' the number of hours, minutes, and seconds remaining
' in the day.
Dim timeFrag As String, hoursFrag As String
Dim minutesFrag As String, secondsFrag As String
Dim crlf As String, message As String
timeFrag$ = Format(Time, "h:mm:ss AM/PM")
hoursFrag$ = Str(23 - Hour(Time))
minutesFrag$ = Str(59 - Minute(Time))
secondsFrag$ = Str(60 - Second(Time))
crlf$ = Chr(13) & Chr(10)      ' Carriage return/line feed
message$ = "Current time: " & timeFrag$ & ". " & crlf$ _
   & "Time remaining in the day: " _
   & hoursFrag$ & " hours, " _
   & minutesFrag$ & " minutes, and " _
   & secondsFrag$ & " seconds."
MessageBox(message$)
```

If...GoTo statement

Conditionally executes one or more statements or transfers control to a labeled statement, depending on the value of an expression.

Syntax

If *condition* **GoTo** *label* [**Else** [*statements*]]

Elements

condition

Any numeric expression. A value of 0 is interpreted as FALSE, and any other value is interpreted as TRUE.

label

The name of a label.

statements

A series of statements, separated by colons.

Usage

An If...GoTo statement must occupy a single line of code. Line continuation with the underscore character (_) is allowed.

If *condition* is TRUE, LotusScript executes the GoTo statement, transferring control to the statement following the label *label*. If *condition* is FALSE, LotusScript executes the block of *statements* in the Else clause. If there is no Else clause, execution continues with the next statement.

You can't use an If...GoTo statement to transfer control into or out of a procedure, and you can't use it at the module level.

Language cross-reference

@If function in formula language

@V2If function in formula language

@Failure function in formula language

Examples: If...GoTo statement

Ask the user to propose a down payment for a house. Elsewhere, the cost has been set at $235,000. Depending on whether or not the user proposes a down payment of at least 10% of cost, respond accordingly.

```
Sub ProcessMortgage(cost As Single)
   Dim downpmt As Single, msg As String
   msg$ = "Cost: " + Format(cost!, "Currency") _
         & ". Enter a down payment:"
   downpmt! = CSng(InputBox(msg$))
   If downpmt! < .1 * cost! GoTo NotEnough
   msg$ = Format(downpmt!, "Currency") & " will do fine!"
   MessageBox msg$
   ' Continue processing the application
   ' ...
   ' ...
   Exit Sub

   NotEnough:
   msg$ = "Sorry, " & Format(downpmt!, "Currency") _
       & " is not enough!"
   MessageBox msg$
End Sub

Dim cost As Single
cost! = 235000
ProcessMortgage(cost!)    ' Call the ProcessMortgage sub.
```

If...Then...Else statement

Conditionally executes one or more statements, depending on the value of an expression.

Syntax

If *condition* **Then** [*statements*] [**Else** [*statements*]]

Elements

condition

Any numeric expression. A value of 0 is interpreted as FALSE, and any other value is interpreted as TRUE.

statements

A series of statements, separated by colons.

Usage

An If...Then...Else statement must occupy a single line of code. Line continuation with the underscore character (_) is allowed.

If *condition* is TRUE, the statements following Then, if any, are executed. If *condition* is FALSE, the statements following Else are executed.

If no statements follow Then, and there is no Else clause, Then must be followed by a colon (:). Otherwise LotusScript assumes that the statement is the first line of an If...Then...Else...End If statement.

Language cross-reference

@If function in formula language

@V2If function in formula language

Examples: If...Then...Else statement

```
Dim x As Integer
If x% > 0 Then Print FALSE Else Print TRUE
' Output:
' True
```

The initial value of x is 0, so LotusScript prints True.

If...Then...ElseIf statement

Conditionally executes a block of statements, depending on the value of one or more expressions.

Syntax

If *condition* **Then**

statements

[**ElseIf** *condition* **Then**

statements]

...

[**Else**

statements]

End If

Elements

condition

Any numeric expression. A value of 0 is interpreted as FALSE, and any other value is interpreted as TRUE.

statements

Statements that are executed if *condition* is TRUE.

Usage

LotusScript executes the statements following the Then keyword for the first condition whose value is TRUE. It evaluates an ElseIf condition if the preceding condition is FALSE. If none of the conditions is TRUE, LotusScript executes the statements following the Else keyword. Execution continues with the first statement following the End If statement.

You can include any number of ElseIf expressions in the block.

You can include an If statement within an If statement. Each If block must be terminated by an End If.

Language cross-reference

@If function in formula language

@V2If function in formula language

Examples: If...Then...ElseIf statement

```
Dim quantity As Integer, pctDiscount As Single
Dim unitPrice As Currency, total As Currency
unitPrice@ = 3.69
quantity% = 50

' Define discount based on quantity purchased.
If quantity% > 99 Then
   pctDiscount! = .20
ElseIf quantity% > 49 Then
   pctDiscount! = .10
Else
   pctDiscount! = 0
End If
total = (quantity% * unitPrice@) * (1 - pctDiscount!)
Print   "Unit price: $"; unitPrice@, _
  "Quantity: "; quantity%, _
  "Discount%: "; pctDiscount!, _
  "Total: $"; total@
```

%If directive

Conditionally compiles a block of statements, depending on the value of one or more product constants.

Syntax

%If *productConst*

statements

[**%ElseIf** *productConst*

statements]

...

[**%Else**

statements]

%End If

Elements

productConst

A constant defined by a Lotus software application, or one of the platform-identification constants described below. Refer to the product's documentation for a list of product-defined constants.

statements

Statements that are compiled if *productConst* evaluates to TRUE.

Usage

You cannot enter %If, %ElseIf, %Else, and %End If directly in the IDE. You must enter these directives in a file and insert the file in the IDE with the %Include directive.

productConst must appear on the same line as %If or %ElseIf. Nothing except a comment can appear on the same line following %If *productConst* or %ElseIf *productConst*, or on the same line with %Else or %End If. None of these lines can be continued with the underscore character (_).

To test each %If condition or %ElseIf condition in this statement, the LotusScript compiler calls the Lotus software application to evaluate the constant *productConst*. The product returns either TRUE (-1) or FALSE (0).

A condition is evaluated only if the product returns FALSE for the preceding condition. LotusScript compiles the *statements* for the first %If condition or %ElseIf condition that the product evaluates as TRUE. Once this happens, no further conditions are evaluated, and no further *statements* are compiled.

If neither the %If condition nor any %ElseIf condition evaluates to TRUE, the %Else *statements* (if any) are compiled.

You can include any number of %ElseIf directives in the block.

You can't include an %If block within an %If block.

LotusScript implements the constants in the following table as product #defines. When one of these is used as *productConst*, the LotusScript compiler does not call the product to evaluate *productConst*. LotusScript itself evaluates the constant as TRUE or FALSE. The value of each constant depends on the platform LotusScript is running on.

Constant	*Platform or functionality*
WIN16	Windows with 16-bit API (Windows 3.1)
WIN32	Windows with 32-bit API (Windows NT or Windows® 95)
WINNT	Windows NT
WIN95	Windows 95
WIN40	Windows 95 or Windows NT 4.0
WIN98	Windows 98
WIN2K	Windows 2000
WINDOWS	Any Windows platform type (any of the above WIN*xx* constants)
HPUX	HP®/UNIX 9.X or greater
SOLARIS	Sun™/OS 5.0 or greater

continued

Constant	*Platform or functionality*
UNIX	Any UNIX type (HP_UX® or Solaris®)
OS2	OS/2, version 2.0 or greater
LINUX	Linux
MAC	Macintosh System 7
OLE	OLE-2 is available
MAC68K	Macintosh Motorola® 68000 version (running on either a 68*xxx* Macintosh or the PowerPC™)
MACPPC	Macintosh PowerPC version
OS400	IBM AS400
OS390	IBM S/390 User System Services
ZOS	IBM z/OS (OS390 is also TRUE)

For example, here are several platforms and the constants that identify them:

Windows 3.1
    WIN16, WINDOWS

Windows 95
    WIN32, WIN95, WIN40, WINDOWS

HP/UNIX 9.X
    HPUX, UNIX

OS/2 2.0 or greater
    OS2

Examples: %If directive

This example compiles and runs in either Windows 3.1, Windows NT, or Windows 95. Depending on whether the application is compiled and run under 16-bit Windows (Windows 3.1) or 32-bit Windows (Windows 95 or Windows NT), you should declare and use an appropriate Windows handle variable and the appropriate version of two Windows API functions.

GetActiveWindow returns the handle (an Integer in 16-bit Windows, a Long in 32-bit Windows) of the currently active window. GetWindowText returns the text in the window title bar.

```
Dim winTitle As String * 80
%If WIN16                       ' 16-bit Windows
   Dim activeWin As Integer     ' Window handles are Integer.
   Declare Function GetActiveWindow% Lib "User" ()
   Declare Function GetWindowText% Lib "User" _
         (ByVal hWnd%, ByVal lpstr$, ByVal i%)
%ElseIf WIN32                   ' 32-bit Windows
```

```
   Dim activeWin As Long       ' Window handles are Long.
   Declare Function GetActiveWindow& Lib "User32" ()
   Declare Function GetWindowText% Lib "User32" _
          Alias "GetWindowTextA" _
      (ByVal hWnd&, ByVal lpstr$, ByVal i&)
%End If

' Print the name of the currently active window.
activeWin = GetActiveWindow()  ' Returns an Integer or a Long.
Call GetWindowText(ActiveWin, winTitle$, 80)
Print winTitle$
```

IMESetMode function

Changes the current input mode (IME) into the mode user specified at its parameter. IMESetMode is supported for Windows DBCS system only.

Syntax

IMESetMode (*IMEMode* **)**

Elements

IMEMode

Integer value for the desired IME mode user prefer to set. You can specify the values listed in the following table for the IMEMode parameter.

Country or Region	*Constant*	*Value*	*Description*
All	IME_ON	1	Set IME on
	IME_OFF	2	Set IME off
Japan	IME_HIRAGANA	4	Double-byte Hiragana
	IME_KATAKANA_DBCS	5	Double-byte Katakana
	IME_KATAKANA_SBCS	6	Single-byte Katakana
	IME_ALPHA_DBCS	7	Double-byte alphanumeric
	IME_ALPHA_SBCS	8	Single-byte alphanumeric
Taiwan	IME_NATIVE_MODE	4	Taiwan native mode
	IME_ALPHA_DBCS	7	Double-byte alphanumeric
	IME_ALPHA_SBCS	8	Single-byte alphanumeric
Korea	IME_HANGEUL	4	Hangeul DBC
	IME_HANJACONVERT	5	Hanja conversion
	IME_ALPHA_DBCS	7	Double-byte alphanumeric

continued

Country or Region	*Constant*	*Value*	*Description*
	IME_ALPHA_SBCS	8	Single-byte alphanumeric
PRC	IME_NATIVE_MODE	4	PRC native mode
	IME_ALPHA_DBCS	7	Double-byte alphanumeric
	IME_ALPHA_SBCS	8	Single-byte alphanumeric

Return values

TRUE

IME mode has been set successfully.

FALSE

Unable to set IME correctly, or unable to find IME on the system.

Usage

IMESetMode is available on interactive execution of LotusScript.

The IMESetMode function is related with the IMEStatus function and generally used with it.

The IMESetMode function is expected to be used upon the Entering event of a Notes field.

Examples: IMESetMode

In this example when the user moves the cursor into a field, IME is automatically invoked into HIRAGANA input mode. When the user moves from the field, IME resets to its original status.

```
Public InitIMEMode As Integer

Sub Entering ( Source As Field )
    InitIMEMode = IMEStatus
    If InitIMEMode <> IME_HIRAGANA Then
        Call IMESetMode ( IME_HIRAGANA )
    End If
End Sub

Sub Exiting ( Source As Field )
    If InitIMEMode <> IMEStatus Then
        Call IMESetMode ( InitIMEMode )
    End If
End Sub
```

IMEStatus function

Returns an integer indicating the current input mode (IME) for extended character sets.

Note that IMEStatus is supported for Windows DBCS only. The codes for PRC and the Taiwan region are supported on Win95 only.

Syntax

IMEStatus

Return value

The function returns a status code indicating the current input mode (IME).

Usage

IMEStatus provides support for languages that use extended character sets. The code returned depends on the country for which the Lotus software application is built. The following table describes the return values. For countries not listed in the table, the return value is 0.

Country or Region	*Constant*	*Value*	*Description*
All	IME_NOT_INSTALLED	0	IME is not installed
	IME_ON	1	IME is on
	IME_OFF	2	IME is off
Japan	IME_HIRAGANA	4	Double-byte Hiragana
	IME_KATAKANA_DBCS	5	Double-byte Katakana
	IME_KATAKANA_SBCS	6	Single-byte Katakana
	IME_ALPHA_DBCS	7	Double-byte alphanumeric
	IME_ALPHA_SBCS	8	Single-byte alphanumeric
Taiwan	IME_NATIVE_MODE	4	Taiwan native mode
	IME_ALPHA_DBCS	7	Double-byte alphanumeric
	IME_ALPHA_SBCS	8	Single-byte alphanumeric
Korea	IME_HANGEUL	4	Hangeul DBC
	IME_HANJACONVERT	5	Hanja conversion
	IME_ALPLHA_DBCS	7	Double-byte alphanumeric
	IME_ALPHA_SBCS	8	Single-byte alphanumeric
PRC	IME_NATIVE_MODE	4	PRC native mode
	IME_ALPHA_DBCS	7	Double-byte alphanumeric
	IME_ALPHA_SBCS	8	Single-byte alphanumeric

Example
See IMESetMode.

Implode function

Concatenates all members of an Array of Strings and returns a string. Elements of the Array are separated by a delimiter, if provided, or the space character (" ").

Syntax
Implode(*sourceArray* **as Variant, [***delimiter* **as String]) as String**

Elements
sourceArray

One-dimensional Array containing the substrings to be concatenated. sourceArray can be an array of Strings, or an array of Variants. If sourceArray is an array of Variants, Implode will attempt to convert any non-string elements to Strings.

delimiter

Optional String containing separation character(s) for the concatenated Strings

Return value
Implode returns a String containing the elements of sourceArray with delimiter between elements, or with the space character " " as a separator if delimiter is not specified.

Usage
Implode creates a String that will hold the concatenation of sourceArray. Implode then iterates through sourceArray, With each iteration, Implode converts the next element of sourceArray to a String, if necessary, and appends it to the concatenation String. If more elements remain in sourceArray, a delimeter (either " " or the specified value) is appended to the concatenation String and Implode continues to iterate. After all elements of sourceArray have been concatenated, Implode returns the concatenation String.

Error handling
Implode will throw a Run-time Type mismatch if:

- An element in a variant array cannot be coerced to a string.
- The delimiter is set to nothing.
- The array passed in is not of either type string or variant.

- A list is passed instead of an array.
- The array passed in contains an element set to nothing.
- The array passed in has not been properly initialized.

Implode will throw a run-time Wrong Number of Dimensions error if the array is not one-dimensional.

Implode will throw a run-time Invalid Use of Null error if the array passed in contains an element set to null or if the delimiter is set to null.

Note Implode is an alias of Join and is identical in every way.

Examples: Implode function

```
Dim A(2) As String

A(0) = "one"
A(1) = "two"
A(2) = "three"

Sub Initialize

   Dim ret As String
   Dim delim As String'this is the delimiter
   delim = "-"
   ret = implode(A, delim)
   Print ret
End Sub

'Output:

'one-two-three
```

%Include directive

At compile time, inserts the contents of a text file into the module where the directive appears.

Syntax

%Include *fileName*

Elements

fileName

A string literal whose value is a file name; you can optionally include a path.

If you omit the file name extension, LotusScript assumes .lss. To include a file that has no extension, include a period at the end of the file name. For example:

```
%Include "orfile."
```

This prevents LotusScript from adding the .lss extension to the file name.

Usage

The %Include directive must be the only item on a line, except for an optional trailing comment. It must be followed by white space (a space character, a tab character, or a newline character).

If you don't specify a path for the included file, the search path depends on the specific Lotus software application you're using. For example, if you are using Lotus Notes, the default directory is the Notes program directory. This is true also if only a partial directory is given. The following table demonstrates the possible search path options depending on what it is entered for *fileName*.

%Include statement	*Path type*	*LotusScript looks for this file:*
%Include "myfile.lss"	relative path, no directory	<program dir>\myfile.lss
%Include "include\myfile.lss"	relative path, with directory	<program dir>\include\myfile.lss
%Include "c:\include\myfile.lss"	absolute path	c:\include\myfile.lss

An included file can itself contain %Include directives. You can nest up to 16 files.

At compile time, LotusScript replaces the %Include directive with the entire contents of the named file. They are then compiled as part of the current script.

If a run-time error occurs in a statement in an included file, the line number reported is that of the %Include directive.

If a compile-time error occurs in a statement in an included file, the file name and the line number within that included file are reported with the error.

The file you include must be a text file containing only LotusScript statements. If anything in the included file cannot be compiled, LotusScript generates a compiler error.

If the file is not found, LotusScript generates an error.

Note EBCDIC platforms may exhibit backwards incompatibility starting with LotusScript Release 5 (Notes/Domino Release 6). Earlier releases interpret an included file as LMBCS (which is the same as ASCII in the single-byte range). Ongoing releases interpret an included file using the platform-native character set. On EBCDIC platforms, included text must be EBCDIC. In particular, if you have shipped ASCII-encoded LotusScript source files without text translation (binary FTP, for example), the files must be translated on EBCDIC platforms prior to inclusion.

Examples: %Include directive

```
' Include the contents of c:\testfile.dat with
' the current script when it is compiled.
%Include "c:\testfile.dat"
```

Input # statement

Reads data from a sequential file and assigns that data to variables.

Syntax

Input #*fileNumber* **,** *variableList*

Elements

fileNumber

The number assigned to the file when you opened it. A pound sign (#) sign must precede the file number.

variableList

A list of variables, separated by commas. The data read from the file is assigned to these variables. File data and its data types must match these variables and their data types.

variableList cannot include arrays, lists, variables of a user-defined data type, or object reference variables. It can include individual array elements, list elements, and members of a user-defined data type or user-defined class.

Usage

The following table shows how the Input # statement reads characters for various data types.

variableList data type	*How Input # reads characters*
Numeric variable	The next non-space character in the file is assumed to begin a number. The next space, comma, or end-of-line character in the file ends the number. Blank lines and non-numeric values are translated to the number 0.
String variable	The next non-space character in the file is assumed to begin a string. Note these special conditions:
	If that character is a double quotation mark ("), it is ignored; however, all characters following it (including commas, spaces, and newline characters) up to the next double quotation mark are read into the string variable.
	If the first character is not a double quotation mark, the next space, comma, or end-of-line character ends the string.
	Blank lines are translated to the empty string (“”).
	Note that tab is a non-space character.
Fixed-length string variable	LotusScript reads this according to its length. For example, LotusScript reads a variable declared as String *10 as 10 bytes.
Variant variable	The next non-space character in the file is assumed to begin the data.
	If the data is:
	Empty (a delimiting comma or blank line), LotusScript assigns the variable the EMPTY value.
	The literal “#NULL#”, LotusScript assigns the variable the NULL value.
	A date/time literal, LotusScript assigns the variable the DataType 7 (Date/Time).
	A whole number, LotusScript assigns the variable the Data Type 2 (integer) if the number is in the legal range for integer; the DataType 3 (Long) if the number is in the legal range for Long but not within the range for integer; and otherwise the DataType 5 (Double).
	A number with a fractional part, LotusScript assigns the variable the DataType 5 (Double).
	If none of the above applies, LotusScript assigns the variable the String type.

If LotusScript encounters an EOF (end-of-file), input terminates and an error is generated.

LotusScript inserts "chr(10)" to represent the newline character in any multi-line string (for example, a string that you type in using vertical bars or braces). If you Print the string to a file, this newline character will be translated into the platform-specific newline character(s). If you Write the string to a file, no translation is done.

Note Newline does not mean either chr(10) or chr(13) on all platforms. Newline is the character or sequence of characters that is used to mark the end of a line. This may be chr(10), or chr(13), but it may also be something else, because the actual value of newline depends on the platform.

Note When reading a multiline string from a sequential file written by the Write # statement, use Input, not Line Input.

When reading record-oriented data, using a random file with the Get statement is easier and more efficient than using Input #. Use Get for reading record-oriented data (a random file); use Input # for reading text data (a sequential file).

Examples: Input # statement

```
Dim fileNum As Integer

Dim fname As String

Dim customer As String, addr As String, city As String

fname = "data.dat"

   fileNum% = Freefile()

   Open fname for Output As fileNum%

   Write #fileNum%, {John Roe

   and family}, "25 Main Street", "Springfield, IL"

   Write #fileNum%, "Mary Johnson", {Fifth Floor

55 Cambridge Parkway}, "Cambridge, MA"

   Close


   fileNum% = FreeFile()

   Open fname For Input As fileNum%

   For i% = 1 to 2

          Input #fileNum%, customer, addr, city

          Print customer
```

```
            Print addr
            Print city
            Print ""
      Next i%

'  Output:
      '  Outputs two groups, each consisting of three String
values,
      '  with some Strings on multiple lines:
      '  John Roe        ' customer, line 1
      '       and family ' customer, line 2
      '  25 Main Street         ' addr
      '  Springfield, IL ' city

      '  Mary Johnson           ' customer
      '  Fifth Floor            ' addr, line 1
      '  55 Cambridge Parkway   ' addr, line 2
      '  Cambridge, MA          ' city
Close fileNum%
```

Input function

Reads a sequence of characters from a sequential or binary file into a string variable, without interpreting the input.

Syntax

Input[**$**] **(** *count* **,** [**#**]*fileNumber* **)**

Elements

count

The number of characters to read.

fileNumber

The number assigned to the file when you opened it.

Return value

The Input function returns a Variant, and Input$ returns a String.

LotusScript returns the specified number of characters, beginning at the current position in the file.

If you request more characters than are available, LotusScript generates an error.

If *count* is 0, LotusScript returns the empty string ("").

Usage

The input data is not filtered or translated in any way. All characters are returned, including newline characters, quotation marks, and spaces.

If you want to work with bytes instead of characters, use the InputB or InputB$ function.

You cannot use the Input, Input$, InputB, or InputB$ functions to read a file opened in Output, Append, or Random mode.

Examples: Input function

```
Dim fileNum As Integer
Dim fileName As String
Dim firstCheck As String

fileNum% = FreeFile()
fileName$ = "data.dat"

' Write out some employee data.
Open fileName$ For Output As fileNum%
Write #fileNum%, "Joe Smith", 123, "1 Smith Road", 25000.99
Write #fileNum%, "Jane Doe", 456, "Two Cambridge Center", _
      98525.66
Close fileNum%

' Read in first 23 characters of data and print.
Open fileName$ For Input As fileNum%
firstCheck$ = Input$(23, fileNum%)
Print firstCheck$   ' Output:  "Joe Smith",123,"1 Smit
Close fileNum%
```

InputB function

Reads a sequence of bytes from a sequential or binary file into a string variable without interpreting the input.

Syntax

InputB[**$**] **(** *count* **,** [**#**]*fileNumber* **)**

count
The number of bytes to read.

fileNumber
The number assigned to the file when it was opened.

Return value

The InputB function returns a Variant, and InputB$ returns a String.

LotusScript returns the specified number of bytes, beginning at the current position within the file. If you request more bytes than are available, LotusScript generates an error.

The length of the returned string (measured in characters, as computed by the Len function) is (# *bytes returned*) / 2 if an even number of bytes is returned, and otherwise (# *bytes returned* + 1) / 2, if an odd number of bytes is returned. If an odd number of bytes is returned, then the last character in the returned string is padded with a 0 byte.

If *count* is 0, LotusScript returns the empty string ("").

Usage

The input data is not filtered or translated in any way. All bytes are returned, including the bytes representing newline, quotation marks, and space.

If you want to work with characters instead of bytes, use the Input or Input$ function.

You cannot use the Input, Input$, InputB, or InputB$ function to read a file opened in Output, Append, or Random mode.

Examples: InputB function

```
Print InputB$(4, 1)
' Prints the next four bytes from file number 1.
```

InputBox function

Displays a dialog box containing a prompt for user entry, and returns input from the user as a string.

Syntax

InputBox[**$**] **(** *prompt* [**,** [*title*] [**,** [*default*] [**,** *xpos* **,** *ypos*]]] **)**

Elements

prompt

A string expression. This is the message displayed in the dialog box. *prompt* can be any length. LotusScript defines, but does not enforce, a minimum supported length of 128. The specific product being used (that is, Notes, ESB, and so on) may impose other limits.

title

Optional. A string expression. This is displayed in the title bar of the dialog box. *title* can be any length. LotusScript defines, but does not enforce, a minimum supported length of 128. The specific product being used (that is, Notes, ESB, and so on) may impose other limits.

If you omit *title*, nothing is displayed in the title bar. If you omit *title* and specify either *default* or *xpos* and *ypos*, include a comma in place of *title*.

default

Optional. A string expression. This is displayed in the text entry field in the dialog box as the default user response. *default* can be any length. LotusScript defines, but does not enforce, a minimum supported length of 512. The specific product being used (that is, Notes, ESB, and so on) may impose other limits.

If you omit *default*, the text input box is empty. If you omit *default* and specify *xpos* and *ypos*, include a comma in place of *default*.

xpos

Optional. A numeric expression that specifies the horizontal distance, in units of 1 pixel, between the left edge of the dialog box and the left edge of the display screen. If you omit *xpos*, the distance is 0. If you specify *xpos*, you have to specify *ypos* as well.

ypos

Optional. A numeric expression that specifies the vertical distance, in units of 1 pixel, between the top edge of the dialog box and the top edge of the screen. If you omit *ypos*, the distance is 0. If you specify *ypos*, you have to specify *xpos* as well.

Return value

The InputBox function returns a Variant containing a string. InputBox$ returns a String. Returned string can be any length. LotusScript defines, but does not enforce, a minimum supported length of 512. The specific product being used (that is, Notes, ESB, and so on) may impose other limits.

Usage

InputBox displays a dialog box with OK and Cancel buttons and a text entry field, interrupting execution of the script until the user confirms the text entry by clicking OK or Cancel. Then InputBox returns that entry. If the user clicks Cancel, InputBox returns the empty string (""). When the user clicks OK or Cancel, execution resumes.

The Lotus software where you are running LotusScript may allow longer strings than described above for *prompt*, *title*, *default*, and the text entered into the text entry field. LotusScript will support longer strings for these items if the Lotus software does, up to the maximum string size.

If you are using LotusScript from within Lotus Notes, note that the InputBox function writes to:

- A dialog box when executing on a Notes client. The user clicks OK, Cancel, Abort, Retry, Yes, or No to continue.
- NOTES.LOG when executing on a Domino server.

Language cross-reference

@Prompt function in formula language

Examples: InputBox function

```
' Ask the user for an integer. Convert user input
' from a string to an integer.

Dim num As Integer
num% = CInt(InputBox$("How many do you want?"))
```

InputBP function

Reads a sequence of bytes (in the platform-native character set) from a sequential or binary file into a string variable without interpreting the input.

Syntax

InputBP[**$**] **(** *count* **,** [**#**]*fileNumber* **)**

count
The number of bytes to read.

fileNumber
The number assigned to the file when it was opened.

Return value

The InputBP function returns a Variant, and InputBP$ returns a String.

LotusScript returns the specified number of bytes, beginning at the current position within the file. If you request more bytes than are available, LotusScript generates an error.

The length of the returned string (measured in characters, as computed by the Len function) is the number of Unicode characters that the bytes translate into. For example, 10 bytes of ASCII characters translate into 10 Unicode characters; 10 bytes of DBCS characters translate into 5 Unicode characters. If the last requested byte read is the lead byte of a DBCS character, the byte is dropped and the file pointer is positioned one byte before the last requested byte.

If *count* is 0, LotusScript returns the empty string (“”).

Usage

The input data is translated into Unicode.

If you want to work with characters instead of platform bytes, use the Input or Input$ function. If you want to work with untranslated bytes, use the InputB or InputB$ function.

You cannot use the Input, Input$, InputB, InputB$, InputBP, or InputBP$ function to read a file opened in Output, Append, or Random mode.

Examples: InputBP function

```
Print InputBP(4, 1)
' Prints the next four bytes from file number 1.
```

InStr function

Returns the position of the character that begins the first occurrence of one string within another string.

Syntax

InStr ([begin ,] string1 , string2)

or

InStr ([*begin* **,**] *string1* **,** *string2* [**,** *compMethod*] **)**

Elements

begin

Optional. A numeric expression with a positive integer value. *begin* specifies the character position in *string1* where InStr should begin searching for *string2*. If you omit *begin*, it defaults to 1. If you specify *compMethod*, you must specify *begin* as well.

string1

The string that InStr searches for the occurrence of *string2*.

string2

The string for which InStr searches to see if it occurs in *string1*.

compMethod

A number designating the comparison method:

Number	*Comparison method*
0	case-sensitive, pitch-sensitive
1	case-insensitive, pitch-sensitive
4	case-sensitive, pitch-insensitive
5	case-insensitive, pitch-insensitive

If you specify *compMethod*, you must specify *begin* as well.

If you omit *compMethod*, the default comparison mode is the mode set by the Option Compare statement for this module. If there is no statement for the module, the default is case-sensitive and pitch-sensitive.

Return value

InStr returns the character position of the first occurrence of *string2* within *string1*. The following table shows how the function responds to various conditions.

Condition	*Return value*
string1 is the empty string ("")	0
string2 is not found after *begin* in *string1*	0
begin is larger than the length of *string1*	0
string2 is the empty string ("")	The value of *begin*. If you omit *begin*, InStr returns the value 1.
string1 is NULL	NULL
string2 is NULL	NULL
begin or *compMethod* is NULL	Error

Usage

If you want to work with bytes, use the InStrB function.

Language cross-reference

@Middle function in formula language

Examples: InStr function

```
' The value 5 (the position of the character where the first
' occurrence of LittleString begins in BigString) is assigned
' to the variable positionOfChar.

Const CaseAndPitch = 0

Const PitchNoCase = 1

Const CaseNoPitch = 4

Const NoCaseNoPitch = 5

' The value 5 (the position of the character where the first

' occurrence of LittleString begins in BigString) is assigned

' to the variable positionOfChar.

Dim BigString As String, LittleString As String

Dim positionOfChar As Long

BigString$ = "abcdefghi"

LittleString$ = "efg"

positionOfChar& = InStr(BigString$, LittleString$)

Print positionOfChar& ' Output:  5
```

```
positionOfChar& = InStr(1, BigString$, LittleString$)

Print positionOfChar& ' Output:  5

positionOfChar& = InStr(1, BigString$, LittleString$, _
NoCaseNoPitch)

Print positionOfChar& ' Output:  5
```

InStrB function

Returns the position of the byte beginning the first occurrence of one string within another string.

Syntax

InStrB ([*begin* **,**] *string1* **,** *string2* **)**

Elements

begin

Optional. A numeric expression with a positive integer value, *begin* specifies the character position in *string1* where InstrB should begin searching for *string2*. If you omit *begin*, it defaults to 1.

string1

The string to be searched.

string2

The string for which InStrB searches.

Return value

InStrB returns the byte position of the first occurrence of *string2* in *string1*. The following table shows how the function responds to various conditions.

Condition	*Return value*
string1 is "" (the empty string)	0
string2 is not found after *begin* in *string1*	0
begin is larger than the length of *string1*	0
string2 is "" (the empty string)	The value of *begin*. (If you omit *begin*, InStrB returns the value 1.)
string1 is NULL	NULL
string2 is NULL	NULL
begin is NULL	Error

Usage

If you want to work with characters, use the InStr function.

Note The byte position returned by InStrB is independent of the platform-specific byte order.

Examples: InStrB function

```
' The value 5 (the position of the byte where the first
' occurrence of littleStr begins in bigStr) is assigned to
' the variable positionOfByte.

Dim bigStr As String, littleStr As String
Dim positionOfByte As Long
bigStr$ = "abcdefghi"
littleStr$ = "efg"
positionOfByte& = InStrB(1, bigStr$, littleStr$)
Print positionOfByte& ' Output:  5
```

InStrBP function

Returns the position of the byte (in the platform-native character set) beginning the first occurrence of one string within another string.

Syntax

InStrBP ([*begin* **,**] *string1* **,** *string2* **)**

Elements

begin

Optional. A numeric expression with a positive integer value, *begin* specifies the character position in *string1* where InStrBP should begin searching for *string2*. If you omit *begin*, it defaults to 1.

string1

The string to be searched.

string2

The string for which InStrBP searches.

Return value

InStrBP returns the byte position in the platform-specific character set of the first occurrence of *string2* in *string1*. The following table shows how the function responds to various conditions.

Condition	*Return value*
string1 is "" (the empty string)	0
string2 is not found after *begin* in *string1*	0
begin is larger than the length of *string1*	0
string2 is "" (the empty string)	The value of *begin*. (If you omit *begin*, InStrB returns the value 1.)
string1 is NULL	NULL
string2 is NULL	NULL
begin is NULL	Error

Usage

If you want to work with characters, use the InStr function.

Examples: InStrBP function

```
' The value 5 or other value depending on platform
' (the position of the byte where the first
' occurrence of littleStr begins in bigStr) is assigned to
' the variable positionOfByte.

Dim bigStr As String, littleStr As String
Dim positionOfByte As Long
bigStr$ = "abcdefghi"
littleStr$ = "efg"
positionOfByte& = InStrBP(1, bigStr$, littleStr$)
Print positionOfByte& ' Output:  5
```

InStrC function

Returns the position of the column that begins the first occurrence of one string within another string for column-based writing systems, such as Thai.

Syntax

InStrc(*off*, *string1*, *string2*)

Elements

off
The number of the offset

string1
A string containing Thai-based columns

string2
A second string containing columns

Return value

The position of the column that begins the first occurrence of one string within another.

Usage

If *off* is greater than the length in bytes of *string1* or *string2*, the function returns an empty string.

Int function

Returns the nearest integer value that is less than or equal to a number.

Syntax

Int (*numExpr* **)**

Elements

numExpr
Any numeric expression.

Return value

The data type of *numExpr* determines the data type of the value returned by the Int function. The following table shows special cases.

numExpr	*Return value*
NULL	NULL
Variant containing a string interpretable as a number	Double

Usage

The value returned by the Int function is always less than or equal to its argument.

The Fix function and the Int function behave differently. Fix removes the fractional part of its argument, truncating toward 0.

Examples: Int function

```
Dim xF As Integer, yF As Integer
Dim xT As Integer, yT As Integer
xF% = Fix(-98.8)
yF% = Fix(98.2)
xT% = Int(-98.8)
yT% = Int(98.2)
Print xF%; yF%
' Output:
' -98  98
Print xT%; yT%
' Output:
' -99  98
```

Integer data type

Specifies a variable that contains a signed 2-byte integer.

Usage

An Integer value is a whole number in the range -32768 to 32767, inclusive.

Integer variables are initialized to 0.

The Integer suffix character for implicit type declaration is %.

LotusScript aligns Integer data on a 2-byte boundary. In user-defined data types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Integer data type

```
' The variable count is explicitly declared as type Integer.
' The variable nextOne is implicitly declared as type Integer
' by the % suffix character.
Dim count As Integer
count% = 1
nextOne% = count% + 1
Print count%; nextOne%' Output:   1  2
```

IsArray function

Tests the value of an expression to determine whether it is an array.

Syntax

IsArray (*expr* **)**

Elements

expr

Any expression.

Return value

IsArray returns TRUE (-1) if *expr* is an array; otherwise IsArray returns FALSE (0).

Examples: IsArray function

```
Dim arrayFixed(1 To 5)
Dim arrayDynam()
Print IsArray(arrayFixed)     ' Output:  True
Print IsArray(arrayDynam)     ' Output:  True

Dim v As Variant
Print IsArray(v)              ' Output:  False
v = arrayFixed
Print IsArray(v)              ' Output:  True
```

IsDate function

Tests the value of an expression to determine whether it is a date/time value.

Syntax

IsDate (*expr* **)**

Elements

expr

Any expression.

Return value

IsDate returns TRUE (-1) if *expr* is any of the following:

- A Variant value of DataType 7 (Date/Time)
- A Variant value of type String, where the string represents a valid date/time value
- A String value representing a valid date/time value

Otherwise IsDate returns FALSE (0).

Usage

A date/time value stored in a Variant is an 8-byte floating-point value. The integer part represents a serial day counted from Jan 1, 100 AD. Valid dates are represented by integers between -657434 (representing Jan 1, 100 AD) and 2958465 (representing Dec 31, 9999 AD). The fractional part represents the time as a fraction of a day, measured from time 00:00:00 (midnight on the previous day). In this representation of date/time values, day 1 is the date December 31, 1899.

Examples: IsDate function

```
Dim x As Variant, y As Variant, z As Variant
x = 100                          ' Numeric value
y = CDat(100)                    ' Numeric date value
z = "Nov 2, 1983"                ' String representing a date

Print IsDate(x)                  ' Output:  False
Print IsDate(y)                  ' Output:  True
Print IsDate(z)                  ' Output:  True
Print IsDate("100")              ' Output:  False
Print IsDate("Nov 2, 1983")      ' Output:  True
```

IsDefined function

Tests a string expression to determine whether it is the name of a product or platform constant at run time.

Syntax

IsDefined (*stringExpr* **)**

Elements

stringExpr

Any string expression.

Return value

IsDefined returns TRUE (-1) if *stringExpr* is the name of a product or platform constant at run time. Otherwise IsDefined returns FALSE (0).

Usage

The IsDefined function is used as a run-time parallel to the %If directive. It is commonly used to test the run-time value of a platform-identification or product constant that may be used to govern conditional compilation.

Note IsDefined is not a LotusScript keyword.

LotusScript implements the platform constants in the following table as product #defines. When one of these is used as *productConst*, the LotusScript compiler does not call the product to evaluate *productConst*. LotusScript itself evaluates the constant as TRUE or FALSE. The value of each constant depends on the platform LotusScript is running on.

Constant	*Platform or functionality*
WIN16	Windows with 16-bit API (Windows 3.1)
WIN32	Windows with 32-bit API (Windows NT or Windows® 95)
WINNT	Windows NT
WIN95	Windows 95
WIN40	Windows 95 or Windows NT 4.0
WIN98	Windows 98
WIN2K	Windows 2000
WINDOWS	Any Windows platform type (any of the above WIN*xx* constants)
HPUX	HP®/UNIX 9.X or greater
SOLARIS	Sun™/OS 5.0 or greater
UNIX	Any UNIX type (HP_UX® or Solaris®)
LINUX	Linux
OS2	OS/2, version 2.0 or greater
MAC	Macintosh System 7
OLE	OLE-2 is available
MAC68K	Macintosh Motorola® 68000 version (running on either a 68*xxx* Macintosh or the PowerPC™)
MACPPC	Macintosh PowerPC version

The constants can define platforms at different levels and are not mutually exclusive. For example, on WinNT, the platform returned can be WIN32_X86, WINNT, WIN32, or WINDOWS.

Product constants are defined by, and are specific to, the host product, for example Notes, 1-2-3, ESB, and so on. Refer to the product's documentation for a list of product-defined constants.

Examples: IsDefined function

```
' Perform operation based on platform

If IsDefined ("WIN32") Then

     < code >

End If

' Perform operation based on product

If Isdefined ("Notes_full_client") Then

     < code >

End If

See the %If directive example for a more detailed sample.
```

IsElement function

Tests a string to determine whether it is a list tag for a given list.

Syntax

IsElement (*listName* **(** *stringExpr* **))**

Elements

listName

The name of a defined list.

expr

Any expression.

Return value

The IsElement function returns TRUE (-1) if *stringExpr* is the list tag for any element of *listName*. Otherwise IsElement returns FALSE (0).

Usage

If *listName* is not the name of a defined list, LotusScript generates an error.

If *expr* is a numeric expression, LotusScript first converts its value to a string.

If the character set is single byte, Option Compare determines whether list names are case sensitive. For example, if Option Compare Case is in effect, the names "ListA" and "Lista" are different; if Option Compare NoCase is in effect, these names are the same. If the character set is double byte, list names are always case and pitch sensitive.

Examples: IsElement function

```
' Use IsElement to determine whether
' the user correctly identifies a list tag.
' Declare a list to hold employee Ids.
Dim empList List As Double
Dim empName As String, Id As Double
Dim found As Boolean
' 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 identify the list item to be removed.
empName$ = InputBox$("Which employee is leaving?")
' Check to see if empName$ corresponds to a list tag.
' If not, display a message and stop. Otherwise,
' validate the employee's Id.
' If everything checks out, remove the item from the list.
If IsElement(empList#(empName$)) = TRUE Then
   Id# = CDbl(InputBox$("What's " & empName$ & "'s Id?"))
   found = FALSE          ' Initialize found to 0 (FALSE)
   ForAll empId In empList#
      If empId = Id# Then
         found = TRUE     ' Set found to -1 (TRUE).
         If ListTag(empId) = empName$ Then
            Erase empList#(empName$)
            ' Verify the removal of the list element.
```

```
            If IsElement(empList#(empName$)) = FALSE Then
               MessageBox empName$ & _
                  " has been removed from the list."
            End If
         Else
            MessageBox "Employee name and Id do not match."
         End If
         ' No need to look farther for Id, so get out
         ' of the ForAll loop.
         Exit ForAll
      End If
   End ForAll
   If found = FALSE Then
      MessageBox "Not a valid employee Id."
   End If
Else
   MessageBox "We have no such employee."
End If
```

IsEmpty function

Tests the value of an expression to determine whether it is EMPTY.

Syntax

IsEmpty (*expr* **)**

Elements

expr

Any expression.

Return value

The IsEmpty function returns TRUE (-1) if *expr* has the value EMPTY. This occurs only if *expr* is a Variant and has not been assigned a value.

Otherwise IsEmpty returns FALSE (0).

Examples: IsEmpty function

```
Dim dynaVar As Variant
Print IsEmpty(dynaVar)          ' Output:  True
dynaVar = PI
Print IsEmpty(dynaVar)          ' Output:  False
```

IsList function

Tests the value of an expression to determine whether it is a list.

Syntax

IsList (*expr* **)**

Elements

expr

Any expression.

Return value

The IsList function returns TRUE (-1) if *expr* is a list; otherwise IsList returns FALSE (0).

Examples: IsList function

```
Dim myList List
Print IsList(myList)     ' Output:  True

Dim v As Variant
Print IsList(v)          ' Output:  False
v = myList
Print IsList(v)          ' Output:  True
```

IsNull function

Tests the value of an expression to determine whether it is NULL.

Syntax

IsNull (*expr* **)**

Elements

expr

Any expression.

Return value

IsNull returns TRUE (-1) if *expr* is NULL; otherwise it returns FALSE (0).

Usage

The IsNull function checks whether a Variant contains NULL. For example:

```
If IsNull(LoVar) Then Print "LoVar is NULL" Else Print LoVar
```

Examples: IsNull function

```
Dim v As Variant
Print IsNull(v)                ' Output:  False
Print IsEmpty(v)               ' Output:  True
v = NULL
Print IsNull(v)                ' Output:  True
```

IsNumeric function

Tests the value of an expression to determine whether it is numeric, or can be converted to a numeric value.

Syntax

IsNumeric (*expr* **)**

Elements

expr

Any expression.

Return value

The IsNumeric function returns TRUE (-1) if the value of *expr* is a numeric value or can be converted to a numeric value. The following values are numeric:

- Integer
- Long
- Single
- Double
- Currency
- Date/Time
- EMPTY
- String (if interpretable as number)
- OLE error
- Boolean (TRUE, FALSE)

If *expr* is not a numeric value and cannot be converted to a numeric value, IsNumeric returns FALSE (0). The following values are not numeric:

- NULL
- Array
- List
- Object (OLE Automation object, product object, or user-defined object)
- String (if it cannot be interpreted as number)
- NOTHING

Usage

A common use of IsNumeric is to determine whether a Variant expression has a numeric value.

Language cross-reference

@IsNumber

Examples: IsNumeric function

```
Dim v As Variant
Print IsNumeric(v)              ' Output:  True (v is EMPTY)
v = 12
Print IsNumeric(v)              ' Output:  True

' A string that is not interpretable as a number
v = "Twelve"
Print IsNumeric(v)              ' Output:  False

' A string that is interpretable as a number
v = "12"
Print IsNumeric(v)              ' Output:  True
```

IsObject function

Tests the value of an expression to determine whether it is a user-defined object, a product object, or an OLE Automation object.

Note The ability to use IsObject on OLE Automation objects is limited to Windows.

Syntax

IsObject (*expr* **)**

Elements

expr

Any expression.

Return value

The IsObject function returns TRUE (-1) if the value of *expr* is an object (user-defined object, product object, or OLE Automation object) or NOTHING. Otherwise IsObject returns FALSE (0).

Examples: IsObject function

```
' Define two classes, Vegetable and Fruit.
Class Vegetable
   ' ... class definition
End Class
Class Fruit
   ' ... class definition
End Class

Dim tomato As Variant, turnip As Variant
Print IsObject(tomato)          ' Output:  False
Set turnip = New Vegetable
Print IsObject(turnip)          ' Output:  True
Set tomato = New Fruit
Print IsObject(tomato)          ' Output:  True
```

IsScalar function

Tests an expression to determine if it evaluates to a single value.

Syntax

IsScalar (*expr* **)**

Elements

expr

Any expression.

Return value

The IsScalar function returns TRUE (-1) if *expr* evaluates to one of the following:

- EMPTY
- Integer
- Long
- Single
- Double
- Currency
- Date/Time
- String

- OLE error
- Boolean (TRUE, FALSE)

Otherwise (if *expr* is an array, list, object, NOTHING, or NULL), IsScalar returns FALSE (0).

Examples: IsScalar function

```
Dim var As Variant
Print IsScalar(var)              ' Output:  True
var = 1
Print IsScalar(var)              ' Output:  True
var = "hello"
Print IsScalar(var)              ' Output:  True

Class SenClass
   ' ... class definition
End Class
Set var = New SenClass
Print IsScalar(var)       ' Output:  False

Dim senArray(1 To 5)
var = senArray
Print IsScalar(var)       ' Output:  False

Dim senList List
var = senList
Print IsScalar(var)       ' Output:  False
```

IsUnknown function

Tests the value of an expression to determine whether it has the OLE value V_IUNKNOWN.

Syntax

IsUnknown (*expr* **)**

Elements

expr

Any expression.

Return value

The IsUnknown function returns True (-1) if *expr* is a Variant and the value of *expr* is V_IUNKNOWN. This value may be returned by a call to a property or method of an OLE Automation object. Otherwise IsUnknown returns False (0).

Examples: IsUnknown function

```
Dim dynaVar As Variant

dynaVar = PI

print IsUnknown(dynaVar) 'prints False
```

Join function

Concatenates all members of an Array of Strings and returns a string. Elements of the Array are separated by a delimiter, if provided, or the space character (" ").

Syntax

Join(*sourceArray* **as Variant, [***delimiter* **as String]) as String**

Elements

sourceArray

One-dimensional Array containing the substrings to be concatenated. sourceArray can be an array of Strings, or an array of Variants. If sourceArray is an array of Variants, Join will attempt to convert any non-string elements to Strings.

delimiter

Optional String containing separation character(s) for the concatenated Strings

Return value

Join returns a String containing the elements of sourceArray with delimiter between elements, or with the space character " " as a separator if delimiter is not specified.

Usage

Join creates a String that will hold the concatenation of sourceArray. Join then iterates through sourceArray, With each iteration, Join converts the next element of sourceArray to a String, if necessary, and appends it to the concatenation String. If more elements remain in sourceArray, a delimeter (either " " or the specified value) is appended to the concatenation String and Join continues to iterate. After all elements of sourceArray have been concatenated, Join returns the concatenation String.

Error handling

Join will throw a Run-time Type mismatch if:

- An element in a variant array cannot be coerced to a string.
- The delimiter is set to nothing.
- The array passed in is not of either type string or variant.
- A list is passed instead of an array.
- The array passed in contains an element set to nothing.
- The array passed in has not been properly initialized.

Join will throw a run-time Wrong Number of Dimensions error if the array is not one-dimensional.

Join will throw a run-time Invalid Use of Null error if the array passed in contains an element set to null or if the delimiter is set to null.

Note Join is an alias of Implode and is identical in every way.

Language cross-reference

@Implode function in formula language

Examples: Join function

```
Dim wordArray(1 to 2) As String

wordArray(1) = "hello"

wordArray(2) = "world"

Phrase = Join(wordArray)

Print Phrase          ' prints "hello world"
```

Kill statement

Deletes a file.

Syntax

Kill *fileName*

Elements

fileName

A string expression whose value is a file name; wildcards are not allowed. *fileName* can contain a drive indicator and path information.

Usage

Use Kill with care. If you delete a file with the Kill statement, you can't restore it with LotusScript statements or operating system commands. Make sure the file is closed before you attempt to delete it.

Kill deletes files, not directories. To remove directories, use the RmDir statement.

Note Kill actually may not be successful. The security setup may prevent LotusScript from actually deleting any files. For example, you can only delete a file when you run as an unrestricted agent in Domino. The Domino administrator sets up the rights to restrict what the agent can and cannot do to limit damage.

Examples: Kill statement

```
' Delete the file c:\test from the file system.
Kill "c:\test"
```

LBound function

Returns the lower bound for one dimension of an array.

Syntax

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

Elements

arrayName

The name of an array

dimension

Optional. An integer argument that specifies the array dimension; the default is 1.

Return value

The LBound function returns an Integer.

Usage

The default value for *dimension* is 1.

LotusScript sets the lower bound for each array dimension when you declare a fixed array or define the dimensions of a dynamic array with a ReDim statement.

The default lower bound for an array dimension is 0 or 1, depending on the Option Base setting.

Examples: LBound function

```
' Single dimension array

Dim minima(10 To 20)

Print LBound(minima)           ' Output:  10

' 2-dimensional array

Dim minima(1 to 5, 2 to 10)

Print LBound(minima,2)   ' Output: 2

' 3-dimensional array

Dim minima(1 to 5, 5 to 10, 10 to 15)

Print LBound(minima,1)   ' Output: 1

Print LBound(minima,2)   ' Output: 5

Print LBound(minima,3)   ' Output: 10
```

LCase function

Returns the lowercase representation of a string.

Syntax

LCase[**$**] **(** *expr* **)**

Elements

expr

Any numeric or String expression for LCase; and any Variant or String expression for LCase$.

Return value

LCase returns a Variant of DataType 8 (a String), and LCase$ returns a String.

Usage

LCase ignores non-alphabetic characters.

LCase(NULL) returns NULL. LCase$(NULL) returns an error.

Language cross-reference

@LowerCase function in formula language

Examples: LCase function

```
Print LCase$("ABC")            ' Output:  "abc"
```

Left function

Extracts a specified number of the leftmost characters in a string.

Syntax

Left[$] (*expr* **,** *n* **)**

Elements

expr

Any numeric or String expression for Left; and any Variant or String expression for Left$. If *expr* is numeric, LotusScript converts it to a string before performing the extraction.

n

The number of characters to be returned.

Return value

Left returns a Variant of DataType 8 (a String), and Left$ returns a String.

If *n* is 0, the function returns the empty string (“”). If *n* is greater than the length (in characters) of *expr*, the function returns the entire string.

Left(NULL) returns NULL. Left$(NULL) is an error.

Language cross-reference

@Left function in formula language

Examples: Left function

```
' Assign the leftmost 2 characters in "ABC".
Dim subString As String
subString$ = Left$("ABC", 2)
Print subString$                  ' Output:  "AB"
```

LeftB function

Lotus does not recommend using the LeftB function in LotusScript Release 3 and later because these releases use Unicode, a character set encoding scheme that represents each character as two bytes. Because a two-byte character can be accompanied by leading or trailing zeroes, extracting characters by byte position no longer yields reliable results.

Use the Left function for left character set extractions instead.

LeftBP function

Extracts a specified number of the leftmost bytes in a string using the platform-specified character set.

Syntax

LeftBP[**$**] **(** *expr* **,** *n* **)**

Elements

expr

Any numeric or String expression for LeftBP; and any Variant or String expression for LeftBP$. If *expr* is numeric, LotusScript converts it to a string before performing the extraction.

n

The number of bytes to be returned using the platform-specified character set.

Return value

LeftBP returns a Variant of DataType 8 (a String), and LeftBP$ returns a String.

If *n* is 0, the function returns the empty string (“”). If *n* is greater than the length (in bytes) of *expr*, the function returns the entire string.

LeftBP(NULL) returns NULL. LeftBP$(NULL) is an error.

If a double-byte character is divided, the character is not included.

Examples: LeftBP function

```
' The value "AB" or other value depending on platform
' is assigned to the variable subString.

Dim subString As String
subString = LeftBP$("ABC", 2)
Print subString$  ' Output:  "AB"
```

LeftC function

Extracts the leftmost *n* columns from a string for column-based writing systems, such as Thai and Vietnamese.

Syntax

LeftC[$] (*StringExpr*, *n***)**

Elements

StringExpr

A String expression containing character columns.

n

The number of columns to be returned using the platform-specified character set.

Return value

LeftC returns a Variant containing the columns specified by *n*. LeftC$ returns a String.

Usage

If *n* is 0, the function returns the empty string (“”). If *n* is greater than the length (in columns) of *StringExpr*, the function returns the entire string.

LeftC supports the Thai and Vietnamese languages.

Examples: LeftC function

```
'Extracts the leftmost 6 Thai columns from a string.
LeftC("XXXxxxXXXxxxXxXxxXxxX", 6)
'Returns "XXXxxx"
```

Len function

Returns the number of characters in a string, or the number of bytes used to hold a numeric value.

Syntax

Len ({ *stringExpr* | *variantExpr* | *numericExpr* | *typeName* } **)**

Elements

stringExpr

Any string expression.

variantExpr

Any Variant expression that includes a variable name.

numericExpr

The name of a variable, an element of an array, an element of a list, or a member variable of a user-defined data type or class. The data type of *numericExpr* is numeric.

typeName

An instance of a user-defined data type. It can be a simple variable of that data type, or an element of an array variable or a list variable of that data type.

Return value

For *stringExpr*, Len returns the number of characters in the string expression.

For *variantExpr*, Len returns the number of characters required to hold the value of *variantExpr* converted to a String.

For *numericExpr*, Len returns the number of bytes required to hold the contents of *numericExpr*.

For *typeName*, Len returns the number of bytes required to hold the contents of all the member variables, unless the user-defined data type includes Variant or variable-length String members. In that case, the length of the variable of the user-defined data type may not be the same as the sum of the lengths of its member variables.

Usage

In LotusScript Release 3 and later, Len(NULL) generates an error. In earlier releases of LotusScript, Len(NULL) returned NULL.

Len(*v*), where *v* is EMPTY, returns 0.

To determine the length of a string in bytes rather than in characters, use the LenB function. To determine the length of a string in bytes in the platform-native character set, use the LenBP function.

Language cross-reference

@Length function in formula language

Examples: Len function

Example 1

```
' The length of a string, in characters
Dim theString As String
theString$ = "alphabet"
Print Len(theString$) ' Output:  8
```

```
' The number of bytes used to hold a Single variable
Dim singleVar As Single
Print Len(singleVar!) ' Output:  4
```

Example 2

```
' User-defined data type with variable-length String member
Type OrderInfo
   ordID As String * 6
   custName As String
End Type

' An instance of the user-defined data type
Dim ord As OrderInfo
ord.ordID$ = "OR1234"
ord.custName$ = "John R. Smith"

' Total length of the ord's members is 19.
Print Len(ord.ordID$) + Len(ord.custName)

' Length of ord is 16.
Print Len(ord)
```

LenB function

Returns the length of a string in bytes, or the number of bytes used to hold a variable.

Syntax

LenB ({ *stringExpr* | *variantExpr* | *numericExpr* | *typeName* } **)**

Elements

stringExpr
Any string expression.

variantExpr
Any Variant expression that includes a variable name.

numericExpr
The name of a variable, an element of an array, an element of a list, or a member variable of a user-defined data type or class. The data type of *numericExpr* is numeric.

typeName
An instance of a user-defined data type. It can be a simple variable of that data type, or an element of an array variable or a list variable of that data type.

Return value

For *stringExpr,* LenB returns the number of bytes in the string expression.

For *variantExpr,* LenB returns the number of bytes required to hold the value of *variantExpr* converted to a String.

For *numericExpr,* LenB returns the number of bytes required to hold the contents of *numericExpr*.

For *typeName,* LenB returns the number of bytes required to hold the contents of all the member variables, unless the user-defined data type includes Variant or variable-length String members. In that case, the length of the variable of the user-defined data type may not be the same as the sum of the lengths of its member variables.

Usage

In LotusScript Release 3 and later, LenB(NULL) generates an error. In earlier releases of LotusScript, LenB(NULL) returned NULL.

LenB(*v*), where *v* is EMPTY, returns 0.

To determine the length of a string in characters, use the Len function. To determine the length of a string in bytes in the platform-native character set, use the LenBP function.

Examples: LenB function

```
The length of an 8-character string, in bytes

Dim theString As String

theString$ = "alphabet"

Print LenB(theString$)' Output:  16

' The number of bytes used to hold a Single variable

Dim singleVar As Single

Print LenB(singleVar!)' Output:  4
```

LenBP function

Returns the length of a string in bytes, or the number of bytes used to hold a variable, in the platform-native character set.

Syntax

LenBP ({ *stringExpr* | *variantExpr* | *numericExpr* | *typeName* } **)**

Elements

stringExpr

Any string expression.

variantExpr

Any Variant expression that includes a variable name.

numericExpr

The name of a variable, an element of an array, an element of a list, or a member variable of a user-defined data type or class. The data type of *numericExpr* is numeric.

typeName

An instance of a user-defined data type. It can be a simple variable of that data type, or an element of an array variable or a list variable of that data type.

Return value

For *stringExpr,* LenBP returns the number of bytes in the string expression.

For *variantExpr,* LenBP returns the number of bytes required to hold the value of *variantExpr* converted to a String.

For *numericExpr,* LenBP returns the number of bytes required to hold the contents of *numericExpr*.

For *typeName,* LenBP returns the number of bytes required to hold the contents of all the member variables, unless the user-defined data type includes Variant or variable-length String members. In that case, the length of the variable of the user-defined data type may not be the same as the sum of the lengths of its member variables.

Usage

LenBP(NULL) generates an error.

LenBP(*v*), where *v* is EMPTY, returns 0.

To determine the length of a string in characters, use the Len function. To determine the length of a string in bytes in the LotusScript internal character set, use the LenB function. To determine the length of a string in columns (for column-based languages) use the LenC function.

Examples: LenBP function

```
' with String, returns length of String

Dim word As String

word = "hello"

Print LenBP(word)       ' prints 5
```

```
' with Integer, returns number of bytes used to store variable

Dim x As Integer

x = 15

Print LenBP(x)          ' prints 2

' with Variant, returns number of bytes used to store value of
Variant converted to a String

Dim y As Variant

y = 3.14

Print LenBP(y)          ' prints 4
```

LenC function

Returns the length of a string in number of character columns. The LenC function is used for column based writing systems, such as Thai.

Syntax

LenC(*stringExpr***)**

Elements

stringExpr

A string expression using character columns.

Return value

An Integer indicating the number of columns in the string expression.

Usage

LenC(NULL) generates an error.

LenC(*v*), where *v* is EMPTY, returns 0.

To determine the length of a string in characters, use the Len function. To determine the length of a string in bytes in the LotusScript internal character set, use the LenB function. To determine the length of a string in columns (for column-based languages) use the LenC function.

Let statement

Assigns a value to a variable.

Syntax

[**Let**] *variableID* **=** *expr*

Elements

Let

Optional. The Let statement is chiefly useful as a means of documenting an assignment statement. The absence of the Let keyword has no effect on the assignment.

variableID

A variable or variable element to which the value of *expr* is assigned. *variableID* can be of any data type that LotusScript recognizes, other than an object reference, an array, or a list. *variableID* can take any of these forms:

- *variableName*

  A non-array, non-list variable. The variable may not be an array or list variable, but it may be a Variant containing an array or list.

- *arrayName* **(** *subscripts* **)**

  An array element. *arrayName* is an array variable or a Variant containing an array.

- *listName* **(** *listTag* **)**

  A list element. *listName* is a list variable or a Variant containing a list.

- *typeVar***.***memberVar*

  A member variable of a user-defined data type. *typeVar* is an instance of a user-defined data type. *typeVar* can be an element of an array or list. *memberVar* is a member variable of that user-defined data type. *memberVar* can be a scalar data type, a fixed array, or a Variant containing a scalar data type, an array, a list, or an object reference.

- *object***.***memberVar*
  *object***..***memberVar*
  Me.*memberVar*

  A member variable or property of a class. *object* is an expression whose value is an object reference. *memberVar* is a member variable or property of that class, or an element of an array member variable, or an element of a list member variable. Use **Me** only within a procedure defined within the class.

expr

Any expression except one whose value is an object reference. The *expr* must be of the same data type as *variableID*, or else must be convertible to the data type of *variableID*. The rules for data type conversion determine how (if at all) LotusScript converts the value of *expr* before assigning it to *variableID*.

Usage

LotusScript assigns the value of *expr* to the variable or variable element named by *variableID*.

Do not use the Let statement to assign an object reference to a variable. Use the Set statement to do that.

Language cross-reference

DEFAULT keyword in formula language

Examples: Let statement

```
' This example shows several cases of assignment.
' Wherever the keyword Let appears, it can be omitted
' without effect.

Dim a As Integer, b As Integer, c As Integer
Let a% = 2
Let b% = a%
Print b%                          ' Output:  2
Let c% = b% + 1
Print c%                          ' Output:  3

' Assign the value of b to an array element.
Dim devArray(3)
Let devArray(1) = b%
Print devArray(1)                 ' Output:  2

' Assign the value of c to a list element.
Dim devList List
Let devList("one") = c%
Print devList("one")              ' Output:  3

' For an instance of a user-defined data type,
' assign the value of c - a to a member variable.
Type DevType
   num As Integer
End Type
Dim inst As DevType
Let inst.num% = c% - a%
Print inst.num%                    ' Output:  1

' For an instance of a user-defined class,
' assign the value of a + b to a member variable.
```

```
Class DevClass
   Public num% As Integer
End Class
Set devObj = New DevClass
Let devObj.num% = a% + b%
Print devObj.num%                    ' Output:  4
```

Line Input # statement

Reads a line from a sequential file into a String or Variant variable.

Syntax

Line Input #*fileNumber* **,** *varName*

Elements

#*fileNumber*

The number assigned to the file when you opened it. A # sign must precede the file number.

varName

A String or Variant variable to hold the contents of one line of the file.

Usage

Line Input # reads characters from a sequential file until it encounters a newline character. Line Input # does not read the newline character into the variable.

Note Newline does not mean either chr(10) or chr(13) on all platforms. Newline is the character or sequence of characters that is used to mark the end of a line. This may be chr(10), or chr(13), but it may also be something else, because the actual value of newline depends on the platform.

The Line Input # statement will handle the line end character appropriate for the current platform. It will not necessarily handle line ends properly if the file is written on one platform and read on another.

When reading a multiline string from a sequential file, use the Input # statement, not the Line Input # statement.

Examples: Line Input # statement

```
' Display the contents of c:\config.sys a line at a time.

Dim text As String, fileNum As Integer
fileNum% = FreeFile()

Open "c:\config.sys" For Input As fileNum%
Do While Not EOF(fileNum%)
   Line Input #1, text$
```

```
   Print text$          ' Prints one line of config.sys
Loop

Close fileNum%
```

ListTag function

Returns the name of the list element currently being processed by a ForAll statement.

Syntax

ListTag (*refVar* **)**

Elements

refVar

The reference variable in a ForAll list iteration loop.

Return value

ListTag returns a String that is the name of the list element currently referred to by *refVar.*

ListTag generates an error if *refVar* is not the reference variable specified in the ForAll statement.

If Option Compare NoCase is in effect and the character set is single byte, names are returned as all uppercase. Option Compare has no effect if the character set is double byte.

Usage

The ListTag function is valid only inside a ForAll block whose target is a list.

Examples: ListTag function

```
Dim loft List As Integer
loft%("first") = 0
loft%("second") = 1
loft%("third") = 2

' Print list tags for the elements of Loft,
' each on its own line.
ForAll i In Loft%
   Print ListTag(i)
End ForAll
' Output:
' first
' second
' third
```

LOC function

Returns the current position of the file pointer in a file.

Syntax

LOC (*fileNumber* **)**

Elements

fileNumber

The number assigned to the file when you opened it.

Return value

The following table presents the LOC return values for random, sequential, and binary files.

File type	*Return value*
Random	The number of the last record read from or written to the file. This is the file pointer position, minus 1.
Sequential	The byte position in the file, divided by 128 and truncated to an integer.
Binary	The position of the last byte read from or written to the file. This is the file pointer position, minus 1.

Examples: LOC function

```
Type PersonRecord
   empNumber As Integer
   empName As String *20
End Type

Dim rec1 As PersonRecord, rec2 As PersonRecord
Dim fileNum As Integer
Dim fileName As String
fileNum% = FreeFile()
fileName$ = "data.txt"

' Create a sample file.
Open fileName$ For Random As fileNum%

' Write at record 1.
rec1.empNumber% = 123
rec1.empName$ = "John Smith"
Put #fileNum%, 1, rec1
Print LOC(fileNum%)       ' Output:  1

' Write at record 2.
rec2.empNumber% = 456
rec2.empName$ = "Jane Doe"
```

```
Put #fileNum%, 2, rec2
Print LOC(fileNum%)       ' Output:  2

' Read from record 1.
Get #fileNum%, 1, rec2
Print LOC(fileNum%)       ' Output:  1

Close fileNum%
```

Lock and Unlock statements

Provide controlled access to files.

Syntax

Lock [**#**]*fileNumber* [**,** *recordNumber* | { [*start*] **To** *end* }]

Unlock [**#**]*fileNumber* [**,** *recordNumber* | { [*start*] **To** *end* }]

Elements

fileNumber

The number assigned to the file when you opened it.

recordNumber

In a random file, the number of the record that you want to lock or unlock. In a binary file, the byte that you want to lock or unlock. The first record in a random file is record number 1; the first byte in a binary file is byte number 1. LotusScript locks or unlocks only the specified record or byte.

In a sequential file, LotusScript locks or unlocks the whole file, regardless of value you specify for *recordNumber*.

start **To** *end*

In a random file, the range of record numbers you want to lock or unlock. In a binary file, the range of bytes that you want to lock or unlock. If you omit *start,* LotusScript locks records or bytes from the beginning of the file to the specified *end* position. In a sequential file, LotusScript locks or unlocks the whole file, regardless of the *start* and *end* values.

Usage

In Windows 3.1, you must run SHARE.EXE to enable the locking feature if you are using MS-DOS® version 3.1 or later. Earlier versions of MS-DOS do not support Lock and Unlock.

Always use Lock and Unlock statements in pairs whose elements — *fileNumber*, *recordNumber*, *start*, and *end* — match exactly. If you do not remove all locks, or if the elements do not match exactly, unpredictable results can occur.

Language cross-reference

@DocLock function in formula language

Examples: Lock and unlock statements

```
Type PersonRecord
   empNumber As Integer
   empName As String * 20
End Type

Dim rec1 As PersonRecord, rec2 As PersonRecord
Dim fileNum As Integer, recNum As Integer
Dim fileName As String
recNum% = 1
fileNum% = FreeFile()
fileName$ = "data.txt"

' Create a record.
Open fileName$ For Random As fileNum%
rec1.empNumber% = 123
rec1.empName$ = "John Smith"
Put #fileNum, recNum%, rec1
Print rec1.empName$ ; rec1.empNumber%
' Output:
' John Smith           123

' Lock and update the record.
Lock #fileNum%, recNum%
Get #fileNum%, recNum%, rec2
Print rec2.empName$ ; rec2.empNumber%
' Output:
' John Smith           123
rec2.empName$ = "John Doe"
Put #fileNum%, recNum%, rec2
Print rec2.empName$ ; rec2.empNumber%
' Output:
' John Doe             123

' Release the lock.
Unlock #fileNum%, recNum%
Close fileNum%
```

LOF function

Returns the length of an open file in bytes.

Syntax
LOF (*fileNumber* **)**

Elements
fileNumber
The number assigned to the file when you opened it.

Return value
The LOF function returns a value of type Long.

Usage
LOF works only on an open file. To find the length of a file that isn't open, use the FileLen function.

Language cross-reference
@DocLength function in formula language

Examples: LOF function
```
Dim izFile As Integer
Dim fileName As String, fileContents as String

izFile% = FreeFile()
fileName$ = "c:\autoexec.bat"
Open fileName$ For Input As izFile%

' Use LOF to find the file length, and Input$ to read
' the entire file into the string veriable izFile.
fileContents$ = Input$(LOF(izFile%), izFile%)
Print fileContents$        ' Display the file contents.
```

Log function

Returns the natural (base *e*) logarithm of a number.

Syntax
Log (*numExpr* **)**

Elements
numExpr
Any numeric expression greater than zero.

Return value
The Log function returns a value of type Double.

Usage
The base for natural logarithms (*e*) is approximately 2.71828.

Language cross-reference
@Ln function in formula language

Examples: Log function

Example 1

```
Dim natLog As Double
natLog# = Log(18)               ' Assigns 2.89037175789617
```

Example 2

```
' Compute the base 10 logarithm of a number.
Function Log10 (inVal As Single) As Single
   Log10 = Log(inVal!) / Log(10)
End Function

Print Log10(10)                 ' Output:  1
Print Log10(100)                ' Output:  2
Print Log10(1 / 100)            ' Output:  -2
Print Log10(1)                  ' Output:  0
```

Long data type

Specifies a variable that contains a signed 4-byte integer.

Usage
The Long suffix character is &.

Long variables are initialized to 0.

A Long value is a whole number in the range -2,147,483,648 to 2,147,483,647 inclusive.

LotusScript aligns Long data on a 4-byte boundary. In user-defined types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Long data type

```
' Explicitly declare a Long variable.
Dim particles As Long

' Implicitly declare a Long variable.
bigInt& = 2094070921
```

```
particles = bigInt&
Print bigInt&; particles ' Output:  2094070921   2094070921
```

LSet statement

Assigns a specified string to a string variable and left-aligns the string in the variable.

Syntax

LSet *stringVar* **=** *stringExpr*

Elements

stringVar

The name of a string variable. It may be a fixed-length String variable, a variable-length String variable, or a Variant variable.

stringExpr

The string to be assigned to the variable and left-aligned.

Usage

If the length of *stringVar* is greater than the length of *stringExpr*, LotusScript left-aligns *stringExpr* in *stringVar* and sets the remaining characters in *stringExpr* to spaces.

If the length of *stringVar* is less than the length of *stringExpr*, LotusScript copies only that many of the leftmost characters from *stringExpr* to *stringVar*.

If *stringVar* contains a numeric value, LotusScript converts it to a string to determine the length of the result.

If *stringVar* is a Variant, it can't contain NULL.

You can't use LSet to assign values from an instance of one user-defined data type to another.

Examples: LSet statement

```
Dim x As Variant
x = "qq"            ' Length of x is 2
LSet x = "abc"      ' Assigns leftmost 2 characters
Print x             ' Prints "ab"
LSet x = "c"
' Assigns "c" and pads on the right with a space
' because length of x is 2
Print x & "high"    ' Prints "c high"
x = "c"             ' Ordinary assignment; new length of x is
```

```
1
Print x & "high"    ' Prints "chigh"
```

LTrim function

Removes leading spaces from a string and returns the result.

Syntax

LTrim[$] (*stringExpr* **)**

Elements

stringExpr

Any string expression.

Return value

LTrim returns the trimmed version of *stringExpr* without modifying the contents of *stringExpr* itself. LTrim returns a Variant of DataType 8 (a String), and LTrim$ returns a String.

Examples: LTrim function

```
Dim trimLeft As String
trimLeft$ = LTrim$("   abc ")
Print trimLeft$
Print Len(trimLeft$)
' Output:
' abc
'  4
' The string "abc " is assigned to trimLeft.
' Note that the trailing space was not removed.
```

MessageBox function and statement

Displays a message in a message box and waits for user acknowledgment. The function form returns a value corresponding to the button the user presses.

Function Syntax

MessageBox (*message* [**,** [*buttons* + *icon* + *default* + *mode*] [**,** *boxTitle*]] **)**

Statement Syntax

MessageBox *message* [**,** [*buttons* + *icon* + *default* + *mode*] [**,** *boxTitle*]]

The MessageBox function and statement are identical, except that only the function has a return value.

MsgBox is acceptable in place of MessageBox.

Elements

message

The message to be displayed in the message box (a string). The length of *message* is dependent on the operating system.

buttons

Defines the number and type of buttons to be displayed in the message box:

Constant name	*Value*	*Buttons displayed*
MB_OK	0	OK
MB_OKCANCEL	1	OK and Cancel
MB_ABORTRETRYIGNORE	2	Abort, Retry, and Ignore
MB_YESNOCANCEL	3	Yes, No, and Cancel
MB_YESNO	4	Yes and No
MB_RETRYCANCEL	5	Retry and Cancel

icon

Defines the icons to be displayed in the message box:

Constant name	*Value*	*Icon displayed*
MB_ICONSTOP	16	Stop sign
MB_ICONQUESTION	32	Question mark
MB_ICONEXCLAMATION	48	Exclamation point
MB_ICONINFORMATION	64	Information

default

Defines the default button in the message box. Pressing ENTER has the same effect as clicking the default button:

Constant name	*Value*	*Default button*
MB_DEFBUTTON1	0	First button
MB_DEFBUTTON2	256	Second button
MB_DEFBUTTON3	512	Third button

mode

Defines the message box modality:

Constant name	*Value*	*Description*
MB_APPLMODAL	0	Application modal. Stops the current application until the user responds to the message box.
MB_SYSTEMMODAL	4,096	System modal. Stops all applications until the user responds to the message box.

boxTitle

The string to appear in the title bar of the message box. *boxTitle* can be up to 128 characters in length.

Return value

The MessageBox function return value is an integer in the range of 1 to 7, inclusive. This value indicates which button the user pressed in the message box, as shown in the following table.

Return value	*Button*	*Constant*
1	OK	IDOK
2	Cancel	IDCANCEL
3	Abort	IDABORT
4	Retry	IDRETRY
5	Ignore	IDIGNORE
6	Yes	IDYES
7	No	IDNO

Usage

The valid values for the *buttons*, *icon*, *default*, and *mode* elements listed in the preceding tables are defined as constants in the file LSCONST.LSS. If you want to use the constants instead of numbers, include this file in your script.

The Lotus software where you are running LotusScript may allow longer strings than described above for *message* and *boxTitle*. LotusScript will support longer strings for these items if the Lotus software does.

Note The length of *message* is dependent on the operating system. If you are launching applications in a mixed environment (for example, PC and Mac), keep your *message* length equal to or shorter than the smallest limit of the operating systems to be used.

Use the newline character to force line breaks in the *message* element. Or use vertical bars or braces to specify a multiline string. If you don't force line breaks, the text wraps automatically in the message box.

Note Newline does not mean either chr(10) or chr(13) on all platforms. Newline is the character or sequence of characters that is used to mark the end of a line. This may be chr(10), or chr(13), but it may also be something else, because the actual value of newline depends on the platform. If newlines are desired in the output, it is the programmer's responsibility to ensure that the string contains the correct newline for the platform.

If you are using LotusScript from within Lotus Notes, note that the MessageBox function writes to:

- A dialog box when executing in the foreground on a Notes client. The user clicks OK, Cancel, Abort, Retry, Yes, or No to continue.
- NOTES.LOG when executing on a Domino server without pausing or as a scheduled agent in the Notes client.

Note Whenever a MessageBox function is executed in the back end, it will return zero, regardless of the defaults or modes. Only the prompt is displayed. The display goes to the server console, Notes log, and anywhere that debugging output is redirected (DEBUG_OUTFILE) if on a server, or to the debug console if on the client. This does not appy to the MessageBox statement.

Examples: MessageBox function and statement

Example 1

```
' Display the message "Do you want to continue?"
' in a message box labeled "Continue?" and containing
' Yes and No buttons. Assign the return value from
' the MessageBox function to the variable answer.
%Include "lsconst.lss"
Dim boxType As Long, answer As Integer
boxType& = MB_YESNO + MB_ICONQUESTION
answer% = MessageBox("Do you want to continue?", boxType&, _
  "Continue?")
```

Example 2

```
' Use the MessageBox statement to display a
' multiline message in a message box labeled "Demo"
' and containing an OK button.
%Include "lsconst.lss"
Dim twoLiner As String
twoLiner = |This message
is on two lines|
MessageBox twoLiner, MB_OK, "Demo"
```

Mid function

Extracts a string from within another string, beginning with the character at a specified position.

Syntax

Mid[**$**] **(** *expr* **,** *start* [**,** *length*] **)**

Elements

expr

Any numeric or string expression. LotusScript converts a numeric to a string before performing the extraction.

start

The position of the first character to extract from the string, counting from 1 for the leftmost character.

length

The number of characters to extract from the string.

Return value

Mid returns a Variant of DataType 8 (a string), and Mid$ returns a String.

If there are fewer than *length* characters in the string beginning at the *start* position, or if you omit the *length* argument, the function returns a string consisting of the characters from *start* to the end of *expr*.

If *start* is greater than the length of *expr*, the function returns the empty string (“”).

Language cross-reference

@Middle function in formula language

Examples: Mid function

```
Dim subString As String
subString$ = Mid$("ABCDEF", 2, 3)
Print subString$        ' Output: BCD
```

Mid statement

Replaces part or all of one string with characters from another string.

Syntax

Mid[**$**] **(** *stringVar* **,** *start* [**,** *length*] **) =** *stringExpr*

Elements

stringVar

A String variable, or a Variant variable containing a string value. The *stringVar* cannot be a literal string.

start

The position of the first character in *stringVar* that you want to replace.

length

Optional. The number of characters you want to use from *stringExpr*.

stringExpr

A string expression. Characters from *stringExpr* replace characters in *stringVar.*

Usage

Mid can alter the size of *stringVar* in bytes if you are working with multibyte characters. For example, if you are replacing a single-byte character with a double-byte character, the size of the string in bytes increases.

Otherwise, Mid does not alter the length of *stringVar*. That is, Mid does not append characters to *stringVar*. Mid uses as many characters of *stringExpr* as will fit in *stringVar* beginning at *start* and ending at *start* + *length* – 1.

To direct Mid to use all of *stringExpr*, either omit *length*, or specify a *length* greater than the length of the value in *stringExpr*.

If *start* is greater than the length of *stringVar*, LotusScript generates an error.

Language cross-reference

@ReplaceSubstring function in formula language

Examples: Mid statement

```
Dim string1 As String, string2 As String
string1$ = "ABCDEF"
string2$ = "12345"
' Replace the characters "BCD" in string1
' with the characters "123" in string2.
```

```
Mid$(string1$, 2, 3) = string2$
Print string1$        ' Output: A123EF
```

The three-character string "BCD", beginning at the second character of string1, is replaced with the first three characters contained in string2, "123".

MidB function

Lotus does not recommend using MidB in LotusScript Release 3 or later. Because these releases use Unicode, extracting characters by byte position no longer yields reliable results.

Instead, use the Mid function for character set extractions.

MidB statement

Lotus does not recommend using MidB statements in LotusScript Release 3 or later. Because these releases use Unicode, replacing characters by byte position no longer yields reliable results.

Instead, use the Mid statement for character set replacement.

MidBP function

Extracts a number of bytes (using the platform-specified character set) from within another string, beginning at a specified position.

Syntax

MidBP[$] (*expr* **,** *start* **[,** *length***])**

Elements

expr

Any numeric or String expression for MidBP; and any Variant or String expression for MidBP$. If *expr* is numeric, LotusScript converts it to a string before performing the extraction.

start

The position of the first byte in *expr* that you want to return.

length

Optional. The number of characters you want to use from *expr*.

Return value

MidBP returns a Variant of DataType 8 (a String), and LeftBP$ returns a String.

If there are fewer than *length* bytes in the string beginning at the *start* position, or if you omit the *length* argument, the function returns a string consisting of the characters from *start* to to the end of *expr*.

If *start* is greater than the length in bytes of *expr*, the function returns an empty string.

If a double-byte character is divided, the character is not included.

Examples: MidBP function

```
' The value "BCD" or other value depending on platform
' is returned.

Print MidBP("ABCDE"; 2; 3)
```

MidC function

Extracts a number of character columns from a string starting at a character column offset, searching left to right. The MidC function is used for column-based writing systems, such as Thai.

Syntax

Midc(*string*, *off*, *n***)**

Elements

string

A string containing character-based columns

off

The number of the offset where you want to begin extraction

n

The number of columns to be extracted

Return value

MidC returns a string of length *n*.

Usage

If there are fewer than *n* columns in the string beginning at the *off* position, or if you omit the *n* argument, the function returns a string consisting of the characters from *off* to the end of *string*.

If *off* is greater than the length in bytes of *string*, the function returns an empty string.

Minute function

Returns the minute of the hour (an integer from 0 to 59) for a date/time argument.

Syntax

Minute (*dateExpr* **)**

Elements

dateExpr

Any of the following kinds of expression:

- A valid date/time string of type String or Variant.

  For Notes or Domino, LotusScript interprets a 2-digit designation of a year in a date/time string so that:

- 50 through 99 represent the years 1950 through 1999.
- 00 through 49 represent the years 2000 through 2049.

  For SmartSuite, LotusScript interprets the years differently. For more information, see the Year 2000 item on the Help menu of each SmartSuite product.

- A numeric expression whose value is a Variant of DataType 7 (Date/Time).
- A number within the valid date range: the range -657434 (representing Jan 1, 100 AD) to 2958465 (Dec 31, 9999 AD), inclusive.
- NULL.

Return value

Minute returns an integer between 0 and 59.

The data type of the return value is a Variant of DataType 2 (Integer).

Minute(NULL) returns NULL.

Language cross-reference

@Minute function in formula language

Examples: Minute function

```
' Construct a message that displays the current time and
' the number of hours, minutes, and seconds
' remaining in the day.
Dim timeFrag As String, hoursFrag As String
Dim minutesFrag As String, secondsFrag As String
Dim crlf As String, message As String
timeFrag$ = Format(Time, "h:mm:ss AM/PM")
hoursFrag$ = Str(23 - Hour(Time))
```

```
minutesFrag$ = Str(59 - Minute(Time))
secondsFrag$ = Str(60 - Second(Time))
crlf$ = Chr(13) & Chr(10)   ' Carriage return/line feed
message$ = "Current time: " & timeFrag$ & ". " & crlf$ _
   & "Time remaining in the day: " _
   & hoursFrag$ & " hours, " _
   & minutesFrag$ & " minutes, and " _
   & secondsFrag$ & " seconds."
MessageBox(message$)
```

MkDir statement

Creates a directory.

Syntax

MkDir *path*

Elements

path

A string expression whose value is the name of the directory you want to create.

Usage

A drive letter in *path* is optional. If it is not included, the current drive is used. Relative pathnames may also be used.

Use the path syntax for the platform on which you are running LotusScript. The maximum allowable length of the *path* string varies with the platform.

LotusScript generates an error if the directory cannot be created.

Examples: MkDir statement

```
' Create directory TEST, in the root directory of drive C.
MkDir "c:\test"
```

Month function

Returns the month of the year (an integer from 1 to 12) for a date/time argument.

Syntax

Month (*dateExpr* **)**

Elements

dateExpr

Any of the following kinds of expression:

- A valid date/time string of String or Variant data type.

  For Notes or Domino, LotusScript interprets a 2-digit designation of a year in a date/time string so that:

- 50 through 99 represent the years 1950 through 1999.
- 00 through 49 represent the years 2000 through 2049.

  For SmartSuite, LotusScript interprets the years differently. For more information, see the Year 2000 item on the Help menu of each SmartSuite product.

- A numeric expression whose value is a Variant of DataType 7 (Date/Time).
- A number within the valid date range: the range -657434 (representing Jan 1, 100 AD) to 2958465 (Dec 31, 9999 AD), inclusive.
- NULL.

Return value

Month returns an integer between 1 and 12.

The data type of the return value is a Variant of DataType 2 (Integer).

Month(NULL) returns NULL.

Language cross-reference

@Month function in formula language

Examples: Month function

```
Dim x As Long
Dim mm As Integer
x& = DateNumber(1994, 4, 1)
mm% = Month(x&)
Print mm%
' Output:
' 4
```

Name statement

Renames a file or directory.

Syntax

Name *oldName* **As** *newName*

Elements

oldName

A string expression whose value is the name of an existing file or directory, optionally including a path.

newName

A string expression whose value is the name to be given to the file or directory, optionally including a path. The *newName* cannot be another file or directory that already exists.

Usage

To move a file, specify complete paths in both *oldName* and *newName*. Use the same file name for both arguments if you don't want to rename it.

You can't move a file from one drive to another except under Windows NT and Windows 95.

You can't rename a file or directory to itself except under Windows NT and Windows 95.

You can rename a directory, but you can't move it except under UNIX.

You can't rename the current directory.

Examples: Name statement

The following example is specific to Windows:

```
' Rename the file WINDOWS\TEST1 to TEST2 and
' move it to the root directory of drive C.
Name "C:\WINDOWS\TEST1" As "C:\TEST2"
```

Now function

Returns the current system date and time as a date/time value.

Syntax
Now

Return value
Now returns the current system date and time as a Variant of DataType 7 (Date/Time).

Usage
A date/time value is an eight-byte floating-point value. The integer part represents a serial day counted from the date January 1, 100 AD. The fractional part represents the time as a fraction of a day, measured from midnight on the preceding day.

You can call the function as either Now or Now().

Language cross-reference
@Now function in formula language

@Today function in formula language

Examples: Now function
The following example is specific to Windows:

```
' Display the current date and time in the Long Date format
' (in Windows 3.1, determined by the system's LongDate
' International setting).

Print Format(Now(), "Long Date")
' Output:
' Tuesday, October 06, 1998
```

Oct function

Returns the octal representation of a number as a string.

Syntax
Oct[$] (*numExpr* **)**

Elements
numExpr

Any numeric expression. If *numExpr* evaluates to a number with a fractional part, LotusScript rounds it to the nearest integer before deriving its octal representation.

Return value

Oct returns a Variant of DataType 8 (String), and Oct$ returns a String.

Return values will only include the numerals 0 to 7, inclusive. The maximum length of the return value is 11 characters.

Usage

If the data type of *numExpr* is not Integer or Long, then LotusScript attempts to convert it to a Long. If it cannot be converted, a type mismatch error occurs.

Examples: Oct function

```
Print Oct$(17)                          ' Prints "21"

' Converts Double argument to Long.
Print Oct$(17.0)                        ' Prints "21"

' Rounds Double argument, then converts to Long.
Print Oct$(17.3)                        ' Prints "21"

' Computes product 16.587, rounds to 17.0, then
' converts to Long.
Print Oct$(17.1 * .97)                  ' Prints "21"
```

On Error statement

Determines how an error will be handled in the current procedure.

Syntax

On Error [*errNumber*] { **GoTo** *label* | **Resume Next** | **GoTo 0** }

Elements

errNumber

Optional. An expression whose value is an Integer error number. If this is omitted, this statement refers to all errors in the current procedure. This value can be any error number that is defined in LotusScript at the time the On Error statement is encountered.

GoTo *label*

Specifies that when the error *errNumber* occurs, execution continues with an error-handling routine that begins at *label*. The error is considered handled.

Resume Next

Specifies that when the error *errNumber* occurs, execution continues with the statement following the statement which caused the error. No error-handling routine is executed. The values of the Err, Erl, and Error functions are not reset. (Note that a Resume statement does reset these values.) The error is considered handled.

GoTo 0

Specifies that when the error *errNumber* occurs, the error should not be handled in the current procedure. If *errNumber* is omitted, no errors are handled in the current procedure.

Usage

The On Error statement is an executable statement. It allows the procedure containing it to change the way LotusScript responds to particular errors. If no On Error statement is used, an error ordinarily causes execution to end. On Error allows a procedure to handle the error and continue execution appropriately.

How does On Error work?

An On Error statement is in effect from the time the statement runs until the procedure that contains it returns control to the calling program or procedure:

- If a procedure includes several On Error *errNumber* statements with the same error number, only the most recently executed one is in effect for that error number.
- The most recently executed On Error statement (with no *errNumber* element) is in effect for that error number if there is no On Error *errNumber* statement for that error number.
- If no On Error statement (without an *errNumber* element) has been executed, then the current procedure doesn't handle the error.

  In this case, LotusScript seeks an On Error statement for the error in the procedure's calling procedure, following the same rules for applying an On Error statement. If the caller doesn't handle the error, LotusScript looks in the caller's caller. If no applicable On Error statement is found by this process, execution ends, and the error message for the error is printed to the output window.

How does the error-handling routine work?

An error-handling routine begins with a labeled statement. The routine ends when LotusScript encounters a Resume, Exit Sub, Exit Property, or Exit Function statement. If an error occurs in the error-handling routine, execution ends.

While the error-handling routine is running, the Err, Erl, and Error functions describe the error being handled. A Resume statement will reset these values.

Where are error numbers and messages defined?

LotusScript specifies a standard set of errors, and corresponding error numbers (as constants), in the file lserr.lss. To define these errors and their numbers, include this file (using %Include) in a script that you compile or load before running any other script. Then these error numbers can be used in On Error statements to control error handling in the session.

Use the Error statement to define new error numbers and messages.

Language cross-reference

@Error function in formula language

@IfError function in formula language

@Failure function in formula language

Examples: On Error statement

In this example, the On Error statement directs LotusScript to continue execution at the next statement after any error that occurs while the function Best is running.

The Call statement generates a division-by-zero error at the attempted division of y by z. Execution resumes at the next statement, the If statement. The current error number is the value of the constant ErrDivisionByZero, which was defined in the file lserr.lss previously included in the script by the %Include statement. Therefore the Print statement is executed. Then the Exit Function statement terminates execution within Best(), without executing further statements within the procedure; and control returns to the caller.

```
%Include "lserr.lss"
Function Best()
   Dim x As Integer, y As Integer, z As Integer

   ' After any error-generating statement, resume
   ' execution with the next statement.
   On Error Resume Next
   ' ...
   y% = 3
   z% = 0
   ' ...
   x% = y% / z%  ' Generates division-by-zero error.
   If Err = ErrDivisionByZero Then
      Print "Attempt to divide by 0. Returning to caller."
      Exit Function
```

```
   End If
   ' ...
End Function
Call Best()
```

On Event statement

Binds an event-handling sub or function to an event associated with a Lotus software object, or breaks an existing binding.

Note The Lotus software application may provide an empty sub or function for each object event, in which case you do not need to use On Event statements. You can enter a script in the appropriate sub or function, and the script automatically executes when the event occurs. For details, see the product documentation.

Syntax

On Event *eventName* **From** *prodObject* { **Call** *handlerName* | **Remove** [*handlerName*] }

Elements

eventName

The name of an event specified in the product class definition.

prodObject

An expression whose value is a reference to a product object. (Events cannot be specified in user-defined class definitions.)

Call

Binds the *handlerName* sub or function to the specified *eventName* from the specified *prodObject*.

handlerName

The name of an event-handling sub or function for the specified *eventName* and *prodObject*. Whenever the specified event happens on the specified object, *handlerName* is called.

Remove

Detaches the *handlerName* sub or function from the object-event pair. If no *handlerName* is specified, this statement detaches all event-handling subs from the object-event pair.

Usage

An event-handling sub or function is defined like any other sub or function, with the restriction that its first parameter must be a reference to the product object that can raise the event. The remaining parameters are defined by the event in the product class, and are used in the handler call.

You can specify multiple event-handling subs or functions for the same event from the same object, using multiple On Event statements. The order of execution of event-handling subs or functions bound to the same event is undefined.

A function is necessary only if the event requires a return value from the handler.

Note Of the three types of objects LotusScript understands (OLE/COM objects, LotusScript product objects, and LotusScript Native objects), only LotusScript product objects can register events.

Examples: On Event statement

This code on a Domino form demonstrates using the Alarm event of the NotesTimer class.

```
REM Globals
Dim elapsedTime As Integer
Dim elapsedTimer As NotesTimer

REM Create a timer with 1-second interval
REM Handler for Alarm event gets call every second
Sub Onload(Source As Notesuidocument)
   Set elapsedTimer = New NotesTimer(1, _
   "Elapsed time since opening document")
   elapsedTime = 0
   On Event Alarm From elapsedTimer _
   Call elapsedTimerHandler
End Sub

REM The handler adds 1 to a global integer
Sub elapsedTimerHandler(Source As NotesTimer)
   elapsedTime = elapsedTime + 1
End Sub

REM This is an action that displays the global integer
Sub Click(Source As Button)
   Messagebox elapsedTime & " seconds",, "Elapsed time"
End Sub
```

On...GoSub statement

Transfers control to one of a list of labels, processes statements until a Return statement is reached, and returns control to the statement immediately following the On...GoSub statement.

Syntax

On *numExpr* **GoSub** *label* [**,** *label, ...*]

Elements

numExpr

A numeric expression whose value determines which of the labels is the target of the transfer of control. The value of *numExpr* must not exceed 255.

label

A label that specifies the location of a series of statements to execute. The last statement in this series is a Return statement.

Usage

The On...GoSub statement, its labels, and the Return statement must all reside in the same procedure.

LotusScript transfers control to the first *label* if *numExpr* is 1, to the second *label* if *numExpr* is 2, and so on. Execution continues from the appropriate label until a Return statement executes. Then control returns to the statement immediately following the On...GoSub statement. If LotusScript encounters a statement (such as Exit or GoTo) that forces an early exit from the procedure before reaching a Return statement, the Return statement is not executed.

LotusScript rounds *numExpr* to the nearest integer before using it to determine the target label. If *numExpr* is 0, or is larger than the number of labels in the list, the On...GoSub statement is ignored and execution continues at the statement that immediately follows it.

LotusScript generates an error if *numExpr* evaluates to a number less than 0 or greater than 255.

Examples: On...GoSub statement

```
' The On...GoSub statement transfers control to Label3 and
' "Went to Label 3" is printed. Then control is returned to
' the statement following the On...GoSub statement, and
' "Successful return" is printed.
Sub Cleanup
   Dim x As Integer
   x% = 3
   On x% GoSub Label1, Label2, Label3
```

```
   Print "Successful return"   ' This prints
   Exit Sub
Label1:
   Print "Error"               ' This does not print
   Return
Label2:
   Print "Error"               ' This does not print
   Return
Label3:
   Print "Went to Label 3"     ' This prints
   Return
End Sub
```

On...GoTo statement

Transfers control to one of a list of labels.

Syntax

On *numExpr* **GoTo** *label* [**,** *label*]...

Elements

numExpr

A numeric expression whose value determines which of the labels is the target of the transfer of control. The value of *numExpr* must not exceed 255.

label

A label that specifies where control is to be transferred.

Usage

On...GoTo can't be used at the module level or to transfer control into or out of a procedure.

LotusScript transfers control to the first *label* if *numExpr* is 1, to the second *label* if *numExpr* is 2, and so on.

LotusScript rounds *numExpr* to the nearest integer before using it to determine the target label. If *numExpr* is 0, or is larger than the number of labels in the list, the On...GoTo statement is ignored and execution continues at the statement following it.

LotusScript generates an error if *numExpr* evaluates to a number greater than 255.

Examples: On...GoTo statement

This example illustrates On...GoTo and On Error.

The user enters a value. If the value is 1, 2, or 3, the On...GoTo statement transfers control to label1, label2, or label3. If the value is another number in the legal range for On...GoTo (the range is 0 to 255), control moves to the next statement. If the user enters a number that is out of range for On...GoTo, or that the CInt function cannot convert to an integer, an error occurs; and LotusScript transfers control to the OutOfRange label, in accordance with the On Error statement.

Depending on the user's entry, the OneTwoThree sub displays an appropriate message. If the entry is valid, an Exit Sub statement exits the Sub. If the entry is not valid, a GoTo statement transfers control to the EnterNum label, and the user is given another chance to make a valid entry.

```
Sub OneTwoThree
   Dim num As Integer
   On Error GoTo OutOfRange
EnterNum:
   num% = CInt(InputBox("Enter 1, 2, or 3"))
   On num% GoTo label1, label2, label3
  ' The user did not enter 1, 2, or 3, but a run-time error
  ' did not occur (the user entered a number in
  ' the range 0 - 255).
   MessageBox "You did not enter a correct value! Try again!"
   GoTo EnterNum
label1:
   MessageBox "You entered 1."
   Exit Sub
label2:
   MessageBox "You entered 2."
   Exit Sub
label3:
   MessageBox "You entered 3."
   Exit Sub
   ' An error condition has occurred.
OutOfRange:
   MessageBox "The value you entered is negative, " _
      & "greater than 255, or not a number. Try again!"
   GoTo EnterNum
End Sub
OneTwoThree ' Call the OneTwoThree sub.
```

Open statement

Opens a file, enabling access to it for reading or writing data.

Syntax

Open *fileName*

[**For** { **Random** | **Input** | **Output** | **Append** | **Binary** }]

[**Access** { **Read** | **Read Write** | **Write** }]

[{ **Shared** | **Lock Read** | **Lock Read Write** | **Lock Write** }]

As [**#**]*fileNumber*

[**Len =** *recLen*]

[**Charset** = *MIMECharsetName*]

This statement must appear on one line, unless you use an underscore (_) for line continuation.

Elements

fileName

A string expression indicating the file to open. *fileName* may include a complete path. If you do not specify a drive and a directory, LotusScript looks for the file in the default directory on the default drive. If you specify a drive but no directory, LotusScript looks for the file in the default directory of the specified drive. On platforms without drive letters, the default directory is used. If you specify a *fileName* that does not exist, LotusScript generates an error if the mode is Input; for all other modes, LotusScript creates the file and opens it.

For *mode*

Optional. Specifies the file's mode; the default is Random.

- **Random**

  Default mode. Designates random access mode; that is, the file is accessible by record number. Use the Get and Put statements to read and write the file. If you omit the Access clause, LotusScript makes three attempts to open the file, using Read Write access, then Write access, and finally Read access. If all three attempts fail, an error is generated.

- **Input**

  Designates sequential input mode. Use the Input and Input # statements to read the file. If the mode conflicts with the Access type, LotusScript generates an error. For example, you can't open a file in Input mode with Write access.

- **Output**

  Designates sequential output mode. Use the Write # and Print # statements to write to the file. If the mode conflicts with the Access type, LotusScript generates an error. For example, you can't open a file in Output mode with Read access.

- **Append**

  Designates sequential output mode, beginning at the current end-of-file. If the mode conflicts with the Access type, LotusScript generates an error. For example, you can't open a file in Append mode with Read access. Unless you use the Seek statement to move to a file position other than the end of the file, the Print # and Write # statements append text to the end of the file.

- **Binary**

  Designates binary file mode. Use the Get and Put statements to read and write the file. If you omit the Access clause, LotusScript makes three attempts to open the file, using Read Write access, then Write access, and finally Read access. If all three attempts fail, an error is generated.

Access *operations*

Optional. Specifies what operations can be performed on the file. An error is generated if the access type conflicts with the file mode specified in the For clause.

- **Read**

  Default access type for Input mode. Only read operations are permitted.

- **Read Write**

  Default access type for Random mode. Both read and write operations are permitted.

- **Write**

  Default access type for Output, Append, and Binary modes. Only write operations are permitted.

Lock *type*

Optional. The default is Shared. Determines how the open file can be shared when accessed over a network by other processes, including processes owned by other users.

Under Windows 3.1, you must run SHARE.EXE to enable the locking feature if you are using MS-DOS version 3.1 or later. Lock is not supported for earlier versions of MS-DOS.

- **Shared**

  Default locking type. No file locking is performed. Any process on any machine on the network can read from or write to the file.

- **Lock Read**

  Prevents other processes from reading the file, although they can write to it. The lock is applied only if read access has not already been granted to another process.

- **Lock Read Write**

  Prevents other processes from reading and writing to the file. The lock is applied only if read or write access has not already been granted to another process. If a lock is already in place, it must be released before opening a file with Lock Read Write.

- **Lock Write**

  Prevents other processes from writing to the file, although they can read from it. The lock is applied only if write access has not already been granted to another process.

fileNumber

An integer expression with a value between 1 and 255, inclusive. This number is associated with the file when you open the file. Other file-manipulation commands use this number to refer to the file.

recLen

Optional. Designates the record length; use an integer expression with a value between 1 and 32767, inclusive.

For a Random file, *recLen* is the record length for the file (all records in a single file must have the same length). The default record length is 128 bytes.

For a sequential (Input, Output, or Append) file, *recLen* is the number of characters to be read from the file into an internal buffer, or assigned to an internal buffer before it is written to the file. This need not correspond to a record size, because the records in a sequential file can vary in size. A larger buffer uses more memory but provides faster file I/O. The default buffer size is 512 bytes.

For a Binary file, *recLen* is ignored.

MIMECharsetName

Optional. Designates the character set to use for sequential file I/O. If no character set is provided, file I/O is done in the platform code page with the following exception:

If a UTF-16 or UTF-8 byte order mark (BOM) is detected at the beginning of the file, the file I/O is done in the code page specified by the BOM.

See MIME charset names for a list of valid MIME charset values.

Usage

If a file is already open in Binary, Random, or Input mode, you can open a copy of the file using a different file number, without closing the open file. If a file is already open in Append or Output mode, you must close it before opening it with a different file number.

LotusScript limits the number of open files to 255. Depending on your operating system environment and the Lotus software you are running, the actual number of files that you can open may be 15 or less. See your product documentation for details.

Examples: Open statement

```
' In this example, LotusScript reads the contents of a
' comma-delimited ASCII file (c:\123w\work\thenames.txt)
' into an array of RecType. RecType is a user-defined
' data type.
' c:\123w\work\thenames.txt consists of the following:
' "Maria Jones", 12345
' "Roman Minsky", 23456
' "Joe Smith", 34567
' "Sal Piccio", 91234

Type RecType
   empId As Double
   employee As String
End Type
Dim arrayOfRecs() As RecType
' A dynamic array that will get sized to
' the number of lines in c:\123w\work\thenames.txt
Dim txt As String
Dim fileNum As Integer
Dim counter As Integer
Dim countRec As Integer
' Get an unused file number so LotusScript can open a file.
fileNum% = FreeFile()
counter% = 0
Open "c:\123w\work\thenames.txt" For Input As fileNum%
```

```
Do While Not EOF(fileNum%)
   ' Read each line of the file.
   Line Input #fileNum%, txt$
   ' Increment the line count.
   counter% = counter% + 1
Loop
' Return the file pointer to the beginning of the file.
Seek fileNum%, 1
' The file has counter number of lines in it, so
' arrayOfRecs() is defined with that number of elements.
ReDim arrayOfRecs(1 To counter%)
' Read the contents of the file into arrayOfRecs.
For countRec% = 1 To counter%
   Input #fileNum%, arrayOfRecs(countRec%).employee$, _
      arrayOfRecs(countRec%).empId#
Next
Close fileNum%
Print arrayOfRecs(2).employee$ & " " arrayOfRecs(2).empId#
' Output:
' Roman Minsky 23456

'Examples using MIMEcharsetname

Open "EBCDIC.TXT" for output  access write _
   as ff CHARSET="ebcdic-us"

Open "UNICODE.TXT" for output access write _
   as ff CHARSET="utf-16"

Open "ASCII.TXT" for output Access write _
   as ff CHARSET="ascii"
```

Option Base statement

Sets the default lower bound for array subscripts to 0 or 1.

Syntax

Option Base *base*

Elements

base

The default lower bound (either 0 or 1) for all dimensions of all arrays in the module in which the Option Base statement occurs.

Usage

Option Base can be specified only once in a module, and only at the module level. If you use Option Base, it must precede all array declarations and all ReDim statements in the module.

The value set by Option Base applies to all arrays in the module that are either declared by Dim statements or redefined by ReDim statements.

If the module does not include an Option Base statement, the default lower bound for all dimensions of all arrays is 0. For example, a one-dimensional array of 10 elements would use subscripts 0-9.

Examples: Option Base statement

```
Option Base 1
' Create a one-dimensional array with 20 elements,
' which can be referred to as sample(1) to sample(20).
Dim sample(20) As Integer
```

Option Compare statement

Specifies the method of string comparison.

Syntax

Option Compare *option1* [**,** *option2*]

Elements

Option can be any of the following:

Binary
Comparison is bit-wise. If Binary is specified, no other option can be specified.

Case or **NoCase**
Comparison is case sensitive (default) or case insensitive. Only one of these options can be specified. The keyword Text is acceptable in place of NoCase.

Pitch or **NoPitch**
Comparison is pitch sensitive (default) or pitch insensitive. Only one of these options can be specified. These options apply to Asian (double byte) characters.

Usage

The Case, NoCase, Pitch, and NoPitch keywords specify string comparison using the character collation sequence determined by the Lotus software that you are using. The Binary keyword specifies string comparison in the platform's collation sequence: the effect is platform sort-order, case-sensitive, pitch-sensitive comparison.

Option Compare can be specified more than once per module, but the options cannot conflict. Option Compare can appear anywhere at module level. Option Compare applies to all string comparisons in the module. If you omit the Option Compare statement, the default method of string comparison is the same as Option Compare Case and Option Compare Pitch.

In certain functions such as InStr and StrCompare, the case and pitch sensitivity established by Option Compare or by default can be overridden by case-sensitivity and pitch-sensitivity arguments.

Examples: Option Compare statement

Example 1

The following example is specific to Windows. In this example, the first call to function StrCompare uses the default (case-sensitive) setting without the optional argument that specifies a comparison method. In case-insensitive comparison, "A" equals "a", so StrCompare returns FALSE (0).

The second call to the function StrCompare specifies case-sensitive comparison in the country/language collation order, overriding the default established by Option Compare NoCase. In this comparison, "A" occurs earlier in the sort order than "a", so StrCompare returns TRUE (-1).

```
' The following results are for LotusScript in English,
' running on Windows 3.1.

Option Compare NoCase

' No method specified in StrCompare; use NoCase.
Print StrCompare("A", "a")   ' Output: 0, these two strings
are equal.

' Use case-sensitive comparison
' (in country/language collation order).
Print StrCompare("A", "a", 0)  ' Output: 1, string1 greater
than
                                        ' string 2.  Strings are
not equal.
```

Example 2

In this example, no Option Compare statement appears in the module, so the list tags "a" and "A" are different tags, because case-sensitive comparison in the country/language collation order is the default. Thus, the assignments to Loft("a") and Loft("A") refer to two different list elements. Within the ForAll statement, the ListTag function retrieves a list tag; and the Print statement prints it on a separate line.

```
Dim loft List As Integer
loft%("a") = 2
```

```
loft%("A") = 17
ForAll i In loft%
   Print ListTag(i)    ' Output: "a" and "A"
End ForAll
```

Example 3

In this example, the Option Compare NoCase statement specifies case-insensitive comparison in the country or region/language collation order as the default method for string comparison, so the list tags “a” and “A” are the same tag. Thus, the assignments to loft(“a”) and loft(“A”) refer to the same list element. There is only one list tag for the ListTag function to retrieve and print.

```
Option Compare NoCase
Dim loft List As Integer
loft%("a") = 2
loft%("A") = 17
ForAll i In loft%
   Print ListTag(i)    ' Output: "A"
End ForAll
```

Example 4

In this example, the Option Compare Binary statement specifies bit-wise (platform sort-order, case-sensitive) comparison as the default method for string comparison, so the list tags “a” and “A” are different tags. Thus, the assignments to loft(“a”) and loft(“A”) refer to different list elements.

```
Option Compare Binary
Dim loft List As Integer
loft%("a") = 2
loft%("A") = 17
ForAll i In loft%
   Print ListTag(i)    ' Output: "a" and "A"
End ForAll
```

Option Declare statement

Disallows implicit declaration of variables.

Syntax
Option Declare

Explicit is acceptable in place of Declare.

Usage
Option Declare can be specified only once in a module, and only at the module level.

If the Option Declare statement appears in a module, then undeclared variables will generate syntax errors. When Option Declare is in effect, you must use the Dim statement to declare variables, except for arrays. You can still define an array implicitly using the ReDim statement.

Option Declare must be used before any variables are implicitly declared.

Examples: Option Declare statement
```
' Turn off implicit declaration of variables.
Option Declare
Dim y As Integer
y% = 10            ' No error
x = 20             ' Compiler error (x has not been declared)
ReDim simAry(2, 2) ' No error
```

Option Public statement

Specifies that module-level explicit declarations are Public by default.

Syntax
Option Public

Usage
Option Public can be specified only once in a module, and only at the module level. It must appear before any declarations in the module.

Option Public applies to module-level declarations for any variable, constant, procedure, user-defined data type, user-defined class, or external C function. It does not apply to label definitions, ForAll reference variables, or any implicitly declared variables.

The IDE automatically puts an Option Public statement in (Globals) (Options), so all (Globals) declarations are public by default. If you delete the Option Public statement, you must explicitly specify the Public keyword to make (Globals) declarations public.

If a variable of a user-defined data type or an object reference variable is Public, the data type or the class to which it refers cannot be Private.

Use the Private keyword in a declaration to override Option Public for that declaration.

Examples: Option Public statement

```
' In this example, the Private keyword overrides
' Option Public in the declaration of the
' variables x, y, and z.
Option Public
Private x, y, z      ' x, y, and z are Private variables.
Dim i As Integer     ' i is Public.
```

Print statement

Prints data to the screen.

Syntax

Print [*exprList*]

Elements

exprList

A list of expressions separated by semicolons, spaces, or commas.

Usage

If *exprList* is omitted, Print prints a blank line.

Use the Spc and Tab functions to insert spaces and tabs between data items.

The Print statement adds a newline character to the end of *exprList* (to force a carriage return), unless *exprList* ends with a semicolon or a comma.

LotusScript inserts "chr(10)" to represent the newline character in any multiline string (for example, a string that you type in using vertical bars or braces). If you use Print to print the string, this newline character will be translated into the platform-specific newline character(s).

Note Newline does not mean either chr(10) or chr(13) on all platforms. Newline is the character or sequence of characters that is used to mark the end of a line. This may be chr(10), or chr(13), but it may also be something else, because the actual value of newline depends on the platform.

The following table shows how the Print statement handles data items specified in *exprList*.

Data item	*Print statement behavior*
variable	Prints the value of the variable.
string	Prints the string.
date/time value	Prints the date as a string in the operating system Short Date and Time format. If either the date part or the time part is missing from the value, only the supplied part is printed.
Variant with the value EMPTY	Prints an empty string ("").
Variant with the value Null	Prints the string "#NULL#".

The following table shows the effect of semicolons and commas in the Print statement.

Punctuation character	*Print statement behavior*
Semicolon or space in *exprList*	The next data item is printed with no spaces between it and the previous data item.
Semicolon at end of *exprList*	The next Print statement continues printing on the same line, with no spaces or carriage returns inserted.
Comma in *exprList*	The next data item is printed beginning at the next tab stop. (Tab stops are at every 14 characters.)
Comma at end of *exprList*	The next Print statement continues printing on the same line, beginning at the next tab stop. (Tab stops are at every 14 characters.)

If you are in Lotus Notes, note that the Print statement writes to the following:

- The status bar when executing on a Notes client in non-debug mode.
- The status bar and output window when executing on a Notes client in debug mode.
- NOTES.LOG when executing on a Domino server.

If the request is from the Web, Print will be re-directed to the source. Print can be used to dynamically generate a Web page via QueryOnEvent.

Examples: Print statement

```
Dim a As Integer, b As Integer, c As Integer
a% = 5
b% = 10
c% = 15
```

```
Print a%, b%, c%        ' Prints 5  10  15
' LotusScript prints the values of a, b, and c,
' separating them with tabs and ending the line
' with a newline character.
```

Print # statement

Prints data to a sequential text file.

Syntax

Print #*fileNumber* **,** [*exprList*]

Elements

fileNumber

The file number assigned to the file when it was opened. Note that the pound sign (#), the file number, and the comma are all required.

exprList

Optional. A list of string and/or numeric expressions separated by semicolons, spaces, or commas. If you omit *exprList*, Print # prints a blank line.

Usage

Use Print # only on files opened in Output or Append mode. Unlike the Write # statement, the Print # statement does not separate the printed data items with formatting characters such as commas and quotation marks.

Use the Spc and Tab functions to insert spaces and tabs between data items.

If you set a width for the file using the Width statement, then the following occurs:

- A comma moves the next print position to the next tab stop. If this moves the print position past the defined width, the next data item is printed at the beginning of the next line.
- If the current print position is not at the beginning of a line and printing the next item would print beyond the defined width, the data item is printed at the beginning of the next line.
- If the item is larger than the defined width, it's printed anyway because Print # never truncates items. However, the line is terminated with a newline character to ensure that the next data item is printed on a new line.

Note Newline does not mean either chr(10) or chr(13) on all platforms. Newline is the character or sequence of characters that is used to mark the end of a line. This may be chr(10), or chr(13), but it may also be something else, because the actual value of newline depends on the platform.

The preceding statements about the effect of the Width statement apply for a width of 0, as well as any positive width.

The following table shows how the Print # statement handles data items specified in *exprList*.

Data item	*Print # statement behavior*
variable	Prints the value of the variable.
string	Prints the string.
date/time value	Prints the date as a string in the operating system Short Date and Time format. If either the date part or the time part is missing from the value, only the supplied part is printed.
Variant with the value EMPTY	Prints nothing to the file for the data item.
Variant with the value Null	Prints the string "NULL" to the file.

The following table shows the effect of semicolons and commas in the Print # statement.

Punctuation character	*Print statement behavior*
Semicolon or space in *exprList*	The next data item is printed with no spaces between it and the previous data item.
Comma in *exprList*	The next data item is printed beginning at the next tab stop. (Tab stops are at every 14 characters.)

Examples: Print # statement

```
Dim nVar As Variant, eVar As Variant
nVar = NULL

Dim fileNum As Integer
fileNum% = FreeFile()
Open "printext.txt" For Output As fileNum%

' Print two lines to the file and close it.
' First line: two String values, with no separation between.
Print #fileNum%, "First line, " ; "with two String items"
' Second line: NULL value, EMPTY value, Integer variable
' value, and String value, separated on the line by tabs.
Print #fileNum%, nVar, eVar, fileNum%, "at next tab"
Close fileNum%
```

```
' Open the file, print it, and close the file.
Dim text As String
Open "printext.txt" For Input As fileNum%
Do Until EOF(fileNum%)
   ' Read and print to console, one line at a time.
   Line Input #fileNum%, text$
   Print text$
Loop
Close fileNum%
' Output:
' First line, with two String items
' NULL                      1            at next tab
```

Property Get/Set statements

Define a property. A property is a named pair of Get and Set procedures that can be used as if they were a single variable.

Syntax

[Static] [Public | **Private] Property** { **Get** | **Set** } *propertyName* **[(** [*paramList*] **)]** [**As** *type*]

[*statements*]

End Property

Elements

Static

Optional. Specifies that the values of a Static property's variables are saved between calls to the property.

Public | **Private**

Optional. Public specifies that the property is visible outside the scope (module or class) where the property is defined, as long as this module is loaded. Private specifies that the property is visible only within the current scope.

A property in module scope is Private by default. A property in class scope is Public by default.

The Property Get and Property Set definitions for a property must use the same Public or Private setting.

Get | **Set**

Specifies which operation the procedure performs. A Property Get procedure retrieves the value of the property. A Property Set procedure assigns a value to the property.

propertyName

The name of the property. This name can have a data type suffix character appended to declare the data type of the value passed to and returned by the property.

paramList

Optional. A comma-separated list of declarations indicating the parameters to be passed to this property in Get and Set operations. The Get and Set operations must have the same number of arguments.

The syntax for each parameter declaration is:

[**ByVal**] *parameter* [**( )** | **List**] [**As** *type*]

ByVal means that *parameter* is passed by value: that is, the value assigned to *parameter* is a local copy of a value in memory, rather than a pointer to that value.

*parameter***()** is an array variable. *parameter* **List** identifies *parameter* as a list variable. Otherwise, *parameter* can be a variable of any of the other data types that LotusScript supports.

As *dataType* specifies the variable's data type. You can omit this clause and append a data type suffix character to *parameter* to declare the variable as one of the scalar data types. If you omit this clause and *parameter* has no data type suffix character appended (and isn't covered by an existing Def*type* statement), its data type is Variant.

Enclose the entire list of parameter declarations in parentheses.

type

Optional. The data type of values passed to and returned by the property.

type can be any of the scalar data types, a Variant, or a class name.

If **As** *Type* is not specified, the property name's data type suffix character determines the value's type. Do not specify both a *type* and a data type suffix character, as LotusScript treats that as an error.

If no *type* is specified and the property name has no data type suffix character appended, the property's value is either of data type Variant or of the data type specified by a Def*type* statement.

The *types* in the Property Get and Property Set definitions must be the same.

statements

Statements to retrieve or assign a property value.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

A property usually consists of two procedures with the same name: a Property Get and a Property Set. However, you are not required to provide both.

A property member of a class cannot be declared Static. That is, a Property Get or Property Set statement within a class definition cannot begin with Static.

Using Property Get

A Property Get procedure is like a function. For example:

```
' These statements assign the value of saveInt to x
Dim saveInt As Integer
Property Get pInt As Integer
   pInt% = saveInt%
End Property
x = pInt%
```

Or:

```
' These statements assign the value of saveInt plus
' increment to x
Dim saveInt As Integer
Property Get pInt (increment As Integer) As Integer
   pInt% = saveInt% + increment%
End Property
x = pInt%(1%)
```

Using Property Set

A Property Set procedure is the reverse of a Property Get procedure. On entry into a Property Set procedure, an implicitly declared variable whose name and data type are the same as those of the Property Set procedure contains a value to be used inside the Property Set procedure. Inside the Property Set procedure, use the value of the variable instead of assigning a value to it.

Call a Property Set procedure by using its name on the left side of an assignment statement. The value on the right side of the statement is used by the Property Set procedure. For example:

```
' These statements assign the value of x to SaveInt
Dim SaveInt As Integer
Property Set pInt As Integer
   saveInt% = pInt%
```

```
End Property
pInt% = x
```

Or:

```
' These statements assign the value of x + increment
' to SaveInt
Dim SaveInt As Integer
Property Set pInt (increment As Integer) As Integer
   saveInt% = pInt% + increment%
End Property
pInt%(1%) = x
```

Referencing a property that returns an array, list, or collection

If a Get operation returns an array, list, or collection, a reference to the property can contain subscripts according to the following rules:

- If the property has parameters, the first parenthesized list following the reference must be the argument list. A second parenthesized list is treated as a subscript list. For example, p1(1,2)(3) is a reference to a property p1 that has two parameters and returns a container.
- If the property has no parameters and the return type is a variant or collection object, a single parenthesized list following the reference is treated as a subscript list. For example, p1(1) is a reference to a property p1 that either contains one parameter or contains no parameters but is a container.
- If the property has no parameters and the return type is not a variant or collection object, any parenthesized list following the reference is an error, except that a single empty list is allowed. For example, p1() is a reference to a property p1 that contains no parameters and may or may not be a container; if p1 is a container, the reference is to the entire container.

In a Set operation, the property reference cannot be subscripted. A parenthesized list following the reference must be the argument list. For example, p1(1) is a reference to a property p1 with one parameter; p1(1,2)(3) or p1()(3) is illegal in a Set operation.

Passing a property to a function

A LotusScript property (a property defined by Property Get or Property Set) can be passed to a function by value only, not by reference.

Examples: Property Get/Set statements

```
' This example illustrates basic operations with a property.
' The counter is a property; it receives a starting value.
' Each time the property is used, it returns a value that is
' 1 greater than the previous value, until a new starting
' value is set. In this example, counter is set to 100.
```

```
' Then the property is used to print 101 and again
' to print 102.

' A variable to store values between uses of the property
Dim count As Integer

Property Get counter As Integer
   count% = count% + 1    ' Add 1 to the previous value.
   counter% = count%      ' Return the value.
End Property

Property Set counter As Integer
   count% = counter%      ' Assign the value to count.
End Property
counter% = 100

' Each time the property is used, it increments count
' by 1 and returns count's value, so this prints 101.
Print counter%

' Prints 102
Print counter%
```

Put statement

Writes data from a variable to a binary file or a random file.

Syntax

Put [**#**] *fileNumber* **,** [*recordNumber*] **,** *variableName*

Elements

fileNumber

The file number assigned to the file when it was opened with the Open statement. Note that the pound sign (#), *fileNumber,* and *variableName* are all required.

recordNumber

Optional. The file position (the byte position in a binary file, or the record number in a random file) where data is written. If you omit the *recordNumber*, data is written starting at the current file position.

variableName

The variable holding the data to be written. *variableName* cannot be an array; however, a fixed-length array defined within a data type is allowed (this array could even contain other arrays as elements).

Usage

The first byte or record in a file is always file position 1. After each write operation, the file position is advanced:

- For a binary file, by the size of the variable
- For a random file, by the size of a record

If *variableName* is shorter than the length of a record in the file, Put does not overwrite or delete any data that may already be stored in the remainder of that record.

The following table shows how the Put statement behaves for different data types.

variableName data type	*Put statement's behavior*
Variant	The Put statement writes the DataType as the first two bytes before the value itself. If the DataType is EMPTY or NULL, the Put statement writes no more data. If the DataType is numeric, the Put statement writes the number of bytes of data appropriate for that DataType: Byte: 1 byte Boolean: 2 bytes Integer: 2 bytes Long: 4 bytes Single: 4 bytes Double: 8 bytes Currency: 8 bytes Date/time: 8 bytes
Fixed-length String	The Put statement writes the specified number of characters. For example, if a variable is declared as String * 10, then exactly 10 characters are written.

continued

variableName data type	*Put statement's behavior*
Variable-length String	The Put statement behaves differently, depending on the type of file you're using. *Random files*: The first two bytes written indicate the length of the string. Then the Put statement writes the number of characters specified by that length. If *variableName* is not initialized, the Put statement writes a string of length 0. If *variableName* is longer than a record, LotusScript generates the "Bad record length" error. If *variableName* is shorter than a record, the remainder of the record is not cleared. *Binary files*: The number of bytes written to the file is equal to the length of the string currently stored in *variableName*. If *variableName* is not initialized, no data is written to the file. Note that in binary files, data is written without regard to record length.
User-defined data type	The Put statement writes the sum of the bytes required to write all members of the used-defined data type, which cannot contain a dynamic array, a list, or an object.

When Put writes out String data, the characters are always written in the Unicode character set.

Note Even though strings in LotusScript 4 can be longer than 64K, there are still restrictions with the length of the string you can read or write using the GET and PUT statements. The only combination of filetypes that will work with long strings is with a binary file and a variable-length string. Fixed length strings, strings in variants, and random files will not work with strings greater than 64K in length because they have a two-byte header which contains the length of the string. Two bytes cannot represent more than 64K.

Examples: Put statement

```
Type PersonRecord
   empNumber As Integer
   empName As String * 20
End Type

Dim fileNum As Integer
Dim fileName As String
Dim rec As PersonRecord
fileNum% = FreeFile()
fileName$ = "data.txt"
```

```
' First, open a random file with a record length equal to
' the size of the records to be stored.
Open fileName$ For Random As fileNum% Len = Len(rec)

rec.empNumber% = 123
rec.empName$ = "John Smith"
Put #fileNum%, 1, rec       ' Write this record at position 1.
rec.empNumber% = 456
rec.empName$ = "Jane Doe"
Put #fileNum%, 2, rec       ' Write this record at position 2.
rec.empNumber% = 789
rec.empName$ = "Jack Jones"
Put #fileNum%, , rec        ' Write at current position (3).

Seek fileNum%, 1            ' Rewind file to beginning.
Do While Not EOF(fileNum%)
   ' Get a record, print it out.
   ' Get advances the file position to the next
   ' record automatically.
   Get #fileNum%, , rec
   Print rec.empNumber%, rec.empName$
Loop
' Output:
' 123         John Smith
' 456         Jane Doe
' 789         Jack Jones
Close fileNum%              ' Close the file.
```

Randomize statement

Seeds (initializes) the random number generator.

Syntax

Randomize [*numExpr*]

Elements

numExpr

Any numeric expression. If you omit *numExpr*, Randomize uses the return value from Timer.

Usage

Use Randomize to seed the random number generator before calling Rnd to generate a number.

If you use Randomize with *numExpr* and then repeatedly call Rnd with no arguments, LotusScript returns the same sequence of random numbers

every time you run the script. To generate a different sequence of random numbers each time you run the script, do one of the following:

- Use a variable *numExpr* to make sure that Randomize receives a different seed value every time the script is executed.
- Use Randomize with no *numExpr.* This seeds the random number generator with the return value from Timer.

The particular sequence of random numbers generated from a given seed depends on the platform where you are running LotusScript.

Examples: Randomize statement

Example 1

```
Randomize 17    ' Use 17 to seed the random number generator.
Print Rnd(); Rnd(); Rnd(); Rnd(); Rnd()
' Output:
' .9698573  .8850777  .8703259  .1019439  .7683496
' If you rerun this script (on the same platform), LotusScript
' generates the same sequence of random numbers,
' because the same seed is used.
```

Example 2

```
Randomize               ' Don't provide any seed.
Print Rnd(); Rnd(); Rnd(); Rnd(); Rnd()
                        ' Prints a series of random numbers.
' If you rerun this script, LotusScript produces a different
' sequence of random numbers, because Randomize is called
' with no argument.
```

ReDim statement

Declares a dynamic array and allocates storage for it, or resizes an existing dynamic array.

Syntax

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

[**,** *arrayName* **(** *bounds* **)** [**As** *type*]] ...

Elements

Preserve

Optional. If you've already declared *arrayName,* LotusScript preserves the values currently assigned to it. If you omit Preserve, LotusScript initializes all elements of the array, depending on the data type of the array variable.

Data type of array variable	*Initial value of array element*
Boolean, Byte, Integer, Long, Single, Double, or Currency	0
Fixed-length String	A string of the specified length, filled with the Null character (Chr(0))
Variable-length String	The empty string (“”)
Variant	EMPTY
Class	NOTHING
User-defined data type	The initial value of each element’s own data type

arrayName

The name of an array to be declared or resized. The *arrayName* must designate an array; it cannot be a Variant variable containing an array.

bounds

A comma-separated list of dimension bounds for *arrayName.* Each set of dimension bounds has the following form:

[*lowerBound* **To**] *upperBound*

The *lowerBound* is the minimum subscript allowed for the dimension, and *upperBound* is the maximum. If you don’t specify a *lowerBound,* 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 bounds must fall in the range -32768 to 32767, inclusive.

type

Optional. A valid LotusScript data type, user-defined type, or class that specifies the data type of *arrayName*.

You cannot change the data type of an existing array. If *arrayName* was declared and *type* is specified in the current ReDim statement, *type* must match the original data type of *arrayName*.

Usage

A ReDim statement allocates storage for a dynamic array. You can resize the array with additional ReDim statements as often as you want. Each time you resize the array, LotusScript reallocates the storage for it.

Unlike a Dim statement, ReDim cannot specify an array as Private, Public, or Static. To specify a dynamic array with one of these characteristics, declare it first in a Dim statement. If you declare a dynamic array with a Dim statement, LotusScript doesn’t allocate storage for the array elements. You can’t actually use the array in your script until you allocate storage with ReDim.

Arrays can have up to 8 dimensions. The first ReDim statement for an array sets the number of dimensions for the array. Subsequent ReDim statements for the array can change the upper and lower bounds for each dimension, but not the number of dimensions.

If Preserve is specified, you can change only the upper bound of the last array dimension. Attempting to change any other bound results in an error.

Do not use ReDim on a fixed array (an array already declared and allocated by a Dim statement).

If you're using ForAll on a container variable that is an array of arrays, do not ReDim the reference variable (this generates the "Illegal ReDim" error).

Examples: ReDim statement

Example 1

```
' The array x has not been previously declared,
' so ReDim automatically assigns it the data type Variant.
ReDim x(5)
Print DataType(x(1))                    ' Prints 0.

' The Dim statement declares array y with the
' data type String.
Dim y() As String

' The ReDim statement can't change the data type of an
' existing  array. If you specify a data type for array y in
' the ReDim statement, it must be String.

ReDim y(5) As String
Print DataType(y$(1))                   ' Prints 8.
```

Example 2

```
Option Base 1
' Declare a two-dimensional dynamic array, of Variant type.
ReDim markMar(2, 2)

' Assign a value to each element.
markMar(1, 1) = 1
markMar(2, 1) = 2
markMar(1, 2) = 3
markMar(2, 2) = 4

' Change the upper bound of the last dimension of markMar
' from 2 to 3, preserving the values already stored in
' existing elements of markMar.
ReDim Preserve markMar(2,3)

' Assign values to the additional elements of markMar.
markMar(1, 3) = 5
markMar(2, 3) = 6
```

Rem statement

Indicates a one-line comment in a script.

Syntax
Rem *text*

Elements
text
A one-line comment that LotusScript ignores.

Usage
The Rem statement indicates a comment or "remark" in the script.

The Rem statement need not be the first statement on a line, but it is the last: the LotusScript compiler ignores all text from the Rem keyword to the end of the current line. A line continuation character (an underscore) does not continue a Rem statement.

The apostophe (') has the same effect as the Rem keyword and can appear anywhere on a line without needing a colon (:) to separate the statements. As with Rem, LotusScript ignores everything after the apostrophe.

Language cross-reference
REM keyword in formula language

Examples: Rem statement

Example 1
```
Rem  This is a comment in the script.
'This is also a comment in the script.
```

Example 2
```
x = 5 : Rem The colon is required to separate statements.
x = 5   ' No colon is required before a single quote.
```

%Rem directive

Indicates one or more comment lines in a script.

Syntax
%Rem

text

%End Rem

Elements

text

One or more lines of text that LotusScript ignores.

Usage

The compiler ignores all text between %Rem and %End Rem, including text on the same line.

%Rem and %End Rem must each be the first text on a line (they may be preceded on the line by spaces or tabs). Each must be followed by one or more spaces, tabs, or newline characters before any more text appears.

%Rem...%End Rem blocks cannot be nested.

Note: For compatibility with older versions of the language, LotusScript Release 3 accepts the directive %EndRem (with no space) in place of %End Rem.

Language cross-reference

REM keyword in formula language

Examples: %Rem directive

Example 1

```
' The compiler ignores the lines of text between %Rem and
' %End Rem, and the text on the line beginning %Rem.
' It also ignores the line containing the Rem statement.
%Rem   Note that the compiler ignores all text on this line.
       What follows is ignored by the compiler. It can
       contain comments or non-working statements.
   Check( This, for example, would have been a syntax error.)
%End Rem   This text is ignored as well.
Rem Normal parsing and compilation continues from here.
```

Example 2

```
' %Rem blocks cannot be nested, so the second %Rem
' directive is illegal in the following.
%Rem
Comment line 1
Comment line 2
...
%Rem            ' Error
Comment line
...
%End Rem
%End Rem
```

Replace function

Replaces specific words or phrases in a string with new words or phrases that you specify.

Syntax

Replace(*sourceArray* **as Variant,** *findArray* **as Variant,** *replacementArray* **as Variant[,** *start* **as Integer[,** *count* **as Integer[,** *compMethod* **as Integer]]]) as Variant**

Elements

sourceArray

Array of type String containing the strings to be modified

replaceArray

Array of type String containing the words or phrases to be replaced

replacementArray

Array of type String containing the replacement words or phrases

start

optional Integer specifying the character position to start at in each String

count

optional Integer specifying the maximum number of replacements to make.

compMethod

Optional Integer specifying the type of comparison to use when searching for the delimiter, if the elements are strings.

Number	*Comparison Mode*
0	case sensitive, pitch sensitive
1	case insensitive, pitch sensitive
4	case sensitive, pitch insensitive
5	case insensitive, pitch insensitive

If you omit *compMethod*, the default comparison mode is the mode set by the Option Compare statement for this module. If there is no statement for the module, the default is case sensitive and pitch sensitive.

Return value

Replace returns an Array of type String that contains sourceArray, where any values in replaceArray have been replaced by the corresponding values in replacementArray.

Usage

Replace searches the String in sourceArray for the String in replaceArray. If a match is found, the substring is replaced with a corresponding substring from replacementArray. Each String in replaceArray is scanned against each String in sourceArray as modified by prior substitutions. Replace is case sensitive.

If no matches are found, then a copy of sourceArray is returned.

If more strings are specified in replaceArray than in replacementArray, the extra strings in replaceArray are replaced with the last string in replacementArray. Extra strings in replacementArray are ignored.

For example:

```
sourceArray = ["first second third"]
replaceArray = ["first"]["second"]["1"]["third"]["2"]["3"]
replacementArray = ["1"]["2"]["a"]["3"]["b"]["c"]
```

would return: [“a b c”]

First, Replace substitutes “1” for “first” (the first String in replacementArray replaces the first string in replaceArray):

```
["1 second third"]
```

Then Replace substitutes “2” for “second”:

```
["1 2 third"]
```

Then “a” for “1” (since the first replacement was “1” for “first”):

```
["a 2 third"]
```

Then “3” for “third”:

```
["a 2 3"]
```

“b” for “2”:

```
["a b 3"]
```

And finally, “c” for “3”:

```
["a b c"]
```

If sourceArray, replaceArray, or replacementArray is not either a String, or an Array of type String, then a run-time type mismatch error is thrown.

Language cross-reference

@Replace function in formula language

EditFind @command in formula language

Examples: Replace function

```
Sub Initialize

   Dim array1(2) As string
   Dim array2(2) As string
   Dim array3(2) As string
   Dim ret As Variant

   array1(0) = "original0"
   array1(1) = "ThisShouldNotBeReplaced1"
   array1(2) = "original2"

   array2(0) = "original"
   array2(1) = "ShouldNotFindThis"
   array2(2) = "once"

   array3(0) = "replaced--once--"
   array3(1) = "this should be skipped"
   array3(2) = "twice"

   ret = replace(array1, array2, array3)
   for x = 0 to 2
      Print ret(x)
   Next

End Sub

'OUTPUT
'replaced--twice--0
'ThisShouldNotBeReplaced1
'replaced--twice--2
```

Reset statement

Closes all open files, copying the data from each file to disk.

Syntax

Reset

Usage

Before closing the open files, Reset writes all internally buffered data to the files.

Examples: Reset statement

```
' All open files are closed and the contents of the operating
' system buffer are written to disk.
Reset
```

Resume statement

Directs LotusScript to resume script execution at a particular statement in a script, after an error has occurred.

Syntax

Resume [**0** | **Next** | *label*]

Elements

0

Resumes execution at the statement that caused the current error.

Next

Resumes execution at the statement following the statement that caused the current error.

label

Resumes execution at the specified label.

Usage

Use the Resume statement only in error-handling routines; once LotusScript executes the Resume statement, the error is considered handled.

Resume continues execution within the procedure where it resides. If the error occurred in a procedure called by the current procedure, and the called procedure didn't handle the error, then Resume assumes that the statement calling that procedure caused the error:

- Resume [0] directs LotusScript to execute again the procedure-calling statement that produced the error.

  Note that this may result in an infinite loop, where in every iteration, the procedure generates the error and then is called again.

- Resume Next directs LotusScript to resume execution at the statement following the procedure call.

The Resume statement resets the values of the Err, Erl, and Error functions.

Examples: Resume statement

```
Sub ResumeSub()
   On Error GoTo ErrHandler
   ' ...
   Error 1                  ' Intentionally raise an error.
   Error 10
   Error 100
   ' ...
   Exit Sub

ErrHandler:                 ' Error-handling routine
```

```
   Print "Error " & Err & " at line number" &Erl
   Resume Next              ' Resume the procedure.
End Sub
' The error-handling routine prints information about the
' current error. Then LotusScript resumes execution of the
' script at the statement following the statement that caused
' the current error.
```

Return statement

Transfers control to the statement following a GoSub or On...GoSub statement.

Syntax

Return

Usage

The GoSub and On...GoSub statements transfer control to a labeled statement within a procedure. Execution continues from this statement until a Return statement is encountered. LotusScript then transfers control to the first statement following the GoSub or On...GoSub statement. While executing the procedure, LotusScript can encounter a statement, such as Exit or GoTo, that forces an early exit from the procedure; in this case, the Return is not executed.

The GoSub or On...GoSub statement, its labels, and the Return statement must reside in the same procedure.

Examples: Return statement

```
' In response to user input, LotusScript transfers control to
' one of three labels, constructs an appropriate message,
' and continues execution at the statement following
' the GoSub.
Sub GetName
   Dim yourName As String, Message As String
   yourName$ = InputBox$("What is your name?")
   If yourName$ = "" Then        ' The user enters nothing.
      GoSub EmptyString
   ' A case-insensitive comparison
   ElseIf LCase(yourName$) = "john doe" Then
      GoSub JohnDoe
   Else
      Message$ = "Thanks, " & yourName$ _
         & ", for letting us know who you are."
   End If
   ' The Return statements return control to the next line.
```

```
   MessageBox Message$
   Exit Sub

EmptyString:
   yourName$ = "John Doe"
   Message$ = "Okay! As far as we're concerned, " _
     & "your name is " & yourName$ & ", and you're on _
the run!"
   Return

JohnDoe:
   Message$ = "We're on your trail, " & yourName$ _
     & ". We know you are wanted dead or alive!"
   Return
End Sub
GetName                          ' Call the GetName sub.
```

Right function

Extracts a specified number of the rightmost characters in a string.

Syntax

Right[$] (*expr* **,** *n* **)**

Elements

expr

Any numeric or String expression for Right; and any Variant or String expression for Right$. If the expression is numeric, it is first converted to a string.

n

The number of characters to be returned.

Return value

Right returns a Variant of DataType 8 (String), and Right$ returns a String.

If *n* is 0, Right returns the empty string (""); if *n* is greater than the number of characters in *expr*, Right returns the entire string.

Right(NULL,1) returns NULL. Right$(NULL,1) returns an error.

Usage

LotusScript Release 3 and later represent characters with two bytes instead of one, so Lotus no longer recommends using the RightB function to work with bytes.

Language cross-reference
@Right function in formula language

Examples: Right function
```
Dim subString As String
subString$ = Right$("ABCDEF", 3)
Print subString$                ' Prints "DEF"
```

RightB function

LotusScript Release 3 and later use Unicode, a character set encoding scheme that represents each character as bytes. This means that a character can be accompanied by leading or trailing zeroes, so Lotus no longer recommends using RightB to work with bytes.

Instead, use the Right function for right character set extractions.

RightBP function

Extracts a specified number of the rightmost bytes in a string using the platform-specified character set.

Syntax
RightBP[**$**] **(** *expr* **,** *n* **)**

Elements
expr

Any numeric or String expression for RightBP; and any Variant or String expression for RightBP$. If *expr* is numeric, LotusScript converts it to a string before performing the extraction.

n

The number of bytes to be returned using the platform-specified character set.

Return value
RightBP returns a Variant of DataType 8 (a String), and RightBP$ returns a String.

If *n* is 0, the function returns the empty string (“”). If *n* is greater than the length (in bytes) of *expr*, the function returns the entire string.

RightBP(NULL) returns NULL. RightBP$(NULL) is an error.

If a double-byte character is divided, the character is not included.

Language cross-reference

@Right function in formula language

Examples: RightBP function

```
' The value "BC" or other value depending on platform
' is assigned to the variable subString.

Dim subString As String
subString = RightBP$("ABC", 2)
Print subString$  ' Output:  "BC"
```

RightC function

Extracts the rightmost *n* columns from a string for column-based writing systems, such as Thai and Vietnamese.

Syntax

RightC[$] (*StringExpr*, *n***)**

Elements

StringExpr

A String expression containing character columns.

n

The number of columns to be returned using the platform-specified character set.

Return value

RightC returns a Variant containing the columns specified by *n*. RightC$ returns a String.

Usage

If *n* is 0, the function returns the empty string (“”). If *n* is greater than the length (in columns) of *StringExpr*, the function returns the entire string.

RightC supports the Thai and Vietnamese languages.

Examples: RightC function

```
'Extracts the rightmost 6 Thai columns from a string.

RightC("XXXxxxXXXxxxXxXxxXxxX", 6)

'Returns "xxXxxX"
```

RmDir statement

Removes a directory from the file system.

Syntax
RmDir *path*

Elements
path
A String expression specifying the path of the directory you want to remove.

Usage
The maximum length of *path* depends on the platform you are using.

If the directory named by *path* is not empty, RmDir generates an error.

Examples: RmDir statement
```
' Remove directory c:\test from the file system.
RmDir "c:\test"
```

Rnd function

Generates a random number greater than 0 and less than 1.

Syntax
Rnd [**(** *numExpr* **)**]

Elements
numExpr
Any numeric expression.

Return value
The return value is a number of data type Single. The following table shows how Rnd behaves, depending on the sign of *numExpr*.

Sign of numExpr	*Rnd behavior*
Positive	Returns the next random number in the sequence of random numbers generated from the value that most recently seeded the random number generator.
Zero (0)	Returns the random number most recently generated.
Negative	The random number generator is seeded again with *numExpr*. Rnd returns the first number in the sequence generated from that seed value.

Usage

Use Randomize to seed the random number generator before calling Rnd to generate the number.

If you use Randomize with an argument and then repeatedly call Rnd (with no arguments), LotusScript returns the same sequence of random numbers every time you execute the script. The particular sequence of random numbers generated from a given seed depends on the platform where you are running LotusScript.

If you use Randomize without an argument, LotusScript generates a different sequence of numbers each time you execute the script.

You can call the function with no arguments as either Rnd or Rnd( ).

Language cross-reference

@Random function in formula language

Examples: Rnd function

```
Randomize -1
Print Rnd(); Rnd(); Rnd(); Rnd(); Rnd()
' Output:
' 7.548905E-02  .5189801  .7423341  .976239  .3883555
Randomize -1
Print Rnd(0)
' Output:
' .3142746
Print Rnd(); Rnd(); Rnd(); Rnd(); Rnd()
' Output:
' 7.548905E-02  .5189801  .7423341  .976239  .3883555
Print Rnd(-1)
' Output:
' .3142746
Print Rnd(-2); Rnd(0)
' Output:
' .6285492  .6285492
```

Round function

Rounds a number to a specified number of decimal places.

Syntax

Round (*numExpr* **,** *places* **)**

Elements

numExpr

Any numeric expression. The number to be rounded.

places

Any numeric expression representing the desired number of decimal places. If *places* is not an integer, it is converted to one.

Return value

Round returns a Double.

If the first non-significant digit is 5, and all subsequent digits are 0, the last significant digit is rounded to the nearest even digit. See the example that follows.

If *places* is negative, the number is rounded to *places* digits to the left of the decimal point. See the example that follows.

Language cross-reference

@Round function in formula language

Examples: Round function

```
' Round to one decimal place.
Print Round(4.23, 1)                    ' Prints 4.2
Print Round(4.35, 1)                    ' Prints 4.4
Print Round(4.45, 1)                    ' Prints 4.4

' Round to the nearest hundred.
Print Round(153.33, -2)                 ' Prints 200
```

RSet statement

Assigns a specified string to a string variable and right-aligns the string in the variable.

Syntax

RSet *stringVar* **=** *stringExpr*

Elements

stringVar

The name of a fixed-length String variable, a variable-length String variable, or a Variant variable.

stringExpr

The string to be assigned to the variable and right-aligned.

Usage

If the length of *stringVar* is greater than the length of *stringExpr*, LotusScript right-aligns *stringExpr* within *stringVar* and sets the remaining characters in *stringVar* to spaces.

If the length of *stringVar* is less than the length of *stringExpr*, LotusScript copies only as many leftmost characters from *stringExpr* as will fit within *stringVar*.

If *stringVar* contains a numeric value, LotusScript converts it to String to determine the length of the result.

If *stringVar* is a Variant, it can't contain NULL.

You cannot use RSet to assign variables of one user-defined data type to variables of another user-defined data type.

Examples: RSet statement

Example 1

```
Dim positFin As String * 20    ' String of 20 null characters
RSet positFin$ = "Right"   ' "Right" is shorter than positFin.
Print positFin$
' Prints "               Right"
' The string "Right" is right-aligned in the fixed-length
' String variable named positFin, and the initial 15
' characters in positFin are set to spaces.
```

Example 2

```
Dim x As Variant
x = "q"
RSet x = "ab"
Print x                              ' Prints "a"
' The string "q" is assigned to the Variant variable x, giving
' it a length of 1. The single leftmost character "a" of the
' two-character string expression "ab" is assigned to x.
```

RTrim function

Remove trailing spaces from a string and return the resulting string.

Syntax

RTrim[**$**] **(** *stringExpr* **)**

Elements

stringExpr

Any String expression.

Return value

RTrim returns a Variant of DataType 8 (String), and RTrim$ returns a String. RTrim returns the trimmed version of *stringExpr*, but does not modify the contents of *stringExpr* itself.

Examples: RTrim function

```
Dim trimRight As String
trimRight$ = RTrim$("   abc  ")
Print trimRight$
Print Len(trimRight$)
' Output:
'    abc
'  6
' The string "   abc" is assigned to trimRight.
' Note that the leading spaces were not removed.
```

Run statement

LotusScript Release 3 and after no longer support the Run statement. To execute a Lotus software application macro, use the Evaluate function or statement.

Second function

Returns the second of the minute (an integer from 0 to 59) for a date/time argument.

Syntax

Second (*dateExpr* **)**

Elements

dateExpr

Any of the following kinds of expression:

- A valid date/time string of String or Variant data type.

  For Notes and Domino, a 2-digit designation of a year is interpreted so that 50 through 99 represent the years 1950 through 1999 and 00 through 49 represent the years 2000 through 2049.

  For SmartSuite, LotusScript interprets the years differently. For more information, see the Year 2000 item on the Help menu of each SmartSuite product.

- A numeric expression whose value is a Variant of DataType 7 (Date/Time).
- A number within the valid date range: the range -657434 (representing Jan 1, 100 AD) to 2958465 (Dec 31, 9999 AD).
- NULL.

Return value

Second returns an integer between 0 and 59.

The data type of Second's return value is a Variant of DataType 2 (Integer).

Second(NULL) returns NULL.

Language cross-reference

@Second function in formula language

Examples: Second function

```
' Construct a message that displays the current time and the
' number of hours, minutes, and seconds remaining in the day.
Dim timeFrag As String, hoursFrag As String
Dim minutesFrag As String, secondsFrag As String
Dim crlf As String, message As String
timeFrag$ = Format(Time, "h:mm:ss AM/PM")
hoursFrag$ = Str(23 - Hour(Time))
minutesFrag$ = Str(59 - Minute(Time))
secondsFrag$ = Str(60 - Second(Time))
crlf$ = Chr(13) & Chr(10)       ' Carriage return/line feed
message$ = "Current time: " & timeFrag$ & ". " & crlf$ _
   & "Time remaining in the day: " _
   & hoursFrag$ & " hours, " _
   & minutesFrag$ & " minutes, and " _
   & secondsFrag$ & " seconds."
MessageBox(message$)
```

Seek function

Returns the file position (the byte position in a binary file or the record number in a random file) in an open file.

Syntax

Seek (*fileNumber* **)**

Elements

fileNumber

The number assigned to the file when it was opened with the Open statement.

Return value

Seek returns a Long value between 1 and 2.0E31 - 1, inclusive, unless the file position is very large. For a file position larger than 2.0E30, the return value is negative.

For a binary or sequential file, Seek returns the current byte position within the file.

For a random file, Seek returns the number of the next record within the file.

Usage

The first byte or record in a file is always file position 1.

Examples: Seek function

```
Type personRecord
   empNumber As Integer
   empName As String * 20
End Type

Dim rec1 As personRecord, rec2 As personRecord
Dim fileNum As Integer, recNum As Integer
Dim fileName As String
fileNum% = FreeFile()
fileName$ = "data.txt"
recNum% = 5

Open fileName$ For Random As fileNum% Len = Len(rec1)
rec1.empNumber% = 123
rec1.empName$ = "John Smith"
Print Seek(fileNum%)          ' Prints 1 for current position
Put #fileNum%, recNum%, rec1  ' Write data at record 5
Print Seek(fileNum%)          ' Prints 6

Seek fileNum%, 1              ' Rewind to record 1
Print Seek(fileNum%)          ' Prints 1
Rec2.empNumber% = 456
Rec2.empName$ = "Jane Doe"
Put #fileNum%, , rec2         ' Write at current position
Print Seek(fileNum%)          ' Prints 2

Close fileNum%
```

Seek statement

Sets the file position (the byte position in a binary file or the record number in a random file) in an open file.

Syntax

Seek [**#**]*fileNumber* **,** *position*

Elements

fileNumber

The number assigned to the file when it was opened with the Open statement.

position

The desired file position for the next read or write operation. In a binary or sequential file, this is a non-zero byte location; in a random file, this is a record number (in a random file).

In a binary or sequential file, the first byte is byte number 1; in a random file, the first record is record number 1.

If *position* is zero or is omitted, Seek returns an error.

Usage

The record number in a Get statement or a Put statement overrides a file position set by a Seek statement.

Writing to a file after moving the file position beyond the end of the file appends data to the end of the file.

Examples: Seek statement

```
Type personRecord
   empNumber As Integer
   empName As String * 20
End Type

Dim rec1 As personRecord, rec2 As personRecord
Dim fileNum As Integer, recNum As Integer
Dim fileName As String
fileNum% = FreeFile()
fileName$ = "data.txt"
recNum% = 5

Open fileName$ For Random As fileNum% Len = Len(rec1)
rec1.empNumber% = 123
rec1.empName$ = "John Smith"
Print Seek(fileNum%)          ' Prints 1 for current position
Put #fileNum%, recNum%, rec1  ' Write data at record 5
Print Seek(fileNum%)          ' Prints 6

Seek fileNum%, 1              ' Rewind to record 1
Print Seek(fileNum%)          ' Prints 1
Rec2.empNumber% = 456
Rec2.empName$ = "Jane Doe"
Put #fileNum%, , rec2         ' Write at current position
Print Seek(fileNum%)          ' Prints 2

Close fileNum%
```

Select Case statement

Selects a group of statements to execute, based on the value of an expression.

Syntax

Select Case *selectExpr*

[**Case** *condList*

[*statements*]]

[**Case** *condList*

[*statements*]]

...

[**Case Else**

[*statements*]]

End Select

Elements

selectExpr

An expression whose value is compared with values in the subsequent *condList* conditions. This expression is evaluated once, and its value is used repeatedly for comparison.

condList

Each *condList* is a list of conditions, one of which must be met for the subsequent group of statements to execute. Each condition takes one of the forms listed below, where *expr* is any expression:

- *expr*

  Returns TRUE if *selectExpr* matches *expr* exactly.

- *expr* **To** *expr*

  Returns TRUE if the *selectExpr* falls inclusively within this range.

  For example, if you specify 25 To 50, the corresponding group of statements is executed when *selectExpr* is any value between 25 and 50, inclusive.

- **Is** *comparisonOp expr*

  Returns TRUE when the comparison operation for *selectExpr* and *expr* is true. The comparison operator must be one of the following: **= > < <> >< <= =< >= =>**.

For example, if you specify Is < 37, then the corresponding group of statements is executed when *selectExpr* is less than 37.

statements

Statements to be executed if one of the governing conditions in the associated *condList* is the first condition to be satisfied.

Usage

The *selectExpr* is compared against each condition, within each *condList* in succession. The first time that a condition in some *condList* is satisfied, the group of statements associated with that *condList* is executed and the selection operation ends.

Either a single group of *statements* is executed, or no statements are executed. If you include a Case Else group of statements, it's executed only if *selectExpr* fails all conditions in all c*ondList* arguments.

Examples: Select Case statement

```
' One of five Print statements is selected for execution,
' depending on the value of the variable segSelect.
' Note that the Case Else clause is executed only if
' segSelect is less than 0, between 0 and 1, between 1 and 2,
' between 2 and 3, or between 5 and 6.

Dim segSelect As Double
' ...
For segSelect# = -1 to 7
   Select Case segSelect#
      Case 0       : Print "0"
      Case 1, 2    : Print "1, 2"
      Case 3 To 5  : Print "3 TO 5"
      Case Is >= 6 : Print ">=6"
      Case Else    : Print "Else"
   End Select
Next
' Output:
' Else
' 0
' 1, 2
' 1, 2
' 3 TO 5
' 3 TO 5
' 3 TO 5
' >=6
' >=6
```

SendKeys statement

Enters keystrokes in the active window as if they were entered from the keyboard.

SendKeys is not supported on Macintosh and UNIX platforms and is not supported in Lotus Domino and Notes.

Syntax

SendKeys *string* [**,** *processNow*]

string

Any string expression, specifying a sequence of keystrokes to be sent to the active window.

To repeat a keystroke in *string*, use the code {*key count*}, where *key* is the keystroke to repeat, and *count* is the number of times to repeat it. For example, "{RIGHT 3}" represents pressing the Right Arrow key three times.

Include a space between *key* and *count*; otherwise {*key count*} may be interpreted as a function key specification. For example, "{F 4}" represents pressing the letter F four times, but "{F4}" represents pressing the function key F4.

processNow

Optional. Any numeric value. A nonzero value is interpreted as TRUE; a zero (0) is interpreted as FALSE.

- If *processNow* is TRUE, script execution does not continue until after all characters in *string* have been processed by the active window.
- If *processNow* is FALSE, script execution continues immediately, whether or not *string* has been fully processed.

The default value of *processNow* is FALSE. You will usually want to specify TRUE for *processNow*.

Usage

The SendKeys statement is not legal at the module level.

To send an ordinary keyboard key or sequence of keys, such as A or 8 or DIR, simply include the character(s) in *string*.

To send non-printing keyboard keys, such as Tab or Backspace, or keys that perform actions in the active window, such as Page Up, use the key code from the following table in *string*.

Key	*Code*
Backspace	{BS} or {BKSP} or {BACKSPACE}
Break	{BREAK}
Caps Lock	{CAPSLOCK}
Clear	{CLEAR}
Del	{DEL} or {DELETE}
Down arrow	{DOWN}
End	{END}
Enter	~ or {ENTER}
Esc	{ESC} or {ESCAPE}
Help	{HELP}
Home	{HOME}
Ins	{INSERT}
Left arrow	{LEFT}
Num Lock	{NUMLOCK}
Pg Dn	{PGDN}
Pg Up	{PGUP}
Right arrow	{RIGHT}
Scroll Lock	{SCROLLLOCK}
Tab	{TAB}
Up arrow	{UP}
Function keys	{F1} to {F16}

To include a character from the following table in *string*, enclose it in braces as shown.

Character	*Code*
Brace	{{} *or* {}}
Bracket	{[} *or* {]}
Caret	{^}
Parenthesis	{(} *or* {)}
Percent sign	{%}
Plus sign	{+}
Tilde	{~}

The following table shows how to designate keys pressed in combination with Alt, Ctrl, or Shift.

*Combination **key***	*Code*	*Example*
Alt	%	%{F4} represents Alt+F4
Ctrl	^	^{F4} represents Ctrl+F4
Shift	+	+{F4} represents Shift+F4

To apply a combination key to a sequence of keys, enclose the sequence in parentheses. For example, +(xy) holds down the Shift key for both x and y. It is equivalent to +x+y.

SendKeys cannot send keystrokes to a window that is not a Windows or an OS/2 Presentation Manager program, and cannot send the Print Scrn key to any program. Also, SendKeys cannot send keystrokes to an OS/2 Presentation Manager window if that window is in the same process as the program calling SendKeys.

SendKeys generates an "Illegal function call" error if *string* contains any of the following elements:

- An unmatched parenthesis
- An illegal key code
- An illegal repeat count
- Too many characters

Note that SendKeys is often useful after Shell, to send keystrokes to the program that Shell started. Remember that Shell does not guarantee that the program is loaded before executing the statements that follow it.

Examples: SendKeys statement

```
' Use Shell to open the Windows Notepad, then SendKeys to send
' a note entered by the user to Notepad. The user can continue
' composing the note and use Notepad to save it as
' a text file.
Sub WriteNote
   Dim taskId As Integer, note As String
   note$ = InputBox("Start your note:")
   taskId% = Shell("notepad.exe", 1)
   SendKeys note$, TRUE
End Sub
WriteNote              ' Call the WriteNote sub.
```

Set statement

Assigns an object reference to a variable, or associates an object with a variable.

Use one of the following three syntaxes:

Syntax 1: Create an object and assign a reference

Set *var* **= New** *class* **[([** *argList* **])]**

Elements

var

A Variant variable, an object of the class *class*, an object of a class derived from *class*, or any variable element that accepts an object reference, such as an element of an array, list, or user-defined data type.

class

The name of the user-defined or product class of the object to be created.

argList

For user-defined classes, *argList* is the comma-separated list of arguments required by the class constructor sub New, defined in the class named by *type*. For product classes, consult the product documentation.

Syntax 2: Copy an existing object reference to another variable

Set *var1* **=** *var2*

Elements

var1

A Variant variable, an object of the same class as *var2*, an object of a class derived from *var2*'s class, or any variable element that accepts an object reference, such as an element of an array, list, or user-defined data type.

var2

An expression whose value is NOTHING, an object reference of the same class as *var1,* an object reference of a class derived from *var1*'s class, or an object reference of a class from which *var1* is derived. In the latter case, *var2* must contain an instance of *var1*'s class or a class derived from *var1*.

Syntax 3: Associate a product object with a variable

Set *var* **= Bind [** *prodClass* **] (** *objectName* **)**

Elements

var

A Variant variable, an object of *prodClass*, or any variable element that accepts an object reference, such as an element of an array, list, or user-defined data type.

Bind

The Bind keyword associates *objectName* with *var*. The association is made by name, and is valid until any of the following conditions is true:

- *var* is out of scope.
- *objectName* no longer exists.
- *var* is set to another value.

Note that you should not use Bind to associate a Lotus Notes object with a variable. Notes implicitly binds its supporting objects.

prodClass

Optional. The product class of the object *objectName*. If *prodClass* is not specified, LotusScript assumes that *objectName* is of the same class as *var*. If *var* is a Variant, you must include *prodClass*.

objectName

A string specifying the name and, optionally, the path of the product object of class *prodClass*.

The form of this string is product-specific. For example, the product object name might have the form *"ApplicationWindowName\ObjectName."* Refer to your Lotus software documentation for information about specifying product object names.

Usage

The Set statement is the object reference assignment statement. It is parallel to the Let statement, the general assignment statement for variables of all types except object reference variables.

When you use the user interface, rather than a script, to create a product object, some Lotus products implicitly declare the name you (or the product) have assigned the object as an object reference variable and bind it to the object. This allows you to use the object name in scripts without explicitly declaring a variable and binding it to the object.

To test an object reference variable for the NOTHING value, use the Is operator.

Language cross-reference

@Set function in formula language

Examples: Set statement

Example 1 (Syntax 1)

```
' The variable terPoint is an object reference variable of
' the class Point, which is already defined. The New sub for
' class Point has no arguments. Set creates a new object
' of the class Point and assigns its reference to terPoint.
Dim terPoint As Point
Set terPoint = New Point
```

Example 2 (Syntax 2)

```
' The classes Worker and Carpenter must already be defined,
' with Carpenter as a derived class of Worker. The first Dim
' statement declares x as an object reference variable of
' Worker. The second Dim statement declares y as an object
' reference variable of Carpenter. This statement also creates
' a new object of Carpenter, named "Terry"; and assigns its
' reference to the variable y. The Set statement assigns the
' reference in y to variable x. (A reference to a Carpenter
' can be assigned to a variable of Worker because Worker
' is the base class of Carpenter.)
Dim x As Worker
Dim y As New Carpenter("Terry")
Set x = y
```

Example 3 (Syntax 3)

```
' The Dim statement declares icCheckBox as an object reference
' variable of the pre-defined product class Check. The Set
' statement binds the object reference variable icCheckBox to
' the product object Checkbox1.
Dim icCheckBox As Check
Set icCheckBox = Bind("Checkbox1")
```

SetFileAttr statement

Sets the system attributes of a file.

Syntax

SetFileAttr *fileName* **,** *attributes*

SetAttr is acceptable in place of SetFileAttr.

Elements

fileName

A string expression; you can optionally include a path.

attributes

The attributes to apply to the file, expressed as the sum of any of the following Integer values:

Value	*Description*	*Constant*
0	Normal file	ATTR_NORMAL
1	Read-only	ATTR_READONLY
2	Hidden	ATTR_HIDDEN
4	System	ATTR_SYSTEM
32	Changed since last back-up	ATTR_ARCHIVE

The constants are defined in the file lsconst.lss. Including this file in your script allows you to use constant names instead of the corresponding numeric values.

Usage

Do not use SetFileAttr on an open file, unless the file has been opened as read-only.

Examples: SetFileAttr statement

```
' This script creates a file and uses SetFileAttr to set the
' file attributes to Read-Only, System, and Hidden. It then
' uses GetFileAttr to verify the file attributes.
%Include "lsconst.lss"
Dim fileNum As Integer, attr As Integer
Dim fileName As String, msg As String
fileNum% = FreeFile()
fileName$ = "data.txt"

Open fileName$ For Output As fileNum%
Close fileNum%

SetFileAttr fileName$, ATTR_READONLY + ATTR_SYSTEM + _
ATTR_HIDDEN
attr% = GetFileAttr(fileName$)
If (attr% And ATTR_READONLY) Then
   msg$ = msg$ & " Read-Only "
Else
   msg$ = msg$ & " Normal "
End If
If (attr% And ATTR_HIDDEN)    Then msg$ = msg$ & " Hidden "
If (attr% And ATTR_SYSTEM)    Then msg$ = msg$ & " System "
If (attr% And ATTR_VOLUME)    Then msg$ = msg$ & " Volume "
If (attr% And ATTR_DIRECTORY) Then msg$ = msg$ & " Directory "
Print msg$
```

```
SetFileAttr fileName$, ATTR_NORMAL   ' Reset to normal
Kill fileName$
```

Sgn function

Identifies the sign (positive or negative) of a number.

Syntax

Sgn (*numExpr* **)**

Elements

numExpr

Any numeric expression.

Return value

The following table shows the values that the Sgn function returns.

Sign of numExpr	*Value*
Negative	-1
Zero	0
Positive	1

Language cross-reference

@Sign function in formula language

Examples: Sgn function

```
Dim x As Integer, y As Integer
x% = Sgn(-45)
Print x%                          ' Prints -1
y% = Sgn(12)
Print y%                          ' Prints 1
Print Sgn(x% + y%)                ' Prints 0
```

Shell function

Starts another program.

Syntax

Shell (*program* [**,** *windowStyle*] **)**

Elements

program

A string expression whose value is the name of the program to run, including arguments. *program* can be the name of an executable file that uses a file name extension of BAT, COM, PIF, or EXE. You can omit the file name extension, and you can optionally include a complete path specification.

Using an internal DOS command name generates an error.

windowStyle

Optional. A number designating a valid window style, as specified in the following table.

Style	*Description*	*Constant*
1, 5, or 9	Normal with focus	SHELL_NORMAL_FOCUS
2	Minimized with focus (default)	SHELL_MIN_FOCUS
3	Maximized with focus	SHELL_MAX_FOCUS
4 or 8	Normal without focus	SHELL_NORMAL_NO_FOCUS
6 or 7	Minimized without focus	SHELL_MIN_NO_FOCUS

The constants are defined in the file lsconst.lss. Including this file in your script allows you to use constant names instead of the numeric values assigned to them.

Return value

If LotusScript successfully starts *program,* Shell returns the number 33.

Note To get the program's task ID, use the Shellid function instead.

If LotusScript cannot start *program,* Shell returns an error.

Usage

Shell must be called from within an expression or an assignment statement, so that its return value is used.

After Shell starts a program, LotusScript continues to execute the script without waiting to make sure the program has completed. You cannot be sure that a program started by Shell has finished running before the rest of your script is executed.

Language cross-reference

@LaunchApp function in formula language

AttachmentLaunch @command in formula language

Execute @command in formula language

Examples: Shell function

The following example is specific to Windows:

```
' Start the Windows Calculator as a normal (not minimized)
' window with focus.
Dim result As Integer
result = Shell("CALC.EXE", 1)
```

Shellid function

Starts another program and returns its task ID.

Syntax

Shellid (program [, windowStyle])

Elements

program

A string expression whose value is the name of the program to run, including arguments. *program* can be the name of an executable file that uses a file name extension of BAT, COM, PIF, or EXE. You can omit the file name extension, and you can optionally include a complete path specification.

Using an internal DOS command name generates an error.

windowStyle

Optional. A number designating a valid window style, as specified in the following table.

Style	*Description*	*Constant*
1, 5, or 9	Normal with focus	SHELL_NORMAL_FOCUS
2	Minimized with focus (default)	SHELL_MIN_FOCUS
3	Maximized with focus	SHELL_MAX_FOCUS
4 or 8	Normal without focus	SHELL_NORMAL_NO_FOCUS
6 or 7	Minimized without focus	SHELL_MIN_NO_FOCUS

The constants are defined in the file lsconst.lss. Including this file in your script allows you to use constant names instead of the numeric values assigned to them.

Return value

If the operating system is Windows or Macintosh, and LotusScript successfully starts *program*, Shellid returns the program's task ID, a number that uniquely identifies the program. With other operating systems, if LotusScript successfully starts *program*, Shellid returns the number 33.

If LotusScript cannot start *program*, Shellid returns an error.

Usage

Shellid must be called from within an expression or an assignment statement, so that its return value is used.

After Shellid starts a program, LotusScript continues to execute the script without waiting to make sure the program has completed. You cannot be sure that a program started by Shellid has finished running before the rest of your script is executed.

Examples: Shellid function

The following example is specific to Windows:

```
Dim taskId As Variant

taskId = Shellid("notepad.exe")

Print "Running task " & taskId
```

Sin function

Returns the sine, in radians, of an angle.

Syntax

Sin (*angle* **)**

Elements

angle

Any numeric expression. It is interpreted as an angle expressed in radians.

Return value

Sin returns the sine of *angle,* a Double between -1 and 1, inclusive.

Language cross-reference

@Sin function in formula language

Examples: Sin function

```
' Convert the angle of 45 degrees to radians,
' then compute and print the sine of that angle.
Dim degrees As Double, radians As Double
degrees# = 45
radians# = degrees# * (PI / 180)
Print Sin(radians#)          ' Prints .707106781186548
```

Single data type

Specifies a variable that contains a 4-byte floating-point value.

Usage

The Single suffix character for implicit data type declaration is the exclamation point (!).

Single variables are initialized to zero (0).

The range of Single values is -3.402823E+38 to 3.402823E+38, inclusive.

The smallest nonzero Single value, disregarding sign, is 1.175494351E-38.

LotusScript aligns Single data on a 4-byte boundary. In user-defined data types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Single data type

```
' Explicitly declare a Single variable.
Dim x As Single

' Implicitly declare a Single variable.
mole! = 6.02E23

Print mole!   ' Prints the value of mole.
```

Sleep statement

Causes a script to pause for at least the number of seconds specified. The script may pause longer.

Syntax

Sleep (*numExpr* **)**

Elements

numExpr

Any numeric expression.

Usage

This function provides a way for a script to wait without consuming the system resources of a spin loop. Implementation depends on the platform, but on all platforms except the legacy platforms, this function causes the LotusScript code to give up its time slice.

Accuracy is limited to the accuracy of the platform being used. If the most accurate timing is limited to milliseconds, the time specified is rounded up to the nearest millisecond.

Examples: Sleep statement

```
MessageBox "Now "
Sleep 20
MessageBox "is the time..."
      ' this messagebox appears 20 seconds after the first
messagebox
```

Space function

Returns a specified number of spaces as a string.

Syntax

Space[$] (*numExpr* **)**

Elements

numExpr

Any numeric expression. If numExpr includes a fractional part, LotusScript rounds it to the nearest integer.

Return value

The return value contains numExpr space characters. Space returns a Variant of DataType 8 (String), and Space$ returns a String.

Examples: Space function

```
' Assign a string of four spaces to the variable smallTab.
Dim smallTab As String
smallTab$ = Space$(4)
Print Len(smallTab$)
' Output:
' 4
```

Spc function

Inserts a specified number of spaces in the output from a Print or Print # statement, beginning at the current character position.

Syntax

Spc (*numExpr* **)**

Elements

numExpr

Any numeric expression whose value is between 0 and 32000, inclusive. *numExpr* designates the number of spaces to insert in the Print output.

Usage

If you specify a width for the file, *numExpr* interacts with that width as follows:

- If *numExpr* is smaller than the width, LotusScript prints *numExpr* spaces.
- If *numExpr* is larger than the width, LotusScript prints as many spaces as fit on one line, with the remainder appearing on the next line, until *numExpr* spaces have been printed.

You can set the width only for printed files. If you don't specify a width for the file, LotusScript prints exactly *numExpr* spaces.

Examples: Spc function

The Print # statement prints numbers with a leading space (omitted if the number is negative) and a trailing space.

In this example, Spc(1) inserts another space following each number and its trailing space. The second and fourth lines each begin with two spaces; the first space on the line is generated by Spc(1), and the second space on the line is the leading space before the number first printed on the line (3 or 8).

In the second line, the number 4 is followed by three spaces. These last four characters can be read as "4, trailing space, Spc(1), leading space".

```
Open "spc.tst" For Output As #1
' Define line width in SPC.TST as 10 characters.
Width #1, 10
For i = 0 To 9
   Print #1, i; Spc(1);
Next i
Close #1
' Output to the file (the display of each line here includes
' a leading quote character (') and a leading space):
'  0   1   2
```

```
'    3    4
' 5    6    7
'    8    9
```

Split function

Returns an Array of Strings that are the substrings of the specified String.

Syntax

Split(*expression* **as String[,** *delimiter* **as String[,** *count* **as Integer[,** *compare* **as Integer]]]) as Variant**

Elements

expression

The scalar String to be split into its substrings

delimiter

An optional scalar String containing the characters to separate substrings. If delimiter is not specified, then the space character " " is used

count

An Integer specifying the number of substrings to return. The default value of -1 indicates that all substrings are returned.

compare

An Integer specifying the type of comparison to use when searching for the delimiter

Return value

Split returns an Array of Strings. Each element of this array contains a substring found in *expression*.

Usage

Split parses *expression* into substrings consisting of text delimited by the separator character (or the beginning or end of the String), and not containing the separator character. These substrings are placed into an Array in order, and the Array is returned.

Whitespace is not trimmed. Carriage returns are not trimmed and do not cause separations.

If the number of results specified is greater than the number of actual results, the returned Array will equal the number of actual results

If the number of results specified is less than the number of actual results, the last element of the array returned will contain the remainder of the string. For example,

split(“this is a test”, “ ”, 2)

would return an array with element 0 = “this”, 1 = “is a test”

If count is < -1, a RunTime Arg Out of Range error is thrown

If count is 0, Split returns an array of size 0 with lbound 0 and ubound -1.

Error Handling:

Split will throw a Runtime Type Mismatch if either the expression or the delimiter is not scalar.

Split will throw a Runtime Argument Out of Range error if count is < -1 or optcompare is an invalid value.

Language cross-reference

@Explode function in formula language

Examples: Split function

```
Sub Initialize
   Dim ret As Variant
   dim teststr as string
   Dim delim As String

   teststr = "This is the Connection"
   delim = " "
   ret = split(teststr, delim)

   For x = 0 to 3
      Print ret(x)
   Next

End Sub

'OUTPUT
'This
'is
'the
'Connection
```

Sqr function

Returns the square root of a number.

Syntax
Sqr (*numExpr* **)**

Elements
numExpr
Any numeric expression greater than or equal to zero.

Return value
Sqr returns a Double. If *numExpr* is negative, Sqr returns an error.

Language cross-reference
@Sqrt function in formula language

Examples: Sqr function
```
Dim root As Double
root# = Sqr(169)
Print root#                        '  Prints 13
```

Stop statement

Simulates the occurrence of a breakpoint.

Syntax
Stop

Usage
The Stop statement operates as follows when run on the server:

- If the remote debugger is not running or is running but not enabled, Stop is ignored. Performance of the agent is not affected.
- If the remote debugger is running and enabled, STOP waits according to the time-out value specified in the server record. (Stop uses the 'Agent wait at start time' value in the Remote Debug Manager tab of the Server Tasks tab.) If the debugger does not attach to that agent within the specified time-out value, execution continues. So the STOP statement does not stop the agent completely, but waits to allow the user to attach the remote debugger to it.
- If the remote debugger is attached, the agent stops at the Stop statement as if a breakpoint was set.

The Stop statement operates as follows when in debug mode (File->Tools->Debug LotusScript) on the client:

- If the agent is running as a scheduled agent in the background, Stop is ignored.
- If the agent is run from the Agents or Actions menu, the Stop statement suspends execution of the script and transfers control to the LotusScript debugger as though a breakpoint is set at the Stop statement.
- When not in debug mode, the Stop statement is ignored.

The Stop statement is legal within a procedure or class. It is not legal at the module level.

Str function

Returns the String representation of a number.

Syntax

Str[$] (*numExpr* **)**

Elements

numExpr

Any numeric expression.

Return value

Str returns a Variant of DataType 8 (a string), and Str$ returns a String.

Usage

When LotusScript represents a positive number as a String, it inserts a leading space.

Language cross-reference

@TextToNumber function in formula language

Examples: Str function

```
' Assign the strings " 123" and "-123" to the variables
' string1 and string2, respectively. For the positive value,
' note the addition of a leading space.
Dim string1 As String, string2 As String
string1$ = Str$(123)              ' Assigns " 123"
string2$ = Str$(-123)             ' Assigns "-123"
Print string2$; string1$
' Output:
' -123 123
```

StrCompare function

Compares two strings and returns the result.

Syntax

StrCompare (*string1* **,** *string2* [**,** *compMethod*] **)**

StrComp is acceptable in place of StrCompare.

Elements

string1

Any String expression.

string2

Any String expression.

compMethod

A number designating the comparison method.

Number	*Comparison method*
0	Case Sensitive, Pitch sensitive
1	Case Insensitive, Pitch sensitive
4	Case Sensitive, Pitch insensitive
5	Case Insensitive, Pitch insensitive

Use 2 to specify string comparison in the platform's collation sequence. If 2 is specified, strings are compared bit-wise. If you omit *compMethod*, the default comparison mode is the mode set by the Option Compare statement for this module. If there is no statement for the module, the default is case-sensitive and pitch-sensitive.

Return value

The following table shows what StrCompare returns, depending on the relationship between the strings being compared.

Strings being compared	*StrCompare result*
Either string is NULL	NULL
string1 is less than *string2*	-1
string1 equals *string2*	0
string1 is greater than *string2*	1

Language cross-reference

@Compare function in formula language

Examples: StrCompare function

The following example is specific to Windows:

```
' The following results are for LotusScript in English,
' running on Windows 3.1.
Print StrCompare("abc", "ab", 0)   ' Prints 1
Print StrCompare("ab", "abc", 0)   ' Prints -1
Print StrCompare("AB", "ab", 1)    ' Prints 0
Print StrCompare("AB", "ab", 2)    ' Prints -1
```

StrConv function

Converts a string to a different case or character set.

Syntax

StrConv (*expr* **,** *conversionType* **)**

Elements

expr

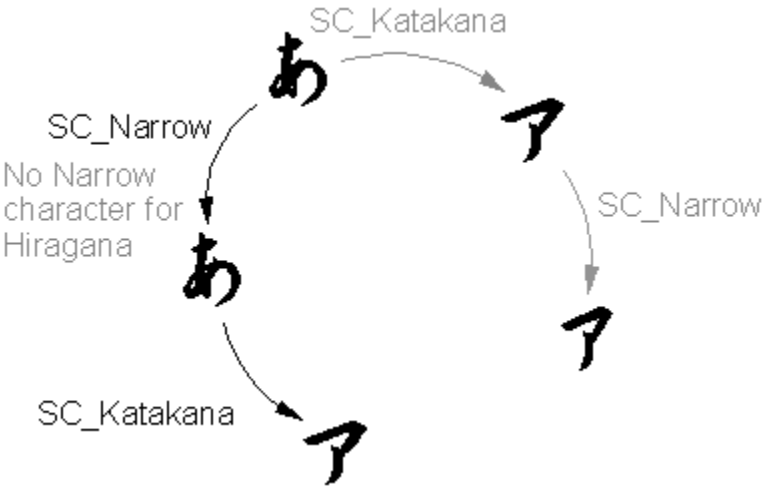
A string or numeric expression. A numeric expression is converted to a string.

conversionType

An integer that defines the type of conversion:

Constant name	*Value*	*Type of conversion*
SC_UpperCase	1	Uppercase
SC_LowerCase	2	Lowercase
SC_ProperCase	3	Proper case
SC_Wide	4	Single byte to double byte
SC_Narrow	8	Double byte to single byte
SC_Katakana	16	Hiragana to Katakana
SC_Hiragana	32	Katakana to Hiragana
SC_NativeDigit	256	Convert 0-9 to native digits
SC_ArabicDigit	512	Convert native digits to 0-9

This diagram shows an example of the conversion order.



Return value

The return value is a variant containing the result of the conversion.

Usage

The valid values for the *conversionType* elements listed in the preceding table are defined as constants in the file lsconst.lss. If you want to use the constants instead of numbers, include this file in your script.

ConversionType values can be combined (*or*ed) as follows:

- Any combination of SC_UpperCase, SC_LowerCase, and SC_ProperCase causes SC_ProperCase.
- Combining SC_Wide and SC_Narrow is illegal.
- Combining SC_Katakana and SC_Hiragana is illegal.
- If combined, the following operations occur in the following order: case operation, SC_Wide, SC_Katakana. Case operations are applied to double-byte alphanumeric characters.

If *expr* is the null string, the result is the null string. If *expr* is Null, the result is Null.

For proper case, the following numeric character codes are treated as word separators in a string literal: 0 (null), 9 (horizontal tab), 12 (form feed), 32 (space), 0x3000 (double-byte space). The following are treated as separators in a multi-line string: 10 (line feed), 13 (carriage return).

Language cross-reference

@ProperCase function in formula language

@LowerCase function in formula language

@UpperCase function in formula language

@Narrow function in formula language

@Wide function in formula language

Examples: StrConv function

```
%INCLUDE "lsconst.lss"
nameString$ = Inputbox$("Name?")
nameProper$ = Strconv(nameString$, SC_ProperCase)
Messagebox "nameProper = " & nameProper$
```

StrLeft function

Searches S1 from left to right and returns a substring consisting of the characters in S1 which are to the left of the substring S2.

Syntax

STRLeft(STRING *S1*, **STRING** *S2* **[, SHORT** *flag* **[, LONG** *occurrences* **]]) As STRING**

STRLeft$(. . .**)**

Elements

S1

A String to search for the specified pattern.

S2

The pattern to search s1 for.

flag

Flags specify which comparison to use.

Flag	*Type of Comparison*
0	Case Sensitive, Pitch Sensitive
1	Case Insensitive, Pitch Sensitive
4	Case Sensitive, Pitch Insensitive
5	Case Insensitive, Pitch Insensitive

Occurrences

Number of occurrences to match before returning the substring. Default = 1 or return on first occurrence found.

Extended example: array and String functions

Language cross-reference
@Left function in formula language

StrLeftBack function

Searches S1 from right to left and returns a substring consisting of the characters in S1 which are to the left of the substring S2.

Syntax
STRLeftBack(STRING *S1*, **STRING** *S2* **[,][SHORT** *flag***] [,] [LONG** *occurrences* **]) As STRING**

STRLeftBack$(. . .)

Elements
S1

A String to search for the specified pattern.

S2

The pattern to search s1 for.

flag

Flags specify which comparison to use.

Flag	*Type of Comparison*
0	Case Sensitive, Pitch Sensitive
1	Case Insensitive, Pitch Sensitive
4	Case Sensitive, Pitch Insensitive
5	Case Insensitive, Pitch Insensitive

Occurrences

Number of occurrences to match before returning the substring. Default = 1 or return on first occurrence found.

Extended example: array and String functions

Language cross-reference
@LeftBack function in formula language

StrRight function

Searches S1 from left to right and returns a substring consisting of the characters in S1 which are to the right of the substring S2.

Syntax

STRRight(STRING *S1*, **STRING** *S2* **[,][SHORT** *flags*] **[,] [LONG** *occurrences* **]) As STRING**

STRRight$(. . .)

Elements

S1

A String to search for the specified pattern.

S2

The pattern to search s1 for.

flags

Flags specify which comparison to use.

Flag	*Type of Comparison*
0	Case Sensitive, Pitch Sensitive
1	Case Insensitive, Pitch Sensitive
4	Case Sensitive, Pitch Insensitive
5	Case Insensitive, Pitch Insensitive

Occurrences

Number of occurrences to match before returning the substring. Default = 1 or return on first occurrence found.

Extended example: array and String functions

Language cross-reference

@Right function in formula language

StrRightBack function

Searches S1 from right to left and returns a substring consisting of the characters in S1 which are to the right of the substring S2.

Syntax

STRRightBack(STRING *S1*, **STRING** *S2* **[,][SHORT** *flag*] **[,] [LONG** *occurrences* **]) As STRING**

STRRightBack$(. . .**)**

Elements

S1

A String to search for the specified pattern.

S2

The pattern to search s1 for.

flag

Flags specify which comparison to use.

Flag	*Type of Comparison*
0	Case Sensitive, Pitch Sensitive
1	Case Insensitive, Pitch Sensitive
4	Case Sensitive, Pitch Insensitive
5	Case Insensitive, Pitch Insensitive

Occurrences

Number of occurrences to match before returning the substring. Default = 1 or return on first occurrence found.

Extended example: array and String functions

Language cross-reference

@RightBack function in formula language

StrToken function

Returns a specified word from a text string.

Syntax

StrToken(*expression* **as STRING,** *delimiter* **as STRING,** *wordNumber* **as LONG**, *compMethod* **as INTEGER) as STRING**

StrToken$(. . .)

Elements

expression

String containing the string to be scanned.

delimiter

String containing the character(s) that will delimit a word in expression.

wordNumber

Long indicating which word from expression should be returned (0 or 1 based).

compMethod

Optional Integer specifying the type of comparison to use when searching for the delimiter.

Number	*Comparison Mode*
0	case sensitive, pitch sensitive
1	case insensitive, pitch sensitive
4	case sensitive, pitch insensitive
5	case insensitive, pitch insensitive

If you omit *compMethod*, the default comparison mode is the mode set by the Option Compare statement for this module. If there is no statement for the module, the default is case sensitive and pitch sensitive.

Return value

Returns a String. The String returned is the specified word from expression.

Usage

StrToken returns the specified word from a text string. A “word” is defined as the part of a string that is delimited by the defined separator character. For example, if you specify a space (“ ”) as the separator, a word is any series of characters preceded by and followed by a space (or by the quotation marks that indicate the beginning or end of the string). e.g. “ hello there ” has four words - “”, “hello”, “there”, “”

Note The first word, "", is considered a word because it is delimited by the beginning of the string on the left and by the delimiter character on the right.

Expression is broken up into words and the word in the position specified by wordNumber is returned.

If the absolute value of wordNumber is greater than the number of words, the specified word is assumed to be the empty string "".

If wordNumber = 0, the word specified is taken to be the 1st word of the String (i.e. wordNumber=0 is equivalent to wordNumber=1)

If wordNumber < 0, the word specfied is found by counting backwards from the last word of the String.

Error Handling

StrToken will throw a Runtime Type Mismatch if the expression or delimiter is not scalar, or if wordNumber is not a long (cannot be coerced to a long by the compiler)

StrToken will throw a Runtime Argument Out of Range Error if the optionCompare value is invalid.

Language cross-reference

@Word function in formula language

Examples: StrToken function

```
Sub Initialize
   Dim delim As String
   Dim ret As String
   Dim teststr as string

   teststr = "Art, Business, Computers, Education"
   delim = ", "
   ret = strtoken(teststr, delim,3)
   Print ret
End Sub

'OUTPUT
'Computers
```

String data type

Specifies a variable used to store text strings, using the character set of the Lotus software application that started LotusScript. All strings are stored internally as Unicode characters. Strings are translated between platform-specific characters and Unicode characters during I/O operations.

Usage

The String suffix character for implicit data type declaration is the dollar sign ($).

The declaration of a string variable uses this syntax:

Dim *varName* **As String** [***** *num*]

The optional *num* argument specifies that *varName* is a fixed-length string variable of *num* characters. A fixed-length string variable is initialized to a string of null characters (the character Chr(0)).

When you assign a string to a fixed-length string variable, LotusScript truncates a longer string to fit into the declared length. It pads a shorter string to the declared length with trailing spaces.

Fixed-length strings are often used in declaring data structures for use in file I/O or C access.

An implicitly declared String variable is always a variable-length string variable.

Variable-length strings are initialized to the empty string (“”).

LotusScript aligns variable-length String data on a 4-byte boundary. In user-defined data types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space. Fixed-length strings are not aligned on any boundary.

Examples: String data type

```
' In this example, the variable-length String variable
' firstName and the fixed-length String variable homeState are
' explicitly declared and assigned appropriate String values.
' The variable adStreet is implicitly declared to be of type
' String by the $ suffix character.

' Explicitly declare a variable-length String variable.
Dim firstName As String
firstName$ = "Mark"

' Explicitly declare a fixed-length String variable.
Dim homeState As String * 4
homeState$ = "  MA"
```

```
' Implicitly declare a variable-length String variable.
adStreet$ = "123 Maple St."

Print firstName$   ' Prints "Mark"
Print adStreet$; homeState$   ' Prints "123 Maple St.  MA"
```

String function

Returns a string consisting of a particular character repeated a number of times. The character is specified as a string, or a value interpreted as a locale-sensitive ASCII character code.

Syntax

String[$] (*stringLen* **,** { *charCode* | *stringExpr* } **)**

Elements

stringLen

A numeric expression whose value is the number of characters to put in the returned string. LotusScript rounds *stringLen* to the nearest integer.

charCode

A numeric expression of data type Long. If LotusScript is running on a native ASCII platform, the value is interpreted as a code in the platform-native character set. If LotusScript is running on a native EBCDIC platform, the value is interpreted as the ASCII equivalent for the platform's current locale. Both single-byte and double-byte characters are acceptable.

stringExpr

Any string expression. The first character in this string is the character to be used in the returned string.

Return value

String returns a Variant of DataType 8 (String), and String$ returns a String.

Examples: String function

```
Dim stars As String, moreStars As String
stars$ = String$(4, Asc("*"))
moreStars$ = String$(8, "* characters")
Print stars$, moreStars$   ' Prints ****      ********
```

Sub statement

Defines a sub.

Syntax

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

[*statements*]

End Sub

Elements

Static

Optional. Directs LotusScript to save the values of the sub's local variables between calls to the sub.

Public | **Private**

Optional. Public specifies that the sub is visible outside the scope (module or class) where the sub is defined, as long as this module is loaded. Private specifies that the sub is visible only within the current scope.

A sub in module scope is Private by default; a sub in class scope is Public by default.

subName

The sub name. The names Delete, Initialize, New, and Terminated are specialized. Use these names only as described in the topics Sub Delete, Sub Initialize, Sub New, and Sub Terminate.

argList

Optional. A comma-separated list of declarations for arguments to be passed to this sub when it is called.

The syntax for each argument declaration is:

ByVal *argument* [**( )** | **List**] [**As** *dataType*]

ByVal specifies that *argument* is passed by value: that is, the value assigned to *argument* is a copy of the value specified in the sub call, rather than a reference to the original value.

*argument***()** is an array variable. *argument* **List** identifies *argument* as a list variable. Otherwise, *argument* can be a variable of any of the other data types that LotusScript supports.

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 *argument* doesn't end in a data type suffix character (and isn't covered by an existing Def*type* statement), LotusScript assigns it the Variant data type.

Enclose the entire list of argument declarations in parentheses.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

Arrays, lists, type instances, and objects can't be passed by value as arguments. They must be passed by reference.

A sub does not return a value.

A sub can be called in either of these two forms:

subName arg1, arg2, ...

Call *subName* (*arg1, arg2, ...*)

A sub definition can't contain the definition of another procedure (a function, sub, or property).

A sub member of a class cannot be declared Static.

You can exit a sub using an Exit Sub statement.

Your Lotus software application can provide special named subs for use in your scripts; see the product documentation for more information.

Examples: Sub statement

Use a sub and a function to compute the cost of buying a house as follows.

- Ask the user for the price of the house, and call the ComputeMortgageCosts sub with *price* as the argument.
- The ComputeMortgageCosts sub gathers down payment (at least 10% of cost), annual interest rate, and the term of the mortgage from the user, then calls the Payment function with 3 arguments. Annual interest and term (years) are passed by value rather than reference, so the Payment function can adjust them to compute monthly rate and monthly payment without changing the values of these variables in the ComputeMortgageCosts sub.
- If the user enters positive values, Payment returns the monthly payment. Otherwise, it returns 0. ComputeMortgageCosts then constructs an appropriate message.

```
Dim price As Single, message As String

Function Payment (princpl As Single, _
                  ByVal intrst As Single, _
                  ByVal term As Integer) As Single
   intrst! = intrst!/12
   term% = term% * 12
   ' If any of the parameters is invalid, exit the function
   ' (Payment will return the value 0).
   If princpl! <= 0 Or intrst! <= 0 Or term% < 1 Then
      Exit Function
   ' The standard formula for computing the amount of the
   ' periodic payment of a loan:
   Payment = princpl! * intrst! /(1 - (intrst! + 1) ^ _
(-term%))
End Function

Sub ComputeMortgageCosts (price As Single)
   Dim totalCost As Single, downpmt As Single
   Dim mortgage As Single, intrst As Single
   Dim monthlypmt As Single, years As Integer
EnterInfo:
   downpmt! = CSng(InputBox("How much is the down payment?"))
   ' The downpayment must be at least 10% of the price.
   If downpmt! < (0.1 * price!) Then
      MessageBox "Your down payment must be at least " _
         & Format(price! * .1, "Currency")
      GoTo EnterInfo:
   End If
   mortgage! = price! - downpmt!
   intrst! = CSng(InputBox("What is the interest rate?"))
   years% = CInt(InputBox("How many years?"))
   ' Call the Payment function to return the monthly payment.
   monthlypmt! = Payment(mortgage!, intrst!, years%)
   totalCost! = downpmt! + (monthlypmt! * years% * 12)
   If monthlypmt! > 0 Then      ' Create a multiline message.
      message$ = _
|Price | & Format(price!, "Currency") & |
Down Payment: | & Format(downpmt!, "Currency") & |
Mortgage: | & Format(mortgage!, "Currency") & |
Interest: | & Format(intrst!, "Percent") & |
Term: | & Str(years%) & | years
Monthly Payment: | & Format(monthlypmt!, "Currency") & |
Total Cost: | & Format(monthlypmt! * years% * 12, "Currency")
   Else
      message$ = "You did not enter valid input."
   End If
End Sub
```

```
' Start here.
price! = CSng(InputBox("How much does the house cost?"))
' Call the Compute MortgageCosts sub.
ComputeMortgageCosts (price!)
' Display the message.
MessageBox message$
```

Sub Delete

A user-defined sub that LotusScript executes when you delete an object belonging to the class for which the Delete sub is defined.

Syntax

Sub Delete

[*statements*]

End Sub

Usage

In the definition for a user-defined class, you can define a destructor named Delete. This sub is automatically executed whenever you delete an object belonging to the class for which you defined the Delete sub.

The Delete sub is always Public: you can't declare it as Private.

The Delete sub can't take any arguments.

The Delete sub can't be called directly; it's invoked only when the object is deleted. The name Delete can only be used as the name of a destructor; for example, it can't be used to name any other procedure or a variable.

Examples: Sub Delete

```
' Define the class Customer.
Class Customer
   Public Name As String
   Public Address As String
   Public Balance As Currency

   ' Define a constructor sub for the class.
   Sub New (Na As String, Addr As String, Bal As Currency)
      Me.Name$ = Na$
      Me.Address$ = Addr$
      Me.Balance@ = Bal@
   End Sub

   ' Define a destructor sub for the class.
   Sub Delete
```

```
      Print "Deleting customer record for: "; Me.Name$
   End Sub
End Class

' Create an object of the Customer class.
Dim X As New Customer("Acme Corporation", _
   "55 Smith Avenue, Cambridge, MA", 14.92)
Print X.Balance@
' Output:
' 14.92

' Delete the object, first running the destructor sub.
Delete X
' Output:
' Deleting customer record for: Acme Corporation."

' Then the object is deleted.
```

Sub Initialize

A user-defined sub that LotusScript executes when the module containing the Initialize sub is loaded.

Syntax

Sub Initialize

[*statements*]

End Sub

Usage

Include in the Initialize sub any statements that you want executed when LotusScript loads the containing module.

The Initialize sub is always Private.

The Initialize sub cannot take any arguments.

Examples: Sub Initialize

```
' When LotusScript loads the module, Initialize saves the name
' of the current working directory.
Dim StartDir As String
Sub Initialize          ' Store the current directory
   StartDir$ = CurDir$
End Sub

' The module changes the working directory.
' ...
```

```
' ...

' When LotusScript unloads the module, Terminate changes the
' working directory back to what it was when the module was
' loaded.
Sub Terminate ' Return to the startup directory.
   ChDir StartDir$
End Sub
```

Sub New

A user-defined sub that LotusScript executes when you create an object of the class for which the New sub is defined.

Syntax

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

[*statements*]

End Sub

Elements

argList

Optional. A comma-separated list of parameter declarations for the New sub, enclosed in parentheses. Use the following syntax for each parameter declaration:

[**ByVal**] *paramName* [**()** | **List**] [**As** *dataType*]

ByVal means that *paramName* is passed by value: that is, the value assigned to *paramName* is a copy of the value specified in the sub call, rather than a reference to the original 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 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.

If the New sub for the derived class has no arguments, and the New sub for the base class has no arguments, omit (*argList*) and *baseClass* (*baseArgList*).

baseClass **(** [*baseArgList*] **)**

Optional. The *baseClass* is the name of the class from which the derived class is derived. This name must match the *baseClass* name in the Class statement for the derived class.

The *baseArgList* is a comma-separated list of arguments for the sub New of the base class. Note that these are actual arguments, not parameter declarations. This syntax enables a call of the New sub for the derived class to furnish actual arguments to the call of the New sub for the base class.

Include this syntax in the New sub only if all of these conditions are true:

- The class being defined is a derived class.
- The New sub for the base class of this derived class requires arguments.

  Note that these arguments must be furnished to the New sub for the base class through the call of the New sub for the derived class.
- The argument list for the sub New of the base class does not match the argument list for the sub New of the derived class in number and data type of arguments; or you want to pass different arguments to the base class sub New than those passed to the derived class sub New.

When the class being defined is a derived class, each call of the New sub for the derived class generates a call of the New sub for the base class. If that base class is itself a derived class of another base class, another call is generated, and so on.

Usage

In the definition for a user-defined class, you can include a definition for the constructor sub, named New. If the definition exists, LotusScript calls this sub whenever it creates an object from that class. LotusScript calls the sub immediately after creating the object.

Examples: Sub New

```
' Define a class.
Class textObject

   ' Declare member variables.
   backGroundColor As Integer
   textColor As Integer
   contentString As String

   ' Define constructor sub.
   Sub New (bColor As Integer, tColor As Integer,_
          cString As String)
```

```
      backGroundColor% = bColor%
      textColor% = tColor%
      contentString$ = cString$
      Print "Creating new instance of text object ..."
      Print "Text object state:"
      Print "Background color:" ; Me.backGroundColor% ; _
         "Text color:" ; Me.textColor%
   End Sub

   ' Define destructor sub.
   Sub Delete
      Print "Deleting text object."
   End Sub

   ' Define a sub to invert background and text colors.
   Sub InvertColors
      Dim x As Integer, y As Integer
      x% = backGroundColor%
      y% = textColor%
      Me.backGroundColor% = y%
      Me.textColor% = x%
   End Sub
End Class

' Create a new object of class textObject.
Dim zz As New textObject(0, 255, "This is my text")
' Output:
' Creating new instance of text object ...
' Text object state:
' Background color: 0 Text color: 255

' Invert the object's background and text colors.
zz.InvertColors
' Delete the object, first running the destructor sub.
Delete zz
' Output: Deleting text object.
```

Sub Terminate

A user-defined sub that LotusScript executes when the module containing the Terminate sub is unloaded.

Syntax

Sub Terminate

[*statements*]

End Sub

Usage

Include in the Terminate sub any statements that you want executed when LotusScript unloads the containing module.

The Terminate sub is always Private.

The Terminate sub cannot take any arguments.

Examples: Sub Terminate

```
' When LotusScript loads the module, Initialize saves
' the name of the current working directory.
Dim startDir As String
Sub Initialize ' Store the current directory.
   startDir$ = CurDir$
End Sub

' The module changes the working directory.
' ...
' ...

' When LotusScript unloads the module, Terminate changes the
' working directory back to what it was when the module was
' loaded.
Sub Terminate ' Return to the startup directory.
   ChDir startDir$
End Sub
```

Tab function

Moves the print position to a specified character position within a line, when called from within a Print or Print # statement.

Syntax

Tab (*column* **)**

Elements

column

Any integer expression between 1 and 32000, inclusive, specifying a character position in the printed output. If *column* is less than 1, the Tab position defaults to 1 (the leftmost print position).

Usage

If you haven't specified a width for the file, Tab checks *column* against the current print position, and acts as follows:

- If you've already printed past the position specified by *column*, Tab prints a newline character, and then prints the next character in the *column* position on the next line.

Note Newline does not mean either chr(10) or chr(13) on all platforms. Newline is the character or sequence of characters that is used to mark the end of a line. This may be chr(10), or chr(13), but it may also be something else, because the actual value of newline depends on the platform.

- If *column* is at the current position, or after the current position, Tab prints enough spaces to move to the position specified by *column* and prints the next character in the *column* position on the current line.

If you print to a file whose width was set with the Width # statement, Tab interacts with that width as described in the following table.

Column	*Tab moves to:*
> *width*	*column* Mod *width*
< 1	column 1
< current print position	(*column* - current position) on the next line
> current print position	(*column* - current position) on the same line

Language cross-reference

@Char function in formula language

Examples: Tab function

```
Dim firstN As String, lastN As String
firstN$ = "Bob"
lastN$ = "Jeremiah"
Print firstN$; Tab(5); lastN$; Tab(1); lastN$; Tab(2); _
lastN$; Tab(3); lastN$
```

LotusScript prints the contents of firstN and lastN, using Tab() to separate them as follows:

```
Bob Jeremiah
Jeremiah
 Jeremiah
  Jeremiah
```

The semicolons in the Print statement are optional; they have no effect on the output, because the print position is determined by Tab.

Tan function

Returns the tangent, in radians, of an angle.

Syntax
Tan (*angle* **)**

Elements
angle

Any numeric expression. It is interpreted as an angle expressed in radians.

Return value
Tan returns a Double.

Language cross-reference
@Tan function in formula language

Examples: Tan function
```
' Convert the angle of 45 degrees to radians, then
' compute and print the tangent of that angle.
Dim degrees As Double, radians As Double
degrees# = 45
radians# = degrees# * (PI / 180)
Print Tan(radians#)    ' Prints 1
```

Time function

Returns the system time as a time value.

Syntax
Time[$]

Return value
Time returns a time value representing the system time.

The return value is the fractional part of the value returned by the Now function. Time returns that value as a Variant of DataType 7 (Date/Time). Time$ returns that value as a String.

Both forms return the time rounded to the nearest second.

Usage
You can call the Time function as either Time or Time( ). You can call the Time$ function as either Time$ or Time$( ).

Examples: Time function

```
Dim current As String
current$ = Time$()
Print current$        ' Prints the system time
```

Time statement

Sets the system time to a specified time. This statement is not valid on UNIX operating systems, for which you need to have root user privileges to change the system time.

Note TSyntax

Time[$] = *timeExpr*

Elements

timeExpr

Any expression whose value is a valid date/time value: either a String in a valid date/time format, or else a Variant containing either a date/time value or a string value in date/time format.

Examples: Time statement

```
' Set the system time to 6:20:15 PM using 24-hour notation.
Time = "18:20:15"
```

TimeNumber function

Returns a time value for a specified hour, minute, and second.

Syntax

TimeNumber (*hour* **,** *minute* **,** *second* **)**

TimeSerial is acceptable in place of TimeNumber.

Elements

hour

A numeric expression representing an hour (0 to 23, inclusive).

minute

A numeric expression representing a minute (0 to 59, inclusive).

second

A numeric expression representing a second (0 to 59, inclusive).

Return value

TimeNumber returns a Variant of DataType 7 (Date/Time). Its value represents time of day as a fraction of 24 hours, measured from midnight.

Usage

You can use expressions for *hour*, *minute*, and *second* to compute a time relative to another time. For example:

```
TimeNumber(3, 5, 5 - 10)
```

computes the time 10 seconds before 3:05:05 AM (the result is 3:04:55 AM).

Examples: TimeNumber function

```
' Print the time value for an hour, minute, and second.
Print TimeNumber(12, 30, 15)   ' Prints 12:30:15 PM
```

Timer function

Returns the time elapsed since midnight, in seconds.

Syntax

Timer

Return value

Timer returns the number of seconds elapsed since midnight as a Single value.

Usage

LotusScript rounds the number of seconds to the nearest hundredth.

The Randomize Statement uses the return value from Timer as its default seed value.

You can call the function as either Timer or Timer( ).

Examples: Timer function

```
' Calculate how long it takes the following loop to iterate
' 1000 times.
Dim startTime As Single
Dim elapsedTime As Single

startTime! = Timer()
For counter% = 1 To 1000
Next counter%
elapsedTime! = Timer() - startTime!

Print "10000 iterations in "; elapsedTime; " seconds"
```

TimeValue function

Returns the time value represented by a string expression.

Syntax
TimeValue (*stringExpr* **)**

Elements
stringExpr

A string expression that represents a valid date/time, or a Variant of DataType 7 (Date/Time). It can use either 12-hour or 24-hour format; for example, both "14:35" and "2:35PM" are valid. If you omit the seconds value in the *stringExpr* argument, it defaults to zero (0).

Return value
TimeValue returns a Variant of DataType 7 that contains a fractional date/time value.

Usage
If *stringExpr* specifies a date, TimeValue validates the date, but omits it from the return value.

Language cross-reference
@TextToTime function in formula language

Examples: TimeValue function
```
Dim fractionalDay As Single
fractionalDay! = TimeValue("06:00:00")
Print fractionalDay!
' Output: .25
' LotusScript assigns the value 0.25 to the variable
' fractionalDay, since 6:00 AM represents a time value
' of 6 hours, or one-quarter of a 24-hour day.
```

Today function

Returns the system date as a date value.

Syntax
Today

Return value
Today returns the system date as a Variant of DataType 7 (Date/Time).

The return value is the integer part of the value returned by the Now function.

Usage

The Today function is equivalent to the Date function.

You can call the function as either Today or Today( ).

Language cross-reference

@Today function in formula language

Examples: Today function

```
' LotusScript assigns Today's date to the String
' variable whenNow.
Dim whenNow As String
whenNow$ = Today()
Print whenNow$
' Output:
' 6/7/95
```

Trim function

Removes leading and trailing spaces from a string and returns the resulting string.

Syntax

Trim[**$**] **(** *stringExpr* **)**

Elements

stringExpr

Any string expression.

Return value

Trim returns the trimmed version of *stringExpr*, but does not modify the contents of *stringExpr* itself.

Trim returns a Variant of DataType 8 (String), and Trim$ returns a String.

Language cross-reference

@Trim function in formula language

Examples: Trim function

```
Dim trimAll As String, testString As String
testString$ = "   a bc   "
' Trim the string, removing leading and trailing spaces.
' Embedded spaces are not removed.
trimAll$ = Trim$(testString$)   ' Assigns "a bc"
Print trimAll$
```

```
Print testString$                ' Unmodified by Trim()
' Output:
' a bc
'    a bc
```

Type statement

Defines a user-defined data type consisting of one or more members.

Syntax

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

member declarations

End Type

Elements

Public | **Private**

Optional. Public specifies that the user-defined data type is visible outside the module where it is defined, as long as that module is loaded. Private specifies that the user-defined data type is visible only within the module where it is declared.

A type is Private by default.

typeName

The name of the type.

member declarations

Declarations for the members of the type. There must be at least one declaration in the type; the declarations cannot include Const statements.

Usage

Defining types

A Type statement is valid only at module level.

The word Object is illegal as a type name.

Declaring type members

A *member* is a variable declaration without the Dim, Private, Public, or Static keywords. A *member* cannot be declared to be Private, Public, or Static; it's automatically Public.

Each *member* statement declares one variable.

The data type of a *member* can be any of the scalar data types, a Variant, a fixed array, or any other user-defined data type. It cannot be the same data type as that being defined by the current Type statement.

A *member* declared as Variant can hold any scalar value, an array (fixed or dynamic), a list, or a reference to a user-defined object, a product object, or an OLE Automation object. The following rules apply to type instances that have Variant members containing arrays, lists, or objects:

- You cannot assign a type instance containing a dynamic array or a list to another type instance.
- You cannot use the Put statement to write data to a file from a type instance containing a dynamic array, a list, or an object.
- When you assign a type instance containing an object to another type instance, LotusScript increments the internal reference count of the object.

A *member* can use any LotusScript keyword, except Rem, as its name.

Declaring a type variable
A user-defined data type name is used in variable declarations in the same way as any other data type. The common variable declaration has the syntax:

Dim *varName* **As** *typeName*

This declaration declares a variable of the type *typeName* and initializes the members of the new variable. The initial values of the members are the same as for ordinary variables:

- Numeric data types (Boolean, Byte, Integer, Long, Single, Double, Currency): 0
- Variants: EMPTY
- Strings, fixed-length: A string filled with the Null character Chr(0)
- Strings, variable-length: The empty string (“”)

If a member is itself a user-defined data type, then it is assigned initial values in the same manner.

Referring to type members
Refer to members of a type using dot notation, in the form *varName.memberName*. Spaces, tabs, and newline characters are legal on both sides of the period (after *varName* and before *memberName*).

Member references can also include array subscripts if the member is an array.

Examples: Type statement

Example 1

```
' Define a type with members to hold name, area code,
' and 7-digit local phone number.
Type phoneRec
   name As String
   areaCode As Integer
   phone As String * 8
End Type

Dim x As phoneRec            ' x is a variable of type
phoneRec.
x.name$ = "Rory"             ' Assign values to x's members.
x.areaCode% = 999
x.phone$ = "555-9320"

Print "Call " & x.name$ & " at " & Str$(x.areaCode%) & "-" & _
        x.phone% Output:
' Call Rory at 999-555-9320"
```

Example 2

```
' Create an array to hold five instances of phoneRec.
Dim multiX(5) As phoneRec
multiX(2).name$ = "Maria"       ' Assign values.
multiX(2).areaCode% = 212
multiX(2).phone$ = "693-5500"

' Retrieve data from a type member.
Dim phoneLocalHold As String * 8
phoneLocalHold$ = multiX(2).phone$
Print phoneLocalHold$
' Output:
' 693-5500
```

Example 3

```
' To maintain a file that contains a phone list,
' read all of the data from the file into LotusScript.
' The data fills a list in which each element
' is an instance of the defined type.

' Create a list to hold records from the file.
Dim phoneList List As phoneRec

' Declare a phoneRec variable to hold
' each record from the file in turn. Open the file.
Dim tempRec As phoneRec
Open "c:\phones.txt" For Random Access Read Write _
   As #1 Len = Len(tempRec)

' Read the file and store the records in the list.
```

```
Dim recNum As Integer
recNum% = 1
While EOF(1) = FALSE
   Get #1, recNum%, tempRec
   phoneList(tempRec.Name$) = tempRec
   recNum% = recNum% + 1
Wend
Close #1
' Note that the Get statement automatically fills each
' member of the tempRec variable. Since tempRec and the
' elements of phoneList are both of data type phoneRec,
' tempRec can be assigned to any element of phoneList
' without reference to its members, which LotusScript
' copies automatically.
```

TypeName function

Returns a string identifying the data type of the value of an expression.

Syntax

TypeName (*expr* **)**

Elements

expr

Any expression.

Return value

Value of expr	*Return value*	*Storage of variable*
EMPTY	"EMPTY"	In Variant only
NULL	"NULL"	In Variant only
Boolean	"BOOLEAN"	
Byte	"BYTE"	
Integer	"INTEGER"	
Long	"LONG"	
Single	"SINGLE"	
Double	"DOUBLE"	
Currency	"CURRENCY"	
Date	"DATE"	In Variant only
String	"STRING"	

continued

Value of expr	*Return value*	*Storage of variable*
NOTHING	"OBJECT"	
OLE object	"OBJECT"	In Variant only
OLE error	"ERROR"	In Variant only
V_UNKNOWN (OLE value)	"UNKNOWN"	In Variant only
User-defined object or product object	The name of the object class, as an uppercase string. For example, for an object of the Employee class, LotusScript returns "EMPLOYEE."	
List	The name of the list data type, plus the word "LIST," all as an uppercase string. For example, for a list of type String, LotusScript returns "STRING LIST."	
Array	The name of the array data type as an uppercase string, followed by parentheses enclosing one space. For example, for an integer array, LotusScript returns "INTEGER( )."	

Language cross-reference

@IsNumber function in formula language

@IsTime function in formula language

@IsText function in formula language

Examples: TypeName function

```
Dim a As Variant
Print TypeName(a)        ' Prints "EMPTY"
a = 1
Print TypeName(a)        ' Prints "INTEGER"
a = "hello"
Print TypeName(a)        ' Prints "STRING"
Dim b As String
Print TypeName(b$)       ' Prints "STRING"

' Arrays
Dim arrayl(1 To 4) As Long
Print TypeName(arrayl&)  ' Prints "LONG( )"
Dim arrayV(1 To 4)
Print TypeName(arrayV)   ' Prints "VARIANT( )"
Dim y As Variant
y = arrayl
Print TypeName(y)        ' Prints "LONG( )"
```

```
' Lists
Dim listStr List As String
Print TypeName(listStr$) ' Prints "STRING LIST"
Dim listVar List
Print TypeName(listVar)  ' Prints "VARIANT LIST"
Dim p As Variant
p = listStr$
Print TypeName(p)        ' Prints "STRING LIST"

' Class instances
Class Employee
   ' ... class definition
End Class
Dim temp As Employee
Print TypeName(temp)     ' Prints "EMPLOYEE"
Set hire = New Employee
Print TypeName(hire)     ' Prints "EMPLOYEE"
Dim emps(3) As Employee
Print TypeName(emps())   ' Prints "EMPLOYEE( )"

' OLE class instances
Set cal = CreateObject("dispcalc.ccalc")
Print TypeName(cal)      ' Prints "OBJECT"
```

UBound function

Returns the upper bound for one dimension of an array.

Syntax

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

Elements

arrayName

The name of an array.

dimension

Optional. An integer argument that specifies the array dimension for which you want to retrieve the upper bound.

Return value

UBound returns an Integer.

Usage

The default value for *dimension* is 1.

LotusScript sets the upper bound for each array dimension when you declare a fixed array, or when you use ReDim to define the array dimensions of a dynamic array.

Examples: UBound function

```
' Single dimension array

Dim maxima(10 To 20)

Print UBound(maxima)    ' Output:  20

' 2-dimensional array

Dim maxima(1 to 5, 2 to 10)

Print UBound(maxima,2)  ' Output: 10

' 3-dimensional array

Dim maxima(1 to 5, 5 to 10, 10 to 15)

Print UBound(maxima,1)  ' Output: 5

Print UBound(maxima,2)  ' Output: 10

Print UBound(maxima,3)  ' Output: 15
```

UCase function

Converts all alphabetic characters in a string to uppercase, and returns the resulting string.

Syntax

UCase[**$**] **(** *expr* **)**

Elements

expr

For UCase, any numeric or string expression. For UCase$, any Variant or string expression.

Return value

UCase returns a Variant of DataType 8 (String). UCase$ returns a String.

UCase(NULL) returns NULL. UCase$(NULL) returns an error.

Usage

The function has no effect on non-alphabetic characters.

Language cross-reference

@UpperCase function in formula language

Examples: UCase function

```
' Convert a string to uppercase.
Dim upperCase As String
upperCase$ = UCase$("abc") ' Assign the value "ABC"
```

UChr function

Returns the character represented by a Unicode numeric character code.

Syntax
UChr[$] (*longExpr* **)**

Elements
longExpr
Any expression with a numeric value between 0 and 65535, inclusive.

Return value
UChr and UChr$ return the Unicode character corresponding to the value of *longExpr*.

UChr returns a Variant of DataType 8 (String). UChr$ returns a String.

Examples: UChr function
```
Dim azAlphabet As String
Dim letterCode As Long

' Iterate through the Unicode values for a through z,
' appending each corresponding letter to azAlphabet.
For letterCode& = Uni("a") To Uni("z")
   azAlphabet$ = azAlphabet$ + UChr$(letterCode&)
Next
Print azAlphabet$     ' Prints abcdefghijklmnopqrstuvwxyz
```

Uni function

Returns the Unicode numeric character code for the first character in a string.

Syntax
Uni (*stringExpr* **)**

Elements
stringExpr
Any string expression.

Return value
Uni returns a Long.

Usage
If *stringExpr* is NULL or the empty string (""), the function returns an error.

Examples: Uni function

```
' Print the Unicode character codes for A and a.
Dim x As Long, y As Long
x& = Uni("A")
y& = Uni("a")
Print x&; y&     ' Prints 65  97
```

Unlock statement

See Lock and Unlock Statements.

Use statement

Loads a module containing Public definitions needed by the module being compiled.

Syntax

Use *useScript*

Elements

useScript

A String literal, or a constant containing a String value, specifying the module to load.

The Lotus software application that you're using determines whether *useScript* must be compiled before use. Consult the product documentation for more information.

Usage

The Use statement can appear only at module level, before all implicit declarations within the module. Note that the Use statement is supported in Lotus Notes.

Loading a used module

Whenever LotusScript loads a module that contains a Use statement, LotusScript executes the Use statement before initializing the module and executing the module's Initialize sub, if the module contains one.

Referring to Public names in a used module

A used module remains loaded until it is explicitly unloaded. When a module is unloaded, references to Public names defined in that module become invalid and result in run-time errors.

Declaring Public names

A module's Public names are not visible to other modules until the first module is used. Multiple Public definitions for the same name cannot be loaded at the same time.

Using modules is transitive: if module A uses module B, and B uses C, then the Public names in C are visible in A.

Use statements must not contain circular references at compile time. If A uses B, then B, or any module that B uses by transitivity, cannot use A.

Examples: Use statement

```
Use "PreModule"
' The previously defined module PreModule is loaded.
' Any Public definitions in PreModule are available in
' the module where the Use statement appears.
```

UseLSX statement

Loads a LotusScript extensions (lsx) file containing Public definitions needed by the module being compiled.

Syntax

UseLSX *lsxLibraryName*

Elements

lsxLibraryName

A string literal specifying the lsx file to load, either a name prepended with an asterisk or the full path name of the file. If you specify a name prepended with an asterisk (for example, "*LSXODBC"), the file is determined by searching the registry, initialization file, or preferences file, depending on the client platform. The Windows 95 registry, for example, might contain an entry for HKEY_LOCAL_MACHINE, SOFTWARE, Lotus, Components, LotusScriptExtensions, 2.0, LSXODBC, whose value is "c:\notes95\nlsxodbc.dll."

lsxLibraryName can contain "?", an optional flag that signals the lsx file is not necessary during run time. The question mark is part of the String, * and % flags follow the ? if desired. e.g. "?*NAME_OF_LIBRARY". For example:

```
const lsxLibraryName = "?NAME_OF_LIBRARY"
UseLSX lsxLibraryName
```

Usage

LotusScript registers the Public classes defined in the lsx file for use in the module containing the UseLSX statement. Other modules that use this containing module can also access these Public classes.

Note that Lotus Notes supports the UseLSX statement. The UseLSX statement loads a .LSX file containing Public definitions. These definitions then become available to the current script. Once the .LSX file has been downloaded, its classes are browsable in the Notes class browser.

The Notes platform has a registry of LSXes. If the file-specification string in the UseLSX statement begins with an asterisk (*), then Notes looks in the registry for the name consisting of the rest of the string. The registry entry for that name specifies the file location in the platform file system.

The "_" is reserved for Notes specific dlls. This is a change put in as of Notes 4.5.1. If you attempt to load a dll in Notes 4.51 or greater using LotusScript and the name of the dll is preceded by an underscore you will receive the error "Error in loading DLL".

A library name prefixed with a '?' is considered to be optional at run time. The library must be present at compile time in order to compile the script, however, if the LSX cannot be loaded at run time, the script will still execute as long as classes defined by the LSX are not referenced or functions/procedures defined by the LSX are not called from the script. If the LSX is not loaded and a line of script references an LSX class or procedure, the following errors would be thrown.

ERR = 230
ERROR = Unknown class instance

ERR = 48
ERROR = Error in loading DLL

Examples: UseLSX statement

```
UseLSX "appdll"
' The file appdll is loaded. Public definitions in the file
' are available to the module where the UseLSX statement
' appears.

const MyLib = "?appdll"
UseLSX MyLib

' Same as above but if the file appdll.dll is not found,
' script continues to load and execute. A run-time error
' occurs if an unknown object is found.
```

UString function

Returns a string of identical characters. You can specify the repeating character either by its Unicode numeric code, or as the first character in a string argument.

Syntax

UString[$] (*stringLen* **,** { *charCode* | *stringExpr* } **)**

Elements

stringLen

A numeric expression whose value is the number of characters to put in the returned string. LotusScript rounds *stringLen* to the nearest integer.

charCode

A numeric expression whose value specifies the Unicode numeric character code for the repeating character. LotusScript rounds *charCode* to the nearest integer.

Unicode codes range from 0 through 65535 inclusive. The Uni function returns the Unicode code for a given character.

stringExpr

Any string expression. The first character in this string is the character to be used for the repeating character.

Return value

UString returns a Variant of DataType 8 (String). UString$ returns a String.

Usage

If the value of *charCode* is less than 0 or greater than 65535, the function returns an error.

Language cross-reference

@Repeat function in formula language

Examples: UString function

```
Dim stars As String, moreStars As String
stars$ = UString$(4, Uni("*"))
moreStars$ = UString$(8, "*chars")
Print stars$, moreStars$   ' Prints ****      ********
```

Val function

Returns the numeric value represented by a string.

Syntax
Val (*stringExpr* **)**

Elements
stringExpr

Any string expression that LotusScript can interpret as a numeric value. It can contain any of the following kinds of characters.

- Digits (0 1 2 3 4 5 6 7 8 9)
- Other characters in hexadecimal integers (a b c d e f A B C D E F)
- Sign characters (+ -)
- Decimal point (.)
- Exponent characters (E e D d)
- Prefix characters in binary, octal, and hexadecimal integers (& B O H)
- Suffix type characters (% & ! # @)

Return value
Val returns the converted part of *stringExpr* as a Double.

Usage
Val strips out spaces, tabs, carriage returns, and newlines from *stringExpr*. It starts converting from the beginning of the string and stops when it encounters a character other than those listed for *stringExpr* in the preceding list.

Language cross-reference
@TextToNumber function in formula language

Examples: Val function
```
Dim hexVal As Double, streetNum As Double
' Assign the hexadecimal value FF (decimal 255).
hexVal# = Val("&HFF")
' Assign the value 106.
streetNum# = Val("     106 Main St.")
Print hexVal#; streetNum#
' Output:
' 255  106
```

Variant data type

Specifies a 16-byte variable that can contain data of any scalar type, an array, a list, or an object.

Usage

A variable that is declared without a data type or a suffix character is of type Variant.

Variant values are initialized to EMPTY.

A Variant variable can contain values of any scalar data type, or any of the following special values.

- Array: A declared array may be assigned to a Variant variable. The reverse is not true; for example, a Variant variable containing an array may not be assigned to a declared array variable.
- List: A list may be assigned to a Variant variable. The reverse is not true; for example, a Variant variable containing a list may not be assigned to a declared list variable.
- Object reference: A reference to any instance of a user-defined class or product class, or to an OLE Automation object, may be assigned to a Variant variable.
- Date/time value: An 8-byte floating-point value representing a date/time may be assigned to a Variant variable. The integer part represents a serial day counted from Jan 1, 100 A.D. Valid dates are represented by integer numbers in the range -657434 (representing Jan 1, 100 A.D.) to 2958465 (representing Dec 31, 9999 A.D.). The fractional part represents the time as a fraction of a day, measured from time 00:00:00 (midnight on the previous day). In this representation of date/time values, day 1 is the date December 31, 1899.
- NULL: A Variant can take the value NULL either by explicit assignment, or by the evaluation of an expression containing NULL as an operand. (For most expressions, if one or both operands are NULL, the expression evaluates to NULL.)
- EMPTY: In expressions, EMPTY is converted to 0 for numeric operations, and to the empty string ("") for string operations. Variants take the value EMPTY only upon initialization, or upon assignment from another Variant whose value is EMPTY.

A Variant cannot contain an instance of a user-defined type.

To determine the data type of the value in a Variant variable, use the DataType or TypeName function.

LotusScript aligns Variant data on an 8-byte boundary. In user-defined data types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Variant data type

```
' Explicitly declare a Variant variable.
Dim someV As Variant

' Use the Variant variable to hold a Currency value.
Dim price As Currency
price@ = 20.00
someV = price@
Print DataType(someV)   ' Prints 6 (Currency)

' Use the Variant variable to hold an object reference.
Class Product
   Sub Sell(toCustomer)
      ' ...
   End Sub
End Class
Dim knife As New Product
Set someV = knife
Call someV.Sell("Joe Smith")   ' Calls Product method

' Use the Variant variable to hold an array.
Dim salesArray()
ReDim salesArray(3)
salesArray(1) = 200
salesArray(2) = 350
salesArray(3) = 10
someV = salesArray
Print someV(1)       ' Prints 200

' Use the Variant variable to hold a list.
Dim customerList List
customerList("one") = "Butcher"
customerList("two") = "Baker"
someV = customerList
Print someV("one")   ' Prints Butcher
```

Weekday function

Returns the day of the week, an integer from 1 to 7, for a date/time argument.

Syntax

Weekday (*dateExpr* **)**

Elements

dateExpr

Any of the following kinds of expression:

- A valid date/time string of String or Variant data type.

  For Notes or Domino, LotusScript interprets a 2-digit designation of a year in a date/time string so that 50 through 99 represent the years 1950 through 1999 and 00 through 49 represent the years 2000 through 2049.

  For SmartSuite, LotusScript interprets the years differently. For more information, see the Year 2000 item on the Help menu of each SmartSuite product.

- A numeric expression whose value is a Variant of DataType 7 (Date/Time).
- A number within the valid date range: -657434, representing Jan 1, 100 A.D., to 2958465, representing Dec 31, 9999 A.D.
- NULL.

Return value

Weekday returns an integer between 1 and 7.

The data type of the return value is a Variant of DataType 2 (Integer).

Weekday(NULL) returns NULL.

Usage

Sunday is day 1 of the week.

Language cross-reference

@Weekday function in formula language

Examples: Weekday function

```
Dim x As Variant, wd As Integer
x = DateNumber(1993, 7, 7)
wd% = Weekday(x)
Print wd%
' Output:
' 4
```

While statement

Executes a block of statements repeatedly while a given condition is true.

Syntax

While *condition*

[*statements*]

Wend

Elements

condition

Any numeric expression. LotusScript interprets a value of 0 as FALSE, and interprets any other value as TRUE.

Usage

LotusScript tests *condition* before entering the loop and before each subsequent repetition. The loop repeats while *condition* is TRUE. When *condition* is FALSE, execution continues with the first statement following the Wend statement.

Language cross-reference

@While function in formula language

@For function in formula language

Examples: While statement

```
' While a user-specified interval (in seconds) is elapsing,
' beep and count the beeps. Then tell the user the number
' of beeps.

Dim howLong As Single, howManyBeeps As Integer
Function HowManyTimes (howLong As Single) As Integer
   Dim start As Single, finish As Single, counter As Integer
   start! = Timer
   finish! = start! + howLong!
   While Timer < finish!
      Beep
      counter% = counter% + 1
```

```
   Wend
   HowManyTimes = counter%
End Function
howLong! = CSng(InputBox _
   ("For your own sake, enter a small number."))
howManyBeeps% = howManyTimes(HowLong!)
MessageBox "Number of beeps:" & Str(howManyBeeps%)
```

Width # statement

Assigns an output width to a sequential text file.

Syntax

Width #*fileNumber* **,** *width*

Elements

#*fileNumber*

The file number that LotusScript assigned to the file when it was opened. The file must be open. You must include both the pound sign (#) and the file number.

width

An integer expression in the range 0 to 255, inclusive, that designates the number of characters LotusScript writes to a line before starting a new line. A *width* of 0, the default, specifies an unlimited line length.

Usage

If data to be written would cause the width of the current line to exceed the Width # setting, that data is written at the beginning of the next line instead.

The Print # statement is the only output statement affected by the Width # statement. Write # ignores the width set by Width #.

Examples: Width # statement

```
Dim fileNum As Integer
Dim fileName As String
fileName$ = "data.txt"
fileNum% = FreeFile()

Open fileName$ For Output As fileNum%
Width #fileNum%, 20

Print #fileNum%, "First line";

' The next data item, a long string, would extend the
' current line beyond 20 characters; so it is written
' to the next line in the file. An individual data item
```

```
' cannot be split across lines; so the entire 33-character
' string is written to one line.
Print #fileNum%, "This will go on one line, though.";

' The next data item is written to the next line
' in the file because the current line is already wider
' than 20 characters.
Print #fileNum%, "But this is on another.";
Print #fileNum%, "The End";
Close fileNum%

' Output:
' First line
' This will go on one line, though.
' But this is on another.
' The End
```

With statement

Provides a shorthand notation for referring to members of an object.

Syntax

With *objectRef*

[*statements*]

End With

Elements

objectRef

An expression whose value refers to a user-defined object, a product object, or an OLE object.

Usage

The With statement lets you refer to the members of an object using a dot to represent the object name.

You can also use a dot outside of a With statement to represent the currently selected product object.

You cannot use a dot to refer to the selected product object in a With statement. LotusScript assumes that any member preceded by a dot is a member of *objectRef*.

You can nest With statements up to 16 levels.

LotusScript does not support entering a With statement using GoTo.

Reassigning the *objectRef* variable inside the With statement does not change the object referred to by the dot. However, any other operation reassigns the object. See the following example.

Examples: With statement

```
Class Employee
   Public empName As String
   Public status As Integer
   Sub SetName
      empName$ = InputBox$("Enter name:")
   End Sub
End Class

Dim emp As New Employee
Dim emp2 As New Employee

With emp
   Call .SetName    ' Calls InputBox$ to prompt
                    ' for an employee name to assign
                    ' to emp.empName.
   Set emp = emp2   ' Reassigns the emp object variable,
                    ' to refer to a different object
                    ' (the same object that emp2 refers to).
   .status% = 1     ' Sets status of the object that emp
                    ' referred to when the With statement
                    ' was entered.
   emp.status% = 0  ' Sets both emp.status and emp2.status,
                    ' because of the preceding Set statement.
   Print .status% ; emp.status% ; emp2.status%
   ' Output:  1  0  0
End With
```

Write # statement

Writes data to a sequential text file with delimiting characters.

Syntax

Write #*fileNumber* [**,** *exprList*]

Elements

#*fileNumber*

The file number that LotusScript assigned to the file when it was opened. You must include both the pound sign (#) and the file number.

exprList

Optional. The list of String or numeric expressions to be written to the file, separated with commas.

If you omit *exprList*, Write # writes a blank line to the file.

The *exprList* can't include arrays, lists, type variables, or objects. The *exprList* can include individual array elements, list elements, or type members.

Usage

Use Write # only with files opened for either Output or Append.

Use the Input # statement to read data written by Write #.

Write # ignores the file width set by the Width # statement. Data items are separated with commas, and a newline character is inserted after all data has been written to the file.

LotusScript inserts "chr(10)" to represent the newline character in any multi-line string (for example, a string that you type in using vertical bars or braces). If you Print the string to a file, this newline character will be translated into the platform-specific newline character(s). If you Write the string to a file, no translation is done.

Note Newline does not mean either chr(10) or chr(13) on all platforms. Newline is the character or sequence of characters that is used to mark the end of a line. This may be chr(10), or chr(13), but it may also be something else, because the actual value of newline depends on the platform.

Note When reading a multiline string from a sequential file written by the Write # statement, use Input, not Line Input.

The following table shows how the Write # statement behaves with various data types specified in *exprList*.

Data type	*Write # statement behavior*
Numeric	Omits leading and trailing spaces.
String	Encloses all strings in double quotation marks. Pads fixed-length strings with spaces as needed.
Variant of DataType 7 (Date/Time)	Uses one of the following date formats: #yyyy-mm-dd hh:mm:ss# #yyyy-mm-dd# #hh:mm:ss# If either the date part or the time part is missing from the value, LotusScript writes only the part provided to the file.
Variant with the value EMPTY	Writes a comma without data to the file. If that variable is the last item on the line, the comma is omitted.
Variant with the value NULL	Writes the string NULL to the file.

Examples: Write # statement

```
Dim fileNum As Integer, empNumber As Integer, I As Integer
Dim fileName As String, empName As String
Dim empLocation As Variant
Dim empSalary As Currency

fileNum% = FreeFile()
fileName$ = "data.txt"

' Write out some employee data.

Open fileName$ For Output As fileNum%
Write #fileNum%, "Joe Smith", 123, "1 Rogers Street", _
    25000.99
Write #fileNum%, "Jane Doe", 456, "Two Cambridge Center", _
    98525.66
Write #fileNum%, "Jack Jones", 789, "Fourth Floor", 0
Close fileNum%

' Read it all back and print it.
Open fileName$ For Input As fileNum%

For I% = 1 To 3
   Input #fileNum%, empName$, empNumber%, empLocation, _
      empSalary@
   Print empName$, empNumber%, empLocation, empSalary@
Next I%

Close fileNum%

' Output:
' LotusScript prints out the contents of the file
' C:\data.txt in groups of four values each. Each group
' consists of a String, an Integer, a Variant, and
' a Currency value, in that order.
```

Year function

Returns the year, as a 4-digit integer, for a date/time argument.

Syntax

Year (*dateExpr* **)**

Elements

dateExpr

Any of the following kinds of expressions:

- A valid date/time string of String or Variant data type.

  For Notes or Domino, LotusScript interprets a 2-digit designation of a year in a date/time string so that 50 through 99 represent the years 1950 through 1999 and 00 through 49 represent the years 2000 through 2049.

  For SmartSuite, LotusScript interprets the years differently. For more information, see the Year 2000 item on the Help menu of each SmartSuite product.

- A numeric expression whose value is a Variant of DataType 7 (Date/Time).
- A number within the valid date range: -657434, representing Jan 1, 100 AD, to 2958465, representing Dec 31, 9999 AD.
- NULL.

Return value

Year returns an integer between 100 and 9999.

The data type of the return value is a Variant of DataType 2 (Integer).

Year(NULL) returns NULL.

Language cross-reference

@Year function in formula language

Examples: Year function

```
Dim x As Variant
Dim yy As Integer
x = DateNumber(1995, 4, 1)
yy% = Year(x)
Print yy%
' Output:
' 1995
```

Yield function and statement

Transfers control to the operating system during script execution.

Note the Yield function and statement are not supported under OS/2.

Syntax
Yield

DoEvents is acceptable in place of Yield.

Return value
The Yield function returns 0 as an Integer value.

Usage
The Yield function and statement transfer control to the operating system, so that it can process the events in its queue. In Windows, the operating system does not return control until it has processed all outstanding events, including those generated by a SendKeys statement.

The Yield function and statement are legal within a procedure or a class. They are not legal at the module level.

You can call the function as either Yield or Yield().

Examples: Yield function and statement
Yield control to allow the user to perform one or more calculations. When the user is done, continue with the script.

The DoCalc sub uses a Shell statement to start the Windows calculator. The Shell statement returns the calculator task ID (also known as the module handle). In a While loop, the sub calls the GetModuleUsage Windows 3.1 API function, which returns the module reference count (how many instances of the calculator are currently running). The Yield statement yields control to the calculator. When the user closes the calculator, GetModuleUsage returns a reference count of 0, the While loop ends, and the sub displays an appropriate message.

If you remove the While loop (try it), the message box appears as soon as the calculator begins running. In other words, the script continues to execute without yielding control to the calculator.

```
' Declare the Windows 3.1 API function at the module level.
Declare Function GetModuleUsage Lib "Kernel" _
   (ByVal taskID As Integer) As Integer

Sub DoCalc
   Dim taskID As Integer
   ' Start the Windows calculator, returning its task ID.
   taskID% = Shell("calc.exe", 1)
```

```
   ' As long as the module is still running, yield.
   Do While GetModuleUsage(taskID%) > 0
      Yield
   Loop
   ' When the user closes the calculator, continue.
   MessageBox "Calculations done"
End Sub

DoCalc                ' Call the DoCalc sub.
```

Appendix A
Language and Script Limits

This appendix describes LotusScript language limits of several kinds: for example, the legal ranges in data representation, the limits on numerical specifications within statements, and the maximum number of different kinds of elements that can be defined in a script.

Limits on numeric data representation in LotusScript

The following table lists the legal range of values for the numeric data types.

Data type	*Range*
Boolean	0 (False) or -1 (True)
Byte	0 to 255
Integer	-32,768 to 32,767
Long	-2,147,483,648 to 2,147,483,647
Single	-3.402823E+38 to 3.402823E+38 *Smallest non-zero value (unsigned):* 1.175494351E-38
Double	-1.7976931348623158E+308 to 1.7976931348623158E+308 *On UNIX platforms:* -1.797693134862315E+308 to 1.797693134862315E+308 *Smallest non-zero value (unsigned):* 2.2250738585072012-308
Currency	-922,337,203,685,477.5625 to 922,337,203,685,477.5625 *On UNIX platforms:* -922,337,203,685,477.5666 to 922,337,203,685,477.5666 *Smallest non-zero value (unsigned):* .0001

The legal range of values of binary, octal, or hexadecimal integers is the range for Long integers (see the preceding table). The following table lists the maximum number of characters needed to represent integers in binary, octal, and hexadecimal notation. This is also the maximum number of characters that the Bin, Oct, or Hex function returns.

Integer type	*Maximum number of characters needed to represent a value*
Binary	32
Octal	11
Hexadecimal	8

Limits on string data representation in LotusScript

The following table lists the limits on representation of string data.

Item	*Maximum*
Number of strings	Limited by available memory.
Total string storage	Limited by available memory.
Length of a string literal	16,267 characters (32,000 bytes).
Length of a string value	2G bytes
Total string literal storage in a module	2G bytes

Note Even though strings in LotusScript 4 can be longer than 64K, there are still restrictions with the length of the string you can read or write using the GET and PUT statements. The only combination of filetypes that will work with long strings is with a binary file and a variable-length string. Fixed-length strings, strings in variants, and random files will not work with strings greater than 64K in length because they have a two-byte header which contains the length of the string. Two bytes cannot represent more than 64K.

Limits on array variables in LotusScript

The following table lists limits on representation of data by array variables.

Item	*Maximum or range*
Array storage size	Limited by available memory
Number of dimensions	8
Bounds of a dimension	-32,768 to 32,767 (the range of values of the Integer data type)
Number of elements	Determined by memory available for data, and by the storage size of each element of the array, which varies with the array data type. For example, a Long one-dimensional fixed array declared in type scope can have 16,128 elements. (The total storage size available for fixed-size data in module scope is 64K bytes, and a Long element requires 4 bytes for storage.)

Limits on file operations in LotusScript

The following table lists limits on miscellaneous items related to file operations and I/O.

Item	*Maximum*
Number of files open simultaneously	Determined by the product from which you start LotusScript
fileNumber in Open statement	255
recLen in Open statement	32,767
Line length of a line written by Write statement	255 characters
Number of items in Print, Write, or Input statement	255
Number of characters in *path* in MkDir, RmDir, or ChDir statement	128. This includes the drive specifier, if any.

Limits in miscellaneous source language statements in LotusScript

The following table lists limits on miscellaneous language elements.

Item	*Maximum*
Number of characters in a LotusScript identifier, not including a data type suffix character	40
Number of arguments in definition of a function or sub	31
Number of labels in an On...GoTo statement	255

Limits on compiler and compiled program structure in LotusScript

The following table lists limits on miscellaneous items related to compiling a script.

Item	*Maximum*
Number of lines per script or source file, not including the contents of %Include files	64K
Depth of nested %Include directives	16
Number of compilation errors before the LotusScript compiler halts	20
Number of symbols in a module's symbol table.	64K
Number of recursive calls (recursion level for a given function)	32Kbyte stack size
Storage size of all data in a given scope (See "Storage size of data," below.)	Module: Limited by available memory. Class: 64K bytes Procedure: 32K bytes
Size of executable module code	Limited by available memory.

Storage size of data

The limits on the storage size of data in a given scope apply to fixed-size variables: scalar variables except for variable-length strings; user-defined type variables; and fixed arrays of these scalar variables and user-defined type variables. Depending on the order of declaration, alignment of variables on storage boundaries can take extra space. For example, an Integer variable is aligned on a 2-byte boundary, and a Long variable is aligned on a 4-byte boundary.

The maximum size of data in each dynamic variable (each variable-length string, each list, each dynamic array, and each instance of a class) is limited by available memory. However, each such variable will use 4 bytes for data in the scope where it is declared.

Because of run-time needs, LotusScript might generate an **Out of stack** error just before it reaches the data storage size limit.

Appendix B
Platform Differences

The LotusScript language and functionality on the OS/2 platform, the UNIX platform, the Macintosh platform, and the OS/400 platform differ in various ways from the language and functionality described in the rest of this language reference. This appendix describes the differences.

OS/2 platform differences in LotusScript

Language construct differences

Construct	*Usage in OS/2*
Command	Command-line arguments are not normally used on OS/2. However, if the Lotus software application permits arguments, they are returned.
CreateObject	Not supported. Generates a run-time error.
GetObject	Not supported. Generates a run-time error.
Shell	The window style option is not supported for an OS/2 system application or for a user application that saves its environments via Profile. The default window style is normal with focus. Shell always returns a valid value greater than 31.

File system differences

LotusScript supports both HPFS and FAT file systems:

- The FAT file system supports conventional file names only. Conventional file names consist of up to 8 characters, a period separator, and up to 3 characters.
- The HPFS file system recognizes both conventional and long file names. Long file names can be up to 254 characters in length, including any number of periods. Blanks are supported if the file name is enclosed in double quotes. A file name consisting either of all periods or all blanks is not supported.

HPFS requires 500K of system memory. Each OS/2 PC must have at least 6MB of memory as a minimum requirement; otherwise performance will be adversely affected.

Files with long file names or blank spaces can be copied only to a diskette or disk formatted with FAT using the direct-manipulation method. Long file names are truncated to conventional file length when moved from a HPFS to a FAT file system. The long file name is saved as an extended attribute until the file is copied back to an HPFS disk using the direct-manipulation method and the workplace shell. The use of HPFS files incorrectly transferred to a FAT file system results in a run-time error.

An asterisk (*) as a wildcard in a file name indicates that any character can occupy that position and all remaining character positions. A question mark (?) as a wildcard in a file name indicates that any character can occupy that position only.

File names are not case sensitive.

Other differences

OLE functions are not supported. This limitation affects CreateObject and GetObject.

OS/2 users can invoke REXX applications from LotusScript.

UNIX platform differences in LotusScript

Language construct differences

Construct	*Usage in UNIX*
ActivateApp	Not supported. Generates a run-time error.
ChDir	A run-time error is generated if LotusScript cannot interpret the argument to ChDir, for example if a drive letter is contained in the argument.
ChDrive	Generates a run-time error unless the drive argument is the empty string (""), signifying the default drive.
CreateObject	Not supported. Generates a run-time error.
CurDir, CurDir$	Generates a run-time error unless the drive argument is the empty string (""), signifying the default drive.
CurDrive, CurDrive$	Return the empty string (""), since there are no drive letters on UNIX.
Date, Date$	For reasons of security and system integrity, only the superuser can change the date on a UNIX system. Attempting to change the date under any other username will generate a run-time error. Attempting to change the date while logged in as superuser will change the date system-wide.

continued

Construct	*Usage in UNIX*
Declare	The Pascal calling convention for external function calls is not supported. All external function calls must use the CDECL calling convention. Specifying an ordinal number (using the Alias clause) is not supported. This will return a run-time error at the point of the call to the illegally declared function.
Dir, Dir$	If ATTR_VOLUME only is specified, returns the empty string. If any other attribute is specified, ignores the attributeMask argument and behaves as if all files have the attribute Normal. Returns all files for "*.*", not just those containing ".". Returns only those files ending with a period for "*.", not every file without an extension.
FileLen, Len, LenB, LenBP, LOF	Strings containing line terminators are smaller than on DOS/Windows platforms. The line terminator is one character (linefeed), not two. Therefore the return value of these functions will be smaller for strings on UNIX than on Windows.
GetFileAttr	Generates a run-time error if a drive letter is included in the argument. Does not return the following attributes: ATTR_HIDDEN, ATTR_ARCHIVE, ATTR_VOLUME, ATTR_SYSTEM.
GetObject	Not supported. Generates a run-time error.
Input #, Input, Input$, InputB, InputB$, Line Input, Print, Write #	Compiled scripts using these constructs may be platform-specific, since file data is stored in a platform-specific manner. UNIX character set, byte order, line terminator, and numeric precision specifics may affect the portability of scripts using these functions.
IsObject, IsUnknown	See "Other differences," below.
Open, Lock, Unlock	No explicit or implicit file locking is supported on UNIX. This implies the following: • LotusScript for UNIX allows the user to copy, open, etc., a file that is already opened for reading. Thus, the Name statement works differently on UNIX. • The Open statement may specify only Shared as its lock status. Lock Read, Lock Write, and Lock Read Write will cause a run-time error. • The Lock and Unlock statements will cause a run-time error.
SendKeys	Not supported. Generates a run-time error.
SetFileAttr	Ignores the attributes ATTR_HIDDEN, ATTR_ARCHIVE, and ATTR_VOLUME.

continued

Construct	*Usage in UNIX*
Shell	Window styles are ignored.
Time, Time$	For reasons of security and system integrity, only a superuser can change the time on a UNIX system. Attempting to change the time under any other username will generate a run-time error. Attempting to change the time while logged in as superuser will change the time system-wide.

File system differences

LotusScript respects all aspects of UNIX file system security. This difference affects Kill, Open, and RmDir.

There are no drive letters on UNIX. All devices reside under the root directory. If you use a pathname containing a drive letter, LotusScript may return an error. For the %Include directive, this is a compiler error; for all other uses, this is a run-time error. (Note that since UNIX allows ":" in file names, the statement Dir$("a:") is legal. It searches the current directory for a file named a:.)

UNIX uses the "/" character (slash) as the directory separator while DOS/Windows platforms use "\" (backslash). LotusScript supports the use of slash and backslash, with the following restrictions:

- String literals. If a slash is used in a string literal that is a pathname argument, the .LSO file generated will not run on other platforms, unless that platform supports slash (for example, the UNIX platform).
- String variables. If you assign a string literal containing a slash to a variable, and then pass the variable as a pathname argument, a run-time error occurs if the platform does not support slash pathnames (for example, the DOS/Windows platform).

UNIX allows a wider variety of characters in pathnames than DOS/Windows platforms. For example, more than one "." may appear in a valid UNIX pathname.

LotusScript cannot use UNIX filenames (as opposed to pathnames) that contain the "\" character, since this character is always a path separator on other platforms.

UNIX uses the linefeed (ASCII 10) character as the line terminator. Other platforms use other characters. This difference means that files manipulated with the same LotusScript code, but executed on different platforms, may have different sizes. For instance, the Macintosh platform uses the carriage return character as the line terminator, so text files written on that platform have the same length as files written on UNIX. Since the Windows platform

uses a two-character sequence, text files written there are larger than text files written on UNIX, given identical source code.

Other differences

Function aliasing with ordinal numbers (using the Alias clause in the Declare statement) is not possible on UNIX, because UNIX has no notion of numbering the routines in a shared library.

Where wildcards are permitted in file path strings, LotusScript supports the use of UNIX regular expressions in addition to the "*" and "?" characters. However, using regular expressions in file path strings makes the script platform-dependent.

The Like operator does not use use the same regular expression syntax as the UNIX shell. It uses LotusScript regular expressions.

OLE is not supported on LotusScript Release 3.0 for UNIX platforms. This difference affects CreateObject, GetObject, IsObject, and IsUnknown. The CreateObject and GetObject functions will raise run-time errors when executed on UNIX platforms. The IsObject function tells if a variable refers to a native or product object, but not an OLE object, since OLE objects don't exist on the UNIX platform. The IsUnknown function always returns FALSE on UNIX, since there is no way for a Variant expression to receive the V_UNKNOWN value.

Macintosh platform differences in LotusScript

Language construct differences

Construct	*Usage in Macintosh*
ChDir	Macintosh hard drive specifications are supported; for example, "Hard drive:folder1: folder2:". DOS drive specifications, such as "C:\", are not supported.
ChDrive	Generates a run-time error unless the drive argument is the empty string (""), signifying the default drive. To change the drive, use ChDir.
Command	Command line arguments are not normally used on the Macintosh. However, if the Lotus software application permits arguments, they are returned.
CurDir	Generates a run-time error unless the drive argument is defaulted or explicitly specified as the empty string (""), signifying the default drive.

continued

Construct	*Usage in Macintosh*
CurDrive	Return the empty string (""), since there are no drive letters on the Macintosh.
Declare	The Pascal calling convention for external function calls is not supported.
Dir	Ignores the attributes Hidden Files, and System. Does not return the directory specifications "." and "..". Returns all files for "*.*", not just those containing ".". Returns only those files ending with a period for "*.", not every file without an extension. If ATTR_VOLUME only is specified, returns the empty string. If any other attribute is specified, ATTR_VOLUME is ignored.
Environ	Returns an empty string. Generates a run-time error only if an illegal argument is passed, such as a variable number greater than the legal limit.
FileLen	Files containing line terminators are smaller than on DOS platforms, because the line terminator is one character, not two.
GetFileAttr	Does not return the following attributes: ATTR_ARCHIVE, ATTR_VOLUME, ATTR_SYSTEM
Len, LenB	Strings that have been read from files containing line terminators are smaller than on DOS platforms, because the line terminator is one character, not two.
Lock	Open files can be manipulated (copied, opened, etc.).
Open	Open files can be manipulated (copied, opened, etc.).
SendKeys	Not supported. Generates a run-time error.
SetFileAttr	Generates a Permission Denied error if passed the attribute ATTR_ARCHIVE or ATTR_SYSTEM.
Unlock	Open files can be manipulated (copied, opened, etc.).

File system differences

Macintosh-style pathnames are assumed unless the pathname contains a backslash. If the pathname contains a backslash, then a DOS-style pathname is assumed.

There are no drive letters on the Macintosh. All devices reside under the root directory. If you use a pathname containing a drive letter, LotusScript may return an error. For the %Include directive, this is a compiler error; for all other uses, this is a run-time error.

Files are not limited to DOS naming rules (8-character name plus 3-character extension).

The Macintosh does not store a default directory for each drive. It maintains only one current directory, not one per drive as in DOS. Drive names can be up to 27 characters in length. This limitation affects ChDir, ChDrive, and CurDir.

The Macintosh does not recognize the directory specifications "." and "..". This limitation affects the Dir function.

The Macintosh does not use the file system attributes Volume, Archive, and System. This limitation affects Dir, GetFileAttr, and SetFileAttr.

Macintosh uses the carriage return (ASCII 13) character as the line terminator. Other platforms use other characters. This difference means that files and strings manipulated with the same LotusScript code but executed on different platforms may have different sizes. For instance, the UNIX platform uses a single character (linefeed) as the line terminator, so text files written on that platform have equal length to those written on Macintosh. Since the Windows platform uses a two-character sequence, text files written there are larger than text files written on Macintosh, given identical source code. This difference affects FileLen, Len, LenB, and LenBP.

Macintosh permits files that are open for reading to be manipulated (copied, opened, etc.) by another application. A file opened for output by LotusScript is locked; other applications cannot open or copy the file, but can move or rename it. Lock and Unlock work only on shared volumes; the file being locked must be on a server or file sharing must be turned on for a local volume ("Sharing Setup" on the control panel). This difference affects Open, Lock, and Unlock.

Other differences

Function aliasing with ordinal numbers (using the Alias clause in the Declare statement) is not possible on the Macintosh PC.

There are no system environment variables on the Macintosh. This limitation affects Environ.

OS/400 platform differences in LotusScript

Language construct differences

Construct	*Usage in OS/400*
ActivateApp	Not supported. Generates a run-time error.
ChDir	A run-time error is generated if LotusScript cannot interpret the argument to ChDir; for example, if a drive letter is specified in the argument.
ChDrive	Generates a run-time error unless the drive argument is an empty string (“”), signifying the default drive.
CreateObject	Not supported. Generates a run-time error.
CurDir, CurDir$	Generates a run-time error unless the drive argument is the empty string (“”), signifying the default drive.
CurDrive, CurDrive$	Returns the empty string (“”), because there are no drives on an iSeries server.
Date, Date$	Changing the date on OS/400 through LotusScript is not supported. Generates a run-time error.
Declare	The Pascal calling convention for external function calls is not supported. All external function calls must use the CDECL calling convention. In addition, you must use the _System linkage keyword when passing arguments other than pointers.
Dir, Dir$	Ignores the attributeMask argument and behaves as if all files have the attribute Normal. Returns all files for “*.*”, not just those containing “.”. Returns those files ending with a period for “*.”, not every file without an extension.
FileLen, Len, LenB, LenBP, LOF	Strings containing line terminators are smaller than on DOS/Windows platforms. The line terminator is one character (line feed), not two. Therefore, the return value of these functions will be smaller for strings on OS/400 than on Windows.
GetFileAttr	Generates a run-time error if a drive letter is included in the argument. Does not return the following attributes: ATTR_HIDDEN, ATTR_ARCHIVE, ATTR_VOLUME, ATTR_SYSTEM.
GetObject	Not supported. Generates a run-time error.
Input #, Input, Input$, InputB, InputB$, Line Input, Print, Write #	Compiled scripts using these constructs may be platform specific, because file data is stored in a platform-specific manner. OS/400 character set, byte order, line terminator, and numeric precision specifics may affect the portability of scripts using these functions.

continued

Construct	*Usage in OS/400*
IsObject, IsUnknown	See "Other differences."
Open, Lock, Unlock	Explicit or implicit file locking is not supported. This implies the following: • LotusScript for OS/400 allows the user to do operations (Such as copy or open) on a file that is already opened for reading. Thus, the Name statement works differently with OS/400. • The Open statement can specify only Shared as its lock status. Lock Read, Lock Write, and Lock Read Write will cause a run-time error. • The Lock and Unlock statements will cause a run-time error.
SendKeys	Not supported. Generates a run-time error.
SetFileAttr	Ignores the attributes ATTR_HIDDEN, ATTR_ARCHIVE, and ATTR_VOLUME.
Shell	Window styles are ignored.
Time, Time$	Changing the time on OS/400 through LotusScript is not supported. Generates a run-time error.

File system differences

There are no drive letters on an iSeries server. If you use a path name containing a drive letter, LotusScript may return an error.

OS/400 uses the slash (/) character as the directory separator, while DOS/Windows use the backslash (\) character. LotusScript supports use of both the slash and backslash, with the following restrictions:

- A Script compiled on any platform other than OS/400 or UNIX that uses a backslash in a path name string literal will not work on the iSeries server.
- LotusScript cannot use file names (in contrast to path names) that contain the backslash character, because this character is always a path separator on other platforms.

Other differences

Function aliasing with ordinal numbers (using the Alias classes in the Declare statement) is not possible with OS/400.

Where wild cards are permitted in file path strings, LotusScript supports the use of UNIX regular expressions in addition to the "*" and "?" characters. However, using regular expressions in file path strings makes the script platform dependent.

OLE is not supported on LotusScript Release 3.1 for OS/400. This difference affects the CreateObject, GetObject, IsObject, and IsUnknown functions. The CreateObject and IsObject functions will raise run-time errors when executed on OS/400 platforms. The IsObject function can determine if a variable refers to a native or product object, but not an OLE object, because OLE objects do not exist on the OS/400 platform. The IsUnknown function always returns FALSE on OS/400, because there is no way for a Variant expression to receive the V_UNKNOWN value.

When passing pointer arguments to C functions, be aware that the pointer size on OS/400 is 16 bytes, not 4 bytes.

Appendix C
LotusScript/REXX Integration

This appendix provides an overview of REXX integration in the LotusScript language.

When you use LotusScript in OS/2, you can use the LTSRXO10.DLL LSX to invoke applications written in the REXX (the OS/2 Procedures Language, 2/REXX). LotusScript and REXX integration allows LotusScript to send values to a REXX application and use REXX functionality to manipulate the return value. When you use LotusScript and REXX together, line items take the form of function-type calls. For example, you can execute a single REXX statement using REXXFunction or execute an external REXX command file with REXXCmd.

For complete information on REXX and LotusScript integration, see the online help available when you are using LotusScript in OS/2.

Appendix D
LotusScript Aliases

This appendix lists the LotusScript aliases and their equivalent text.

An alias is an alternate spelling of a language keyword (usually VB compliant) such as “MsgBox” for the LotusScript “MessageBox” function.

LotusScript Syntax	*Alias*
ActivateApp	AppActivate
CDat	CVDate
DataType	VarType
DateNumber	DateSerial
GetFileAttr	GetAttr
Implode	Join
MessageBox	MsgBox
NoCase (Option Compare)	Text (Option Compare)
Option Declare	Option Explicit
SetFileAttr	SetAttr
StrCompare	StrComp
TimeNumber	TimeSerial
Yield	DoEvents

Appendix E
Charset Names

This chapter lists the acceptable MIME charset values for the *Charset* parameter of the Open statement.

See the Open statement for usage.

Note The MIME names are case insensitive.

Charset Name	*Description*
UTF-8	8 bit encoding of ISO-10646.
UTF-16LE	16 bit little endian encoding of ISO-10646.
UTF-16BE	16 bit big endian encoding of ISO-10646.
UTF-16	16 bit platform native byte order encoding of ISO-10646.
Windows-1250	Windows character set for Central European languages.
Windows-1251	Windows character set for Cryllic languages.
Windows-1252	Windows character set for Western European languages.
Windows-1253	Windows character set for Greek.
Windows-1254	Windows character set for Turkish.
Windows-1255	Windows character set for Hebrew.
Windows-1256	Windows character set for Arabic languages.
Windows-1257	Windows character set for Baltic rim languages.
Windows-1258	Windows character set for Vietnamese.
Windows-874	Windows character set for Thai.
ISO-8859-1	ISO character set for Western European languages.
ISO-8859-2	ISO character set for Central European languages.
ISO-8859-3	ISO character set for Esperanto and Maltese.
ISO-8859-4	ISO character set for Baltic rim languages.
ISO-8859-5	ISO character set for Cryllic languages.
ISO-8859-6	ISO character set for Arabic languages.
ISO-8859-7	ISO character set for Greek.
ISO-8859-8	ISO character set for Hebrew.

continued

Charset Name	*Description*
ISO-8859-9	ISO character set for Turkish.
ISO-8859-15	ISO character set for Western European languages.
KOI8-R	Cryllic character set.
Shift_JIS	Japanese character set.
EUC-JP	Japanese character set.
EUC-KR	Korean character set.
Big5	Traditional Chinese character set.

MIME charset names (continued)

Charset Name	*Description*
GB2312	Simplified Chinese character set.
GB18030	Simplified Chinese character set.
LMBCS	Lotus Multi Byte Character Set.

Note With Release 6, ISO-8859-10, the ISO character set for Nordic languages is not supported.

EBCDIC charset names

Charset Name	*Description*
IBM037	US and Canadian English, Dutch, Protuguese
IBM273	German
IBM277	Danish, Norwegian
IBM278	Finnish, Swedish
IBM280	Italian
IBM284	Spanish
IBM285	International English
IBM297	French
IBM420	Arabic
IBM424	Hebrew
IBM500	Intl. Eglish, Latin-1, Albanian, Belgian English, French
IBM838	Thai
IBM870	Latin-2, Croatian, Czech, Hungarian, Polish
IBM871	Icelandic
IBM875	Greek

continued

Charset Name	*Description*
IBM1025	Bulgarian, Russian, Serbian Cyrillic
IBM1026	Turkish
IBM1047	Latin-1 Open Systems
IBM1112	Latvian, Lithuanian
IBM1122	Estonian
IBM930	Japanese Katakana
IBM933	Korean
IBM935	Simplified Chinese
IBM937	Traditional Chinese
IBM939	Japanese Latin
IBM1388	Simplified Chinese

Appendix F
Compile-time Error Messages

This chapter describes the compile-time error messages in the LotusScript language.

DELETE not valid on: <*name*>
You used the Delete statement on one of the following:

- A variable that is not an object reference variable
- A variable of type Variant that does not contain an object reference
- The return value of a function
- A property

Assign the object to an object reference variable and apply Delete to the variable instead.

Too many nested INCLUDEs
You have more than 16 levels of nested %Include directives. This may be due to circular %Include references.

Reduce the number of nested %Include directives to 16 or fewer. Remove any circular %Include references.

File contains too many source lines
The source file contains too many lines.

Split the source file into two or more files.

Illegal OPTION BASE after array declaration
The Option Base statement appeared after an array declaration or after a ReDim statement.

Move the Option Base statement so that it precedes all array declarations and ReDim statements.

Illegal OPTION DECLARE after implicit declaration
You used an implicit declaration before the Option Declare statement.

Move the Option Declare statement so that it appears before all variable declarations.

Too many items specified in input/output statement

More than 255 items were specified in one of the following:

- A Print statement
- A Write statement
- An Input statement

Reduce the number of items to fewer than 256.

Illegal value for OPTION BASE

The following conditions could have caused this error:

The element following the Option Base statement is not an Integer constant.

The value of the constant is not 0 or 1.

Change the element following the Option Base statement to an Integer constant whose value is 0 or 1.

Too many labels specified in ON...GOTO statement

More than 255 labels were specified in an On...GoTo statement.

Reduce the number of labels to fewer than 256.

SUB NEW arguments do not match parent's SUB NEW arguments

The parameters in the derived class's Sub New differ in number or type from the parameters in the base class's Sub New. For example:

```
Class Baseclass
   Sub New (X As Long)
   End Sub
End Class
Class Derivedclass As Baseclass
   Sub New (X As Long, Y As Long) ' Illegal, because Y is not
                                  ' a parameter
                                  ' in Baseclass's Sub New.
   End Sub
End Class
```

Do one of the following:

- In the derived class's Sub New declaration, specify which arguments to pass to the base class's Sub New, for example as follows:

  ```
  Class Derivedclass As Baseclass
     Sub New (X As Long, Y As Long), Baseclass (X)
     End Sub
  End Class
  ```

- Redefine the derived class's Sub New so that its parameters match those of the base class's Sub New. For example:

```
Class Derivedclass As Baseclass
   Sub New (X As Long)
   End Sub
End Class
```

Name previously declared: *<name>*

A name that has already been declared in the current scope is being declared again in the same name space. Names that reside in the same name space may only be declared once in a scope. Each module, sub, function, property, class, and user-defined data type has a particular scope. LotusScript has three separate name spaces:

- Variable, Const, Sub, Function, and Property names
- Type and Class names
- Labels

For example, a module-scope variable may have the same name as a class defined in that module, because variable names and class names are in different namespaces and therefore don't conflict. However, a module-scope variable may not have the same name as a function defined in that module.

The name space where a name resides doesn't depend on whether the name is declared Public, Private, or external (declared by the external Declare statement). All of these share the same name space.

Remove the duplicate declaration.

Illegal name for class or type: *<name>*

You used the word Object as the name of a user-defined class or data type. Object is a LotusScript reserved word.

Change the name of the user-defined class or data type.

Class is not a parent of this class: *<class name>*

Either of the following conditions could have caused this error:

- A class specified in a Sub New declaration is not the class from which this one is derived.
- A class specified using "dotdot" notation is not the class from which this one is derived.

For example:

```
Class BaseClassOne
   Sub New (X As Integer)
   End Sub
End Class
Class BaseClassTwo
   Sub PrintIt
      ' ...
   End Sub

End Class
Class DerivedClass As BaseClassOne
   Sub New(Y As Integer), BaseClassTwo(x%)
' Illegal because BaseClassTwo is not the base
' class from which DerivedClass is derived.
' The appropriate base class is BaseClassOne.
   End Sub
   Sub PrintIt
      ' ...
   End Sub
   Sub CallPrintIt
      Call BaseClassTwo..PrintIt
' Illegal because BaseClassTwo is not the base
' class from which DerivedClass is derived.
' The appropriate base class is BaseClassOne.
   End Sub

End Class
```

Correct the reference to the base class.

Public symbol is declared in another module: *<name>*

A name declared as Public has already been declared as Public in another loaded module. A name can be declared as Public in only one loaded module at a time. Other loaded modules can only reference that name.

Remove Public from the declaration, or change the Public name so that it does not conflict with the name in the already loaded module.

Member is not a subprogram: *<member name>*

You used "dotdot" notation to refer to a member variable of a base class. This notation is legal only for referring to a member function, sub, or property of a base class. It is not legal for referring to member variables of a base class.

Refer to the variable by its name only.

Illegal executable code at the module level
An executable statement appears at the module level. The product in which you are running LotusScript does not allow executable statements at the module level.

Move the executable statement into a procedure. If you want the statement to be executed when the module is loaded, move the statement into the Initialize sub. If you want the statement to be executed when the module is unloaded, move the statement into the Terminate sub.

Illegal PUBLIC instance of PRIVATE class or type: *<instance name>*
You declared a Public instance of a Private user-defined data type or class, or a Public function or property that returns an instance of a Private class.

Make the class or type Public, or make the instance Private.

Illegal type suffix on name: *<name>*
You appended a data type suffix character to one of the following:

- The name of a user-defined data type
- The name of a class
- A sub name
- A label
- A product event name

Suffix characters are not valid on these names.

Remove suffix characters from any names on which they are invalid.

ISELEMENT argument is not a list or variant: *<name>*
The first argument that you passed to the IsElement function is not the name of a list or the name of a variable of type Variant holding a list.

Change the argument to a list or a Variant holding a list, or remove the call to the IsElement function.

Illegal scope for PUBLIC or PRIVATE on: *<name>*
You used the Public or Private keyword in a declaration within a sub, function, or property. The Public and Private keywords are not legal in declarations in subs, functions, or properties. Public and Private only have meaning in declarations in module scope or within the definition of a user-defined class.

Remove the Public or Private keyword from the declaration.

Illegal constructor clause on: *<sub name>*

You specified a constructor clause on a sub that is not a class constructor sub (Sub New). For example:

```
Class BaseClass
   Sub New (X As long)
   End Sub
End class
Class DerivedClass As BaseClass
   Sub Old (X As Long, Y As Long), BaseClass(X) ' Illegal: Old
is not a
                                                ' constructor
sub.
   End Sub
End Class
```

A class constructor sub must be a part of the definition of a class, and must be named New.

If the sub is not intended to be a class constructor, remove the constructor clause (that is, the comma, the name of the class, and the argument list). Otherwise, rename the sub to New.

Parent SUB NEW has arguments, SUB NEW is required for: *<class name>*

You defined a derived class that has no Sub New. If the corresponding base class's Sub New requires arguments, the derived class must have a Sub New that provides those arguments. For example:

```
Class BaseClass
   Sub New (X As Integer)
   End Sub
End Class
Class DerivedClass As BaseClass
End Class
Dim ObjRefVar As New DerivedClass     ' Illegal because
BaseClass's
                                      ' Sub New needs to be
passed an
                                      ' integer.
```

Define a Sub New for the derived class whose signature includes the arguments required by the base class's Sub New. For example:

```
Class BaseClass
   Sub New (X As Integer)
   End Sub
End Class
Class DerivedClass As BaseClass
   Sub New (X As Integer)
```

```
   End Sub
End Class
Dim ObjRefVar As New DerivedClass(5)     ' Legal
```

Illegal USE or UseLSX statement after declaration

You used a Use or UseLSX statement after an implicit declaration.

Move the Use or UseLSX statement so that it precedes all implicit declarations.

Member declared in a parent class

You tried to declare a member variable in a derived class using the same name as a member variable, sub, function, or property of the base class. This is not allowed.

The name space for variables also includes functions, subs, and properties. This means that if a name is used as a method name in a base class, it may not be used as a variable name in a derived class. For example:

```
Class BaseClass
   X As Integer
   Sub Y
   ' ...
   End Sub
End Class
Class DerivedClass As BaseClass
   X As Integer                    ' Illegal
   Y As Integer                    ' Illegal
   ' ...
End Class
```

Declare the variable using a different name.

Event handler must be a LotusScript SUB or FUNCTION: *<handler name>*

The handler name specified in an On EventLSAZ_ON_EVENT_Statement statement is not the name of a LotusScript sub or function. An event handler may not be an external (C) function or a product object method.

Member of PUBLIC class or type is instance of a PRIVATE class or type: *<member name>*

Within the definition of a Public class or user-defined data type, you declared as PublicLSAZ_DIM_Statement a member variable that refers to a Private class or user-defined data type, or you included a Public method that returns an instance of a Private class or user-defined type. For example, in the following code, the definition of the variable B produces this error condition:

```
Private Type MyType
   A As Integer
End Type
Public Class MyClass
   Public B As MyType   ' Illegal because MyType is defined as
Private
End Class
```

Change the Public class or user-defined data type to Private, or the Private class or user-defined data type to Public.

FORALL alias variable was previously declared: *<name>*

You used a previously declared variable as a ForAll reference variable. Previously declared variables may not be used as ForAll reference variables. A ForAll reference variable may only be used in a ForAllLSAZ_FORALL_Statement statement.

Rename the ForAll reference variable.

FORALL alias variable already in use: *<variable name>*

You used a previously declared variable as a ForAll reference variable. Previously declared variables may not be used as ForAll reference variables. A ForAll reference variable may only be used in a ForAllLSAZ_FORALL_Statement statement.

Rename the ForAll reference variable.

CASE ELSE must be the last CASE in a SELECT statement

You used a Case clause after Case Else in a Select CaseLSAZ_SELECT_CASE_Statement statement. No other Case clause may follow a Case Else clause.

Make Case Else the last clause in the Select Case statement, or omit the keyword Else.

TYPE declaration has no members

You have a TypeLSAZ_TYPE_Statement declaration with no members. A Type declaration must contain at least one variableName As dataType statement.

Add at least one member to the Type declaration, or remove the Type declaration.

Declaration of external subprogram is not legal inside a class

You tried to use a Declare statement inside a class definition to declare an external C function. This is not allowed.

Move the declaration of the external function to the module level.

Illegal use of array or list element as FORALL target

You used an array or list element as the target of a ForAll statement.

To iterate over an array or list, use the array or list name only. For example:

```
Dim Y List As String
' ...
ForAll X In Y(1)        ' Illegal. Target is an array or list
element.
ForAll X In Y           ' OK. Target is an entire array or
list.
```

Illegal use of property: *<property name>*

You tried to use the named property as one of the following:

- The target in a Get or Put statement
- The target in an Input # or Line Input # statement
- The target in an LSet, RSet, or Mid statement

You must use a variable, not a property, for any of these purposes.

This error also occurs when the property appears with a subscript as the target of an assignment statement. For example:

```
Dim privateArray(1 To 2) As String
Property Set MyProperty As Variant
   privateArray(1) = MyProperty(1)
   privateArray(2) = MyProperty(2)
End Property
Property Get MyProperty As Variant
   MyProperty = privateArray
End property

MyProperty(1) = "Fred"    ' Produces error
```

To assign values to MyProperty, assign it a whole array:

```
Dim anArray(1 To 2) As String
anArray$(1) = "Fred"
MyProperty = anArray
```

Wrong data type for argument <*argument name*> in event handler <*event handler name*>
You specified a procedure as the handler in an On Event statement. The declared data type of a parameter in the definition of that procedure does not match the data type of the corresponding parameter specified when the event was registered with LotusScript.

Refer to the documentation of the product in which you are running LotusScript for information about the arguments that the event handler requires. Change the declared data type of the parameter in the subprogram definition to match the registered data type of the corresponding parameter.

Maximum array dimensions (8) exceeded: <*array name*>
You either declared an array with more than eight dimensions or you used more than eight subscripts in referring to an array. An array can have a maximum of eight dimensions.

If the problem is that the declaration of the array specifies more than eight dimensions, reduce the number of dimensions in the declaration to at most eight. If the problem is that you used more than eight subscripts in a statement referring to an array, reduce the number of subscripts in that statement.

Illegal array bound for: <*array name*>
The following conditions could have caused this error:

- One of the array bounds specified does not evaluate to an integer constant. The range of an integer constant is -32768 to 32767 (inclusive).

  Specify the bound so that it evaluates to between -32768 and 32767.

- In one of the specified array dimensions, the lower bound is greater than the upper bound. The lower bound must be less than or equal to the upper bound.

  Respecify the lower bound or the upper bound.

Array size exceeds maximum: *<array name>*

You declared an array whose total size is greater than the maximum allowable size. The maximum allowable array size is 65,536 bytes (64K).

Reduce the array size to 65,536 bytes or less. The size is calculated as (number of elements) * (size of each element in bytes).

Illegal specification of array bounds for: *<array name>*

You included array bounds in specifying a parameter in the declaration of a sub or function. A parameter that is an array should contain empty parentheses only.

Specify the parameter with empty parentheses. For example:

```
Function Comper (X(5,2) As Integer) As Single    ' Illegal
Function Comper (X () As Integer) As Single      ' Corrected
form
```

Declaration not valid in TYPE scope: *<name>*

You declared one of the following as a member of a user-defined data type:

- An object reference variable
- A list variable
- A dynamic array variable

Object reference variables, list variables, and dynamic array variables are not valid members of a user-defined data type.

Remove the invalid member declaration.

Statement is illegal in TYPE block: *<keyword>*

You used an illegal statement in a Type...End Type block. The only legal statements in a Type...End Type block are declarations of variables without the leading keyword Dim, Public, Private, or Static; the Rem statement; and the directives %Rem...%End Rem and %Include. All other statements are illegal.

By extension, when you use the %Include directive in a Type...End Type block, the file to which it refers must not contain any statements that are illegal inside a Type...End Type block.

Remove the statement from the Type...End Type block.

Statement is illegal in CLASS block: *<keyword>*
You used an illegal statement in a Class...End Class block.

The only legal statements in a Class...End Class block are:

- Declarations of variables without the keyword Dim or Static

  A variable may be declared Public or Private, or with no leading keyword
- Definitions and forward declarations of subprograms, without the keyword Static
- Definitions of the constructor and destructor subs (Sub New and Sub Delete) for the class
- The Rem statement
- The directives %Rem...%End Rem and %Include

By extension, when you use the %Include directive in a Class...End Class block, the file to which it refers must not contain any statements that are illegal inside a Class...End Class block.

Remove the illegal statement from the Class...End Class block.

TYPE may not have instance of itself as a member: *<instance name>*
You declared an instance of the user-defined data type being defined as a member of itself. The definition of a user-defined data type may include an instance of another user-defined data type as a member, but not an instance of itself. For example:

```
Type MyFirstType
   X As Integer
End Type
Type MySecondType
   Y As MyFirstType        ' This is legal
   Z As MySecondType       ' This is illegal
End Type
```

Remove the invalid member declaration.

Out of memory
You must free enough memory to perform the operation that caused this error message. To free memory in your computer, do one of the following:

- If you have other programs in memory, end one or more of those programs.
- Reduce the amount or size of Public data.
- Activate extended memory.

Size of data cannot exceed 64K in this scope
The data in the enclosing scope (module or class) exceeds the limit of 64K bytes.

Split the enclosing scope into multiple units, each with less than 65536 bytes of data.

Size of data cannot exceed 32K in this scope
The data in the enclosing scope (sub, function, or property) exceeds the limit of 32K bytes.

Split the enclosing scope into multiple units, each with less than 32768 bytes of data.

Illegal constant expression for: *<CONST name>*
One of the following occurred in a Const statement:

- You used a value of a data type that does not match the data type suffix character of the constant. If the constant and the value are both numeric, the value may be too large for the data type of the constant.

  Change the constant's data type suffix character, or change the value so that it is legal for the constant's data type.

- You tried to define a constant with a nonconstant value. The value assigned by a Const statement must be a constant value; that is, one of the following:

  - A literal
  - A constant previously defined by a Const statement
  - A built-in function whose arguments are constant expressions
  - An expression whose operands are either literals; constants previously defined by Const statements; or one of a number of built-in functions whose arguments are constant expressions

  Change the assigned value to a constant value.

Arguments not legal in declaration of: *<sub name>*
The following conditions could have caused this error:

- You specified parameters in a Sub Initialize or Sub Terminate definition. Because the Initialize and Terminate subs are executed automatically on module load and unload, they cannot be passed arguments.

  Redefine the sub without parameters.

- You specified parameters in a Sub Delete definition. Because the Delete sub is executed automatically when an object reference is deleted, it cannot be passed arguments.

  Redefine the sub without parameters.

Undefined label: <*label name*>
The sub, function, or property just compiled contains a reference to a label that was never defined. The line number of the error message identifies the End Sub, End Function, or End Property statement that marks the end of the offending procedure. Labels must be defined within the same scope in which they are referenced.

Define the label in the sub, function, or property that refers to it.

Illegal data type for argument: <*argument name*>
You used a fixed-length string as a parameter in the declaration of a sub or function. Fixed-length strings are not legal as parameters in subs or functions.

Change the parameter's data type to String or Variant.

Too many arguments for: <*subprogram name*>
You specified more than the limit of 31 parameters in the declaration of a sub or function. The maximum number of parameters that may be specified for a sub or function is 31.

Reduce the number of declared parameters to 31 or fewer.

Cannot subclass: <*class name*>
You specified a product class as the base class of a derived class. A product class may not be used as the base class of a derived class.

Remove the As *BaseClassName* clause in the class declaration, or specify a LotusScript class as the base class.

Derived class may not be PUBLIC when parent is PRIVATE: <*class name*>
You defined a Public class whose base class is Private. The base class from which a Public class is derived cannot be Private.

Change the definition of the base class to Public, or change the derived class to Private.

Illegal use of NEW or DELETE

You used the name New or Delete to name a function, property, or variable within a class definition. Within a class, the names New and Delete are reserved for subs; they may not be functions, properties, or variables.

Rename the function, property, or variable. To specify a sub to be executed on the construction or deletion of an object, include a Sub New or Sub Delete in the class definition.

DIM required on declarations in this scope

You declared a variable at module level without the Dim, Public, or Private keyword, or you declared a variable inside a procedure without the Dim or Static keyword. One of these is required.

Add the appropriate keyword to the declaration.

Illegal PRIVATE declaration of: *<name>*

In defining a class, you declared the Sub New or Sub Delete as Private. New and Delete subs may not be declared as Private.

Remove Private from the declaration of the New or Delete sub.

Illegal PUBLIC declaration of: *<name>*

You declared the Initialize or Terminate sub as Public. The Initialize and Terminate subs may not be declared as Public.

Remove Public from the declaration of the sub.

Name was forward declared as something else: *<name>*

You named a function, sub, or property in a Declare statement and then used that name in the definition of a different kind of procedure. For example:

```
Declare Sub MyProcedure

Property Set MyProcedure   ' Illegal because you previously
                           ' declared MyProcedure as a sub
   ' ...
End Property
```

Change the declaration or its corresponding definition so that both are either functions, subs, or properties.

Duplicate forward declaration: *<name>*
You have used a Declare statement twice to declare the same function, sub, or property in this scope.

Remove one or the other of these Declare statements.

Storage class or visibility does not match forward declaration: *<subprogram name>*
You declared a function, sub, or property with a Declare statement and then defined the procedure with a Function, Sub, Property Set, or Property Get statement. The definition differs from the declaration in one or another of the following respects:

- The declaration contains the keyword Static but the definition doesn't, or vice versa. (The keyword Static specifies that the storage class of the procedure's variables will be static by default.)
- The procedure is declared as Public but defined as Private, or vice versa.

Change the declaration or the corresponding definition of the procedure so that they match.

Return type does not match forward declaration: *<function name>*
You have declared a function or property with a Declare statement and then defined it with a Function, Property Set, or Property Get statement. The data type that you specified as the procedure's return value in the Declare statement is different from the data type you specified as the return value in the definition statement. For example:

```
Declare Property Set MyProperty As Integer
' ...
Property Set MyProperty As Double  ' Illegal because
                                   ' MyProperty's return
' ...                              ' value was already
                                   ' declared as Integer
End Property
```

Change the data type of the return value in the declaration or in the corresponding definition so that they match.

Number of arguments does not match forward declaration: *<subprogram name>*
You declared a function or sub with a Declare statement, and then either of the following happened:

- You defined the function or sub with a Function or Sub statement specifying a different number of parameters than you specified in the Declare statement. For example:

```
Declare Function MyFunction(X As Integer, Y As Double) As
Integer
' ...
Function MyFunction(X As Integer) As Integer  ' Illegal
     ' because Declare specified two parameters
End Function
```

  Make the parameters in the declaration and definition match each other.

- The procedure that you forward declared is a parameterized constructor sub (Sub New) inside a Class statement and you have not defined a Sub New for that class.

  Either remove the Declare statement or define a corresponding Sub New.

Argument does not match forward declaration: *<argument name>*
You declared a function or sub with a Declare statement and then defined it with a Function or Substatement. Either of the following conditions could have caused the error:

- The data type of the indicated parameter in the procedure definition is different from the corresponding parameter in the procedure declaration. For example:

```
Declare Sub MySub(X As Integer)
' ...
Sub MySub(X As Double)  ' Illegal because X was previously
' ...                   ' declared to be of type Integer
End Sub
```

  Change the data type of the indicated parameter in the declaration or the definition of the procedure so that they match.

- The data type of the indicated parameter matches the data type of the corresponding parameter in the procedure declaration, but the parameters represent different kinds of data structure. For example:

```
Declare Function MyFunction(X() As Integer) As Integer
' ...
Function MyFunction(X As Integer) As Integer    ' Illegal
     ' because X is a scalar variable but X() is an array

End Function
```

  Change the parameter specification in the declaration or the definition of the procedure so that the two match.

Illegal function return type for: *<function name>*

You either used a Declare or Function statement to declare or define a function and specified its return type as a fixed-length string or a user-defined data type, or else you used a Declare statement to declare an external C function and specified its return type as Variant, Currency, fixed-length String, or a user-defined data type.

Specify a data type other than the ones listed above for the function's return value.

Method was declared as something else in a parent: *<method name>*

You used a Declare statement or a Function, Sub, Property Set, or Property Get statement to declare or define a procedure within the definition of a base class. In subsequently defining a derived class, you declared or defined a function or sub with the same name as the base class's procedure, but the procedure types are different. For example:

```
Class BaseClass
   Function MyProcedure As Integer
   ' ...
   End Function
End Class
Class DerivedClass As BaseClass
   Sub MyProcedure         ' Illegal because MyProcedure is a
                           ' different
   ' ...                   ' kind of procedure in BaseClass

   End Sub
End Class
```

Change the base class procedure or the corresponding derived class procedure so that both are either subs, functions, or properties.

Method signature does not match parent method: *<method name>*
You used a Declare statement or a Function or Sub statement to declare or define a procedure within the definition of a base class. In subsequently defining a derived class, you declared or defined a procedure of the same kind (a function or sub) with the same name as the base class's procedure but with a different signature.

One of the following does not match:

- The return type
- The number of parameters
- The data type of one of the parameters
- The data structure of one of the parameters

Change the signature of the base class procedure or of the corresponding procedure in the derived class so that they match.

PROPERTY GET and SET must have same storage class and visibility
One of the following occurred:

- You declared a property's variables to be Static by default in either the Property Get statement or the Property Set statement, but not in both. The declarations must agree: either both or neither must specify Static.

  Change either statement to agree with the other.

- You declared a property's scope in a Property Get statement differently from the property's scope in the corresponding Property Set statement. The property must have a single scope: either Public or Private.

  Make them both Public or Private.

Illegal property type for: *<property name>*
In declaring or defining a property with a Declare statement or a Property Get or Property Set statement, you specified its data type as either a fixed-length string or a user-defined data type. Properties cannot be fixed-length strings or user-defined data types.

Declare the property as a different data type.

PROPERTY GET and SET must have same data type
You declared a property's data type in a Property Get statement differently from the property's data type in the corresponding Property Set statement. The property must have a single declared data type.

Change the data type in one statement to the data type in the other.

Property was declared as something else in a parent: *<property name>*
You used a Declare statement or a Function or Sub statement to declare or define a function or sub within the definition of a base class. In defining a derived class, you used Declare or a Property Get or Property Set statement to declare or define a property with the same name as the base class's function or sub. For example:

```
Class Baseclass
   Sub MyProcedure
   ' ...
   End Sub
End Class
Class DerivedClass As Baseclass
   Property Set MyProcedure          ' Illegal because
                                     ' MyProcedure is a sub
                                     ' rather than a property
                                     ' in Baseclass.
   End Property
End Class
```

Change the base class's procedure or the corresponding procedure in the derived class so that both are either subs, functions, or properties

Property type does not match parent property: *<property name>*
You used a Declare statement or a Property Get or Property Set statement to declare or define a property within the definition of a base class. In defining a derived class, you used Declare or Property Set or Property Get to declare or define a property with the same name as the one in the base class, but with a different data type. For example:

```
Class BaseClass
   Property Get MyProperty As Integer
   ' ...
   End Property
End Class
Class DerivedClass As BaseClass
   Property Get MyProperty As Double
   ' ...
   End Property          ' Illegal because MyProperty's return
                         ' type was defined as Integer in
                         ' BaseClass
End Class
```

Change the data type of the derived class's property or the corresponding property in the base class so that they match.

Illegal pass by value: <*argument name*>
One of the following happened:

- In declaring or defining a function or sub, you used the ByVal keyword in specifying a parameter that is an array, list, object reference, or user-defined data type. Arrays, lists, instances of user-defined data types, and object references cannot be passed by value, so ByVal is not allowed in the specification of one of these as a parameter. For example:

```
Type MyType
    A As Integer
End Type
Declare Function MyFunction(ByVal X As MyType) ' Illegal
```

  Remove ByVal from the declaration.

- You tried to pass an array, list, object reference, or instance of a user-defined data type by value in a call to a LotusScript procedure. For example:

```
Type MyType
    A As Integer
End Type
Sub MySub(X As MyType)
' ...
End Sub
Dim Z As MyType
MySub(Z)           ' Illegal: this tries to pass by value.
MySub Z            ' Legal: this passes by reference.
```

  or

```
Dim anArray(1 to 3) As String
Sub MySub2(Z As Variant)
' ...
End Sub
MySub2(anArray())  ' Illegal: this tries to pass by value.
MySub2 anArray()   ' Legal: this passes by reference.
```

  Pass the argument by reference. Remove the parentheses around the argument in the calling statement.

Illegal STATIC on: *<name>*
You used the Static keyword in the declaration of one of the following:

- An external C function
- A class member (a variable, property, function, or sub)
- A variable declared inside a class method or property
- A variable declared at module level

Remove the keyword Static from the declaration.

Illegal external argument: *<argument name>*
You declared a C function and specified the data type of one of its parameters as a fixed-length string or as a list. You cannot specify a C function parameter as a fixed-length string or a list.

For a fixed-length string, declare the parameter as type String, Variant, or Any.

For a list, declare the parameter as type Any.

Illegal construction of type instance: *<instance name>*
You used the keyword New in the declaration of a variable of a user-defined data type or in a statement assigning a value to a variable of a user-defined data type. The keyword New is not allowed in referring to variables of a user-defined type. For example:

```
Type MyType
   A As Integer
End Type
Dim X As New MyType     ' Illegal
```

or:

```
Set X = New MyType     ' Illegal
```

You use the keyword New to declare or assign a value to an object reference variable, that is, an instance of a class.

Remove New from the declaration or assignment statement.

Class or type name not found: *<name>*

You used a name that does not refer to an existing class or user-defined data type where one of these was required. You used the name in one of the following contexts:

- A variable declaration, as in:

  ```
  Dim X As ClassName

  Dim X As User-definedTypeName
  ```

- A derived class declaration, as in:

  ```
  Class NewClassName As ClassName
  ```

  `Class ClassName As ClassName` is also illegal even if `ClassName` exists because a class may not be derived from itself.

- A Set statement, as in:

  ```
  Set X = New ClassName
  ```

- A base class reference in a derived class method, as in:

  ```
  Call ClassName..MethodName
  ```

- A Bind statement (product classes only), as in:

  ```
  Set X = Bind ClassName (objectName)
  ```

Declare the class or user-defined data type before you refer to it.

Illegal range specifier

You used a Def*type* range in one of the following illegal ways:

- No range was specified.
- The beginning of the range was not a single character between A and Z (ASCII uppercase or lowercase), inclusive.
- The end of the range was not a single character between A and Z (ASCII uppercase or lowercase), inclusive.

Correct the error and recompile.

Illegal DEF*type* statement after declaration

A Def*type* statement is located in the wrong part of the module. Def*type* statements must appear before all declarations (both explicit and implicit) in the module.

Move the Def*type* statement so that it precedes the first declaration in the module.

Duplicate range specifier

You included a letter in a Def*type* range that is already included in another Def*type* range in the same module. Once a letter has been included in a Def*type* range, it may not be included in another Def*type* range in the same module. For example:

```
DefInt A-D
DefInt D-G   ' Illegal: D already belongs to a range.
```

If a Def*type* -z has been specified in a module, no other Def*type* range may be specified in that module.

Redefine your Def*type* ranges so that no letter is included in more than one range.

Label is illegal outside of a subprogram

You defined a label at the module level. Labels may not be defined at the module level. Executable statements at the module level are executed as the module is compiled, and then discarded. Therefore, control cannot be transferred to a labeled statement at the module level.

Remove the label, or the entire labeled statement. Revise the script to remove any attempted transfer of control to the labeled statement.

Error number must be INTEGER constant: *<name>*

You used a name as an error number in an On Error statement, but it is not a constant of type Integer. A name used as an error number in an On Error statement must be a constant of type Integer.

Define the name as an integer constant (with the Const statement), or use an integer numeric value. If the name is the name of a LotusScript error constant, use %Include to include the file LSERR.LSS in your module.

Error number must be INTEGER

You used a numeric constant as an error number in an On Error statement, but it is not an integer. The value of a constant used as an error number in an On Error statement must be an integer.

Change the numeric constant to an integer.

Illegal ON ERROR statement

In an On_Error...GoTo statement, the element that follows the GoTo keyword is neither a label nor an integer constant equal to zero, which is what is required.

Change the element following the GoTo keyword to a label or to an integer constant equal to zero.

Statement is illegal outside of a subprogram

You used a statement that is not legal at the module level. These statements include:

- End statement
- Execute statement
- GoSub statement
- GoTo statement
- If...GoTo statement
- On Error statement
- On...GoTo statement
- On...GoSub statement
- Resume statement
- Return statement
- SendKeys statement
- Yield statement

Revise the script to remove any of these statements at the module level.

Not a product class: *<name>*

You used a user-defined class name in the following statement:

```
Set X = Bind ClassName (ObjectName)
```

The class name used in a Set...Bind... statement must be a product class name, not a user-defined class name.

Change the class name following the Bind keyword to a product class name, or remove the statement.

Not a product class instance: *<name>*

Where a reference to a product object was expected in an On Event statement, you used the name of something else. In an On Event statement, the name specified in the From clause must be a product object reference variable, the name of a function or property that returns a product object reference, or a Variant that holds a product object reference.

If a product object was intended, make the reference be to the intended product object. Otherwise, remove the statement.

Not an event name: *<name>*

Where an event name was expected in an On Event statement, you specified a name that is not an event name. Event names are registered with product classes.

Change the name to a product event name, or remove the statement.

Not a sub or function name: *<name>*

In a statement where the name of a function or sub is expected, you specified a name that is not recognized as a sub or function name. The statement is one of the following:

- A Call statement
- A call without the Call keyword (for example, a statement consisting of a name)

If the sub or function has not been defined before being called from within a procedure, use the Declare statement to forward declare it. You must define a sub or function before calling it at module level.

Illegal REDIM on: *<name>*

You used the ReDim statement on a name that is not the name of a dynamic array. For example:

```
Dim anArray(1 To 2) As Integer
ReDim anArray(1 To 3)           ' Illegal because anArray was
                                ' previously declared as a
                                ' fixed array.
```

Either replace the name in the ReDim statement with the name of a dynamic array, or remove the statement.

Illegal RESUME statement

In a Resume statement, you used a numeric to specify the statement at which execution is to continue. If you specify a numeric element in a Resume statement, it must evaluate to zero.

Remove the element or change it to an integer constant or literal with a value of zero. Resume and Resume 0 have the same meaning.

FOR count variable already in use: *<name>*

You used the count variable of an outer For loop as the count variable of an inner For loop. The count variable of an outer For loop may not be reused as the count variable of an inner For loop. For example:

```
For X% = 1 To 10
   For X% = 1 To 5       ' Illegal. X% is already in use.
   ' ...
   Next
Next
```

Change the count variable in one of the For loops so that they are different from each other.

FORALL alias variable is not of same data type: *<name>*

You reused a ForAll reference variable, but the array, list, or collection being iterated over is of a different data type than the collection previously iterated over using the same variable. For example:

```
Dim X(10) As Integer
Dim Y(10) As Long

ForAll I In X
  ' ...
End ForAll

ForAll I In Y     ' Error. I is an Integer above;
                  ' it can't be Long here.
End ForAll
```

Use a different variable: either an existing ForAll reference variable of the correct type, or a new variable.

FOR count variable must be a scalar variable: *<name>*

The count variable of a For statement must be a scalar variable. The variable cannot be an array or list variable, or an element of an array or list. Its type cannot be a user-defined type or a class; and it cannot be a member of a user-defined type or of a class. It cannot be a property, a function, or a constant.

Change the For count variable to a scalar variable.

Illegal type suffix on FORALL alias variable: *<name>*
The ForAll reference variable's declaration or a reference to that variable contains a data type suffix character. Data type suffix characters are not allowed in either the declaration of, or references to, a ForAll reference variable.

Remove the suffix character from the variable's declaration or reference.

Not a PUBLIC member: *<name>*
You referred to a Private member of a class outside of the class's scope. Only Public class members can be referred to outside of their defining class's scope. (By default, member variables are Private, whereas member functions, subs, and properties are Public unless explicitly declared as Private.)

Remove the Private keyword (if any) from the declaration of the class member, and substitute the keyword Public in its place.

Illegal reference to FORALL alias variable: *<name>*
You referred to a name that was previously used as the reference variable in a ForAll reference variable. You referred to that variable outside of the ForAll loop. ForAll reference variables may not be referred to outside of a ForAll loop.

Remove the reference to the variable.

Type suffix does not match data type: *<name>*
You referred to a variable, constant, function, or property with a data type suffix character that does not match its declared data type. If a variable is declared as a Variant, references to that variable may not contain any suffix character.

Change the suffix character to match the declared data type, or remove the suffix character.

Not a member: *<name>*
You referred to a nonexistent member of a class or user-defined data type. For example:

```
Type myType
   A As Integer
End Type
Dim X As myType
X.nonVar% = 10   ' Illegal because nonVar% is not defined in
                 ' myType
```

Define the member within the class or data type definition, or remove the reference.

Variable not declared: *<name>*
You referred to an undeclared variable while the Option Declare statement was in effect. Implicit declarations are illegal when Option Declare is in effect.

Declare the variable, or remove the Option Declare statement.

Illegal single-line IF
A physical end-of-line (with no line-continuation character) appeared before the end of the Then or Else clause in an If...Then...Else statement. For example:

```
If X = Y Then Do : X = X + 1
Loop              ' Illegal. Loop must appear on same line as
                  ' Do.
```

A single-line If...Then...Else statement must be completely contained on one line, including any continuation lines designated by line-continuation characters.

Do one of the following:

- Write the Then clause and the Else clause on the same line as the If.
- Use a line-continuation character.
- Use an If...Then...Else...End If block statement in place of the single-line If...Then...Else statement.

Name does not match FOR count variable: *<name>*
The variable name that immediately follows the Next keyword in a For...Next block does not match the corresponding For count variable.

Match the name with its corresponding For count variable, or remove the name that follows Next: the name is optional.

Not an array, list, collection or variant: *<name>*
The target of a ForAll statement is not an array, list, or collection or a Variant that holds a reference to an array, list, or collection.

Change the target to one of these, or remove the ForAll statement.

ME not valid outside of class scope
You used the keyword Me outside of a procedure within a class. Use the keyword Me only inside procedures within a class. You use Me within the definition of a class when referring to members of that class.

Remove the keyword Me. If you are referring to a class member, use an object reference variable instead of Me.

.. not valid outside of class scope

You used "dotdot" syntax outside of a procedure within a class. The "dotdot" syntax is only valid inside procedures within a class. You use "dotdot" notation when referring to a procedure in a base class when the derived class has a procedure of the same name, as in the following example:

```
Class BaseClass
   Sub MySub
      Print "In BaseClass's MySub"
   End Sub
End Class

Class DerivedClass As BaseClass
   Sub MySub
      Print "In DerivedClass's MySub"
   End Sub

   Sub MyOtherSub
      Call MySub                    ' Print "In DerivedClass's
                                    ' MySub"
      Call BaseClass..MySub         ' Print "In BaseClass's
                                    ' MySub"
   End Sub
End Class
```

Remove the "dotdot" syntax and use an object reference variable in its place.

Reference must contain exactly one subscript: *<name>*

A reference to a list or collection contains either no subscript or more than one subscript. A list or collection reference must contain exactly one subscript.

Specify exactly one subscript in the reference.

Illegal parenthesized reference: *<name>*

You referred to a name followed by parentheses, but the reference is not to an array, list, or a collection, or a Variant containing a reference to one of these, or to a function.

- If the reference is intended to be to one of the above, check the spelling and correct it if necessary.
- If the reference is not intended to be to one of the above, remove the parentheses from the reference.

Wrong number of array subscripts for: *<array name>*
The number of subscripts in an array reference does not match the number of defined dimensions for the array.

Change the number of subscripts to match the number of defined dimensions for the array.

Not an instance name: *<name>*
A name is followed by a dot, but the name is not an object reference variable, a Variant variable containing a reference to an object, or a variable of a user-defined data type. Use "dot" notation only with variables of one of these three kinds.

Replace the name with the name of a valid variable.

Bounds must be specified in REDIM of: *<array name>*
You used the ReDim statement but did not specify the bounds of the array. A ReDim statement must specify bounds.

Specify the bounds within the ReDim statement.

Variable required: *<name>*
In one of the following statements, you used a name that is not the name of a variable, a property, or a ForAll reference variable:

- An assignment statement (Let or =) in either of the following forms:

  Let *name* = ...

  name = ...
- A Set statement in any of the following forms:

  Set *name* = New...

  Set *name* = *ObjectReferenceVariable*

  Set *name* = Bind (*ProductObjectName*)
- A Delete statement
- An Erase statement
- A ForAll statement
- A Get or Put statement
- An Input # or Line Input # statement
- An LSet or RSet statement
- A Mid or MidB statement
- A ReDim statement

In each of these statements, the name must be the name of a variable, a property, or a ForAll reference variable.

Replace the name with a valid name, or remove the invalid statement.

Named product class instance not valid here

In one of the following statements, you used the name of a product object in a context in which it is not allowed:

- An assignment statement (Let or =) in either of the following forms:

  Let *name* = ...

  name = ...

- A Set statement in either of the following forms:

  Set *name* = NEW...

  Set *name* = ...

  Set *name* = Bind...

- A Delete statement
- An Erase statement
- A ForAll statement
- A Get or Put statement
- An Input # or Line Input # Statement
- An LSet or RSet statement
- A Mid or MidB statement
- A ReDim statement

Replace the name with an appropriate name, or remove the invalid statement.

Illegal reference to: *<name>*

You used a name as though it contained or referred to a value, but it doesn't. For example:

```
Sub MySub
   Print "Hello"
End Sub
stringVar$ = MySub     ' Illegal because MySub does not return
                       ' a value
```

Remove this use of the name, or replace it with a name that has a value (for example, a function name instead of a sub name).

Numeric overflow

In defining a constant with the Const statement, you specified a numeric value that is too large for the specified or default data type:

- The value is too large for the data type specified by the value's suffix character.
- If no suffix character is specified, the value is too large for a Double.

For example:

```
Const X = 100000%          ' Illegal because the value is too
                           ' large for the data type Integer

Const Y = 100000!          ' Legal
```

Change the suffix character to match the magnitude of the value, or specify a smaller value.

Numeric underflow

In defining a constant with the Const statement, you specified a numeric value that is too small for the specified or default data type:

- The value is too small for the data type specified by the value's suffix character.
- If no suffix character is specified, the value is too small for a Double.

For example:

```
Const X = .1E-300!         ' Illegal because the value is too
                           ' small for the data type Single

Const X = .1E-300#         ' Legal
```

Change the suffix character to match the magnitude of the value, or specify a larger value.

Illegal numeric constant

You tried to define a numeric constant, assigning it a value that doesn't match the specified or default data type. For example:

```
Const ANINT = 1.2%     ' Illegal because 1.2 is not an Integer
```

Fix the numeric constant.

Illegal product constant: *<name>*

You specified a product constant name that was not recognized by the product.

Check the documentation for the product. Use a correct product constant name (check the spelling), or remove the reference to the product constant.

Name too long: <*name*>
The specified name is too long (it is truncated in the error message). The maximum length of a LotusScript name is 40 characters.

Shorten the name to 40 or fewer characters.

Token is too long
The maximum length of a LotusScript token (a sequence of characters with a unique meaning) may not exceed the maximum allowable length of a string constant (16K characters) plus its delimiters.

Reduce the length of the token.

Declaration may not contain type suffix and data type: <*name*>
You specified a declaration that contains both a data type suffix character and an As *dataType* clause. A declaration may not contain both, even if they match. For example:

```
Dim myInt% As Integer     ' Illegal
```

Remove either the suffix character or the As *dataType* clause from the declaration.

Illegal string length constant for: <*name*>
You specified a length for a fixed-length string as one of the following:

- An item that is not a literal or a constant (created with the Const statement)

  Change the length specifier to a literal or a constant.

- A literal that is not an Integer or Long value, or a constant that does not have an Integer or Long value

  Use an Integer or Long literal, or a constant with an Integer or Long value.

- A value not in the range 1 to 32767

Change the length specifier to a number within this range.

Illegal use of NEW on array or list declaration: <*name*>
You used the keyword New in declaring an array or list. This not allowed. In an array or a list whose type is a class, the elements must be constructed individually.

Remove the New keyword from the declaration of the array or list specified in the error message.

INCLUDE filename must be a string constant

Following the keyword %Include, you specified something other than a quoted literal. For example:

```
Dim myFile As String
myFile$ = "C:\myroot\myfile.lss"
%Include myFile$                    ' Illegal because %Include
                                    ' takes a quoted literal
%Include "C:\myroot\myfile.lss"     ' Correct syntax
```

Use a quoted literal.

Cannot open included file: *<file name>*

One of the following conditions could have caused this error:

- The path or the file name you specified is incorrect.

  Fix the path or the file name, or move the file to the directory specified in the path.

- The file is not in your working directory or in the directory you specified in the path.

  Move the file to your working directory or to the directory you specified in the path.

- The file could not be opened.

  Correct the situation that is preventing you from opening the file.

Unterminated %REM block

You used a %Rem keyword with no corresponding %End Rem. Beginning with the unpaired %Rem, all lines of the script were read as comments.

Insert the corresponding %End Rem.

Unterminated string constant

You omitted the double quotation mark that signals the end of a quoted literal on a single line. Double quotation marks must be paired on the same line. For example:

```
Print "Hi,          ' Illegal because end quotation mark is
                    ' missing.
Martin."

Print "Hi, "  _     ' Legal because string is properly quoted
"Martin."           ' Legal because string is properly quoted
                    ' and preceded by line-continuation
                    ' character
' Output: Hi, Martin.
```

Terminate the string with double quotation marks on the same line where it starts.

Unterminated multiline string

You omitted the vertical bar (|) that marks the end of a multiline string; or you omitted the close brace (}) that marks the end of a multiline string; or you used a brace as one delimiter and the "|" character as the other. For example:

```
Print |Hi,
Martin.
' ...         ' Illegal because there is no matching vertical
              ' bar

Print |Hi,
Martin.}       ' Illegal because the delimiters don't match.
```

Check for any unpaired or improperly paired multiline string delimiters and enclose the string appropriately.

Unterminated square bracket reference

A square bracket reference was not terminated by a close square bracket (]) on the same line. Square brackets are used in some cases when referring to the names of product items.

Terminate the square bracket reference with a close square bracket on the same line. Make sure that the product you are using supports square bracket notation for references.

Illegal character after continuation character

The line-continuation character underscore (_) is followed on the same line by a character that is not the comment character ('). The line-continuation character must be the last character on a line, except for an optional comment, beginning with the comment character.

Remove everything following the line-continuation character on the line, or insert a comment character after it to comment out the rest of the line.

Illegal character after %INCLUDE directive

A %Include directive is followed on the same line by a character that is not the comment character ('). The file name of the file to be included must be the last item on the line, except for an optional comment, beginning with the comment character.

Remove everything following the file name on the line, or insert a comment character following the file name.

SET required on class instance assignment

You attempted to assign an object reference to a variable but omitted the Set keyword. (An object reference can be a reference to an instance of a user-defined class, a product object, an OLE automation object, or the

constant NOTHING.) The Set keyword is required in object reference assignments. For example:

```
Class MyClass
' ...
End Class
Dim MyObj As New MyClass
Dim varV As Variant
varV = MyObj          ' Illegal syntax
```

Insert the Set keyword in the assignment statement:

```
Class MyClass
' ...
End Class
Dim MyObj As New MyClass
Dim varV As Variant
Set varV = MyObj     ' Legal syntax
```

Unterminated <*keyword*> block

You omitted the keyword that marks the end of one of the following block statements:

Class

Do

For

ForAll

Function

If...Then...Else...EndIf

Property Get

Property Set

Select Case

Sub

Type

While

Terminate the statement with the appropriate keyword.

Unexpected: <token>; Expected: <token>

The compiler encountered an unexpected language element.

If the unexpected language element is a number appearing inside square brackets, it represents the ASCII code of an unprintable character. For

example, if you enter the Backspace character in a statement where a name is expected, the following error message appears when you compile the script:

```
Unexpected: [8]; Expected: Identifier
```

For more information, refer to the list of expected language elements following the unexpected language element in the error message.

Parser stack overflow at: *<token name>*

The statement being compiled is too complex. It may contain a complex expression, or deeply nested block statements, such as a Do or For statement.

Reduce the nesting level, or break up the offending statement into multiple, less complex statements.

Unknown statement

The compiler could not parse the statement on the line specified in the error message.

If a statement was intended, check the legal syntax for the statement. If a comment was intended, designate the line as a comment line. Otherwise, remove the incorrect text.

Maximum number of errors reached

The maximum of twenty compilation errors has been reached, causing compilation to stop.

Fix the reported errors and recompile the program.

PROPERTY SET not defined for: *<property name>*

You tried to assign a value to a property, but did not define a Property Set procedure for the property. For example:

```
Dim myInt As Integer
Property Get MyProp As Integer
   MyProp% = myInt%
End Property
MyProp% = 3       ' Illegal because there is no
                  ' Property Set MyProp defined
```

Define a Property Set procedure for the property to which you want to assign a value.

PROPERTY GET not defined for: *<property name>*

You tried to retrieve the value of a property for which you did not define a Property Get procedure. For example:

```
Dim myInt As Integer
Dim myOtherInt As Integer
Property Set MyProp As Integer
    myInt% = MyProp%
End Property
MyOtherInt% = MyProp% ' Illegal because there is no
                      ' Property Get MyProp defined.
```

Define a Property Get procedure for the property whose value you want to retrieve.

Duplicate option

You used the Option Base, Option Declare, or Option Public statements more than once in a module. These statements can only appear once each per module.

Remove any repeated instances of the Option Base, Option Declare, or Option Public statements within the module. To override the lower bound setting specified by the Option Base statement, use explicit lower bounds in a Dim or ReDim statement.

Missing argument for: *<function name>*

The following conditions could have caused this error:

- You did not include a required argument when you called a function. For example:

  ```
  Function MyFunction(A As Integer, B As Integer) As Integer
     ' ...
  End Function
  anInt% = MyFunction%(5)     ' Illegal because MyFunction
                              ' takes two arguments
  ```

  Supply the missing argument in the function call.

- A comma was not followed by an argument. For example:

  ```
  Function MyFunction(A As Integer, B As Integer) As Integer
     ' ...
  End Function
  anInt% = MyFunction(,3)     ' Illegal
  ```

  Remove the comma, or specify the argument.

Expected expression before end of argument list for: *<function name>*

You used a comma before the last optional argument in a call to a built-in function, but you did not supply the argument. For example:

```
myVal% = StrCompare("abc", "abc",)          ' Illegal
```

Remove the comma, or specify the last optional argument:

```
myVal% = StrCompare("abc", "abc")           ' Legal
myVal% = StrCompare("abc", "abc", 1)        ' Legal
```

Wrong number of arguments for: *<name>*

The following conditions could have caused this error:

- You specified the wrong number of arguments when you called a sub or function.

  Change the number of arguments in the sub or function call to the correct number.

- You specified the wrong number of arguments when you called a built-in function.

  For information about the function signature for a specific built-in function, consult the Help topic for that function.

LISTTAG argument is not a FORALL alias variable

You used an invalid argument when you called the ListTag function. The ListTag function may only be passed the ForAll reference variable of the ForAll statement:

```
Dim Y List As String
ForAll X In Y
   Print ListTag(ABC) ' Illegal
   Print ListTag(X)   ' Legal
End ForAll
```

Replace the invalid argument in the ListTag function call with the ForAll reference variable where ListTag appears.

Type mismatch on: *<name>*

The following conditions could have caused this error.

- You tried to pass an argument to a sub or function by reference, but the data types of the argument and the corresponding parameter do not match.

  Pass the argument by value or pass an argument of the correct data type.

- You tried to pass an array, a list, or an object reference to a function or sub, but the corresponding parameter is not defined as one of these or as a Variant.

  Pass an argument of the correct kind.

- You tried to pass a scalar value to a function or sub, but the corresponding parameter is defined as an array, a list, or an object reference variable.

  Pass an argument of the correct kind.

- You tried to assign an instance of a user-defined data type to a Variant. For example:

```
Type myType
   A As Integer
End Type
Dim typeInst As myType
Dim varV As Variant
varV = typeInst      ' Illegal
```

  This is not allowed. Remove the assignment statement.

- You used a Set statement to try to assign a value other than an object reference to an object reference variable (or a Variant holding an object reference). For example:

```
Class MyClass
   ' ...
End Class
Dim X As New MyClass
Dim N As Integer
N% = 5
Set X = N%             ' Illegal
```

  This is not allowed. Remove the assignment statement.

- You used a Set statement to try to assign an object reference to something other than an object reference variable or a Variant. For example:

```
Class MyClass
   ' ...
End Class
Dim X As New MyClass
Dim N As Integer
Set N% = X            ' Illegal
```

  This is not allowed. Remove the assignment statement.

- You used a Set statement to try to assign an object reference variable of one class to an object reference variable of another class. You can only do this when the variables designate instances of the same class or when the target variable designates a base class and the variable whose value is being assigned designates a derived class from that base. For example:

```
Class MyClass
   ' ...
End Class
Class BaseClass
   ' ...
End Class
Class DerivedClass As BaseClass
   ' ...
End Class
Dim A As New MyClass
Dim B As New BaseClass
Dim D As New DerivedClass
Set B = A                ' Illegal
Set D = B                ' Illegal
Set B = D               ' Legal
```

  Remove or revise the assignment.

- You used a Set or Set...New statement to try to create an object (class instance) and assign a reference to it to a variable that is not an object reference variable or a Variant.

```
Class MyClass
   ' ...
End Class
Dim X As New MyClass
Dim N As Integer
Set N% = New MyClass   ' Illegal
```

  Remove or revise the assignment.

- You used a Set or Set...Bind statement in which the target variable is not an object reference variable or a Variant holding an object reference.
- You used a With statement whose target is not an object reference variable or a Variant containing an object reference. The With statement can only be used to operate on objects.
- A ReDim statement contains a data type that does not match the data type in the declaration of the array, or the data type in a previous ReDim statement whose target was that array.

  Change the data type in the ReDim statement so that it matches the data type of the declaration or previous ReDim statement whose target was that array, or remove the data type from the ReDim statement —

once you specify a data type for a dynamic array, it is not necessary to specify the data type again in subsequent ReDim statements.

- You used a variable declared as a non-numeric data type as the count variable in a For statement.

  Replace the count variable with a variable of the appropriate numeric type.

Illegal BYVAL on arguments to: <*subprogram name*>
You used the ByVal keyword in a call to a procedure that is not an external C function. The ByVal keyword may only be used when specifying the parameters in the declaration or definition of a sub or function with a Declare, Sub, or Function statement, in specifying the parameters of an external C function with a Declare, and in calling an external C function with a Call statement.

Remove the ByVal keyword, revise the definition of the sub or function, or use parentheses around the argument in the call statement to pass the argument by value.

Illegal TO in reference to: <*name*>
One of the following conditions could have caused this error:

- You specified a range (*bound1* To *bound2*) as a subscript in an array element reference.

  Remove the range; specify a single subscript.

- You specified a range (*bound1* To *bound2*) as an argument in a call to a procedure.

  Replace the range by a valid argument.

Use ranges in array declarations or ReDim statements only.

Illegal BYVAL
One of the following conditions could have caused this error:

- You specified the ByVal keyword on a subscript in referring to an array element.

  Remove the ByVal keyword.

- You specified the ByVal keyword in an array bounds expression in a ReDim statement.

  Remove the ByVal keyword.

Duplicate label: *<label name>*

You defined the label specified in the error message more than once within the same scope.

Define the label named in the error message only once. Define other labels to replace the other instances of this label.

Illegal EXIT *<EXIT type>*

You used an Exit statement of a particular type outside a block statement of that type. The six types of Exit statement, and the block statements where each can appear, are as follows:

- Exit Do can appear only within a Do statement
- Exit For can appear only within a For statement
- Exit ForAll can appear only within a ForAll statement
- Exit Function can appear only within a Function statement
- Exit Sub can appear only within a Sub statement
- Exit Property can appear only within a Property Get statement or a Property Set statement

If the Exit statement is unintended, remove it.

If the Exit statement has the right type but is misplaced, relocate it to within the intended block of that type.

If the Exit statement is in the intended place within a block but has the wrong type, change its type to the type of that block.

Illegal OPTION PUBLIC after declaration

The Option Public statement was used after an explicit declaration of a variable, constant, procedure, user-defined data type, or class.

Move the Option Public statement so that it precedes all explicit declarations.

Illegal use of ERASE

You used the Erase statement incorrectly. You can only erase an array, a list, a list element, or a Variant that holds an array, a list, or a list element.

Remove the invalid Erase statement or change the reference in the statement to an array, list, list element, or Variant.

SET may only be used on class instance assignments

You used a Set statement to try to assign something other than a object reference to a variable. For example:

```
Class MyClass
   Public X As Integer
End Class
Dim MyObjRef As New MyClass
Set MyObjRef.X = 5           ' Illegal
Let MyObjRef.X = 5           ' Legal
MyObjRef.X = 5               ' Legal
```

Remove the Set keyword or replace it with the Let keyword.

Illegal pass by value

You tried to pass an argument by value that may not be passed by value, either by using parentheses around the argument, or by using the ByVal keyword on an argument in a call to an external C function.

You may have inadvertently put parentheses around an argument in a sub or function call. Use parentheses on arguments in sub and function calls only if you are using the Call keyword.

The following arguments cannot be passed by value:

- Arrays
- Lists
- Variables of a user-defined data type
- Object reference variables

In addition, only arguments of type String or Variant can be passed by value to the LotusScript Len function. Arguments of other data types cannot be passed by value.

Remove the parentheses or the ByVal keyword.

Wrong number of arguments to constructor for class: *<class name>*

You supplied the wrong number of arguments for a class constructor in one of the following statements:

- A declaration of the form:

  Dim X As New *ClassName*

  For example:

  ```
  Class MyClass
     Sub New(A As Integer, B As String)
        ' ...
     End Sub
  End Class
  ```

```
Dim ObjRef As New MyClass(4, "Alex", "Jones")  ' Illegal
    ' because MyClass's Sub New takes only two arguments
Dim ObjRef As New MyClass(4, "Alex Jones")     ' Legal
```

- A Set statement of the form:

  Set X = New *ClassName*

- A declaration of a derived class when the arguments that the derived class's constructor requires are different from the ones that the base class's constructor requires. In this case, constructor arguments for the base class must be specified after the *BaseClassName* clause in the Sub New declaration, as in the following example:

```
Class BaseClass
   Sub New(X As Integer)
      ' ...
   End Sub
End Class
Class DerivedClass As BaseClass
  Sub New(Y As String, X As Integer), BaseClass(X%, Y)  '
Illegal
  Sub New(Y As String, X As Integer), BaseClass(X)
' Legal
     ' ...
  End Sub
End Class
```

Supply the correct number of arguments to the constructor.

Illegal reference to array or list: *<array or list name>*

You used the name of an array or list in an illegal context. Illegal contexts include the following, where X is the name of an array or list:

- As the target of an assignment or Set statement, as in X = Y, Set X = Y, Set X = New Y, Set X = Bind Y
- As the target of a Delete statement, as in Delete X
- As though it were an object reference variable or a variable of a user-defined data type and you were referring to one of its members, as in X.Y

Remove the illegal use of the array or list.

Illegal type suffix on keyword: *<keyword>*

You included an illegal data type suffix character in the name of a LotusScript built-in function. Certain LotusScript built-in functions can end in the $ type suffix character; no other data type suffix character is valid on these functions. The names of other functions cannot end in a data type suffix character. For example:

```
Print Date()      ' Legal
Print Date$()     ' Legal
Print Date#       ' Illegal
Print CDat(Date)  ' Legal
Print CDat$(Date) ' Illegal
```

Remove the suffix character.

Compiler statement stack overflow at: *<token name>*

The statement being compiled is too complex. It may contain deeply nested block statements, or single-line If statements.

Reduce the nesting level, or break up the offending statement into multiple, less complex statements.

Maximum allowable code size exceeded

The module you are compiling contains more than 64K bytes of executable code.

Split the module into multiple modules and recompile.

Maximum allowable data size exceeded

The module you are compiling contains more than 64K bytes of data.

Split the module into multiple modules and recompile, or reduce the amount of data in the module.

Maximum allowable symbol table size exceeded

The module you are compiling contains more than 64K bytes of symbols (names).

Split the module into multiple modules and recompile, or reduce the number of names in the module.

PUBLIC is not allowed in this module

An Option Public statement, or a declaration of a name as Public, appears in the current module. The product in which you are running LotusScript does not allow Public declarations anywhere within this module.

Move the Option Public statement or the Public declaration to a module where Public declarations are allowed. Alternatively, remove the Option Public statement or remove the keyword Public from the declaration.

Illegal call to: <*sub name*>

You tried to call a class's Sub New or Sub Delete. A class's Sub New is called automatically when an object (class instance) is constructed. It may not be called directly. A class's Sub Delete is called automatically when an object is deleted. It may not be called directly.

Empty parentheses not legal on: <name>

You included empty parentheses in referring to a variable of type Variant or an undefined function or sub (which LotusScript interprets as a reference to an implicitly declared variable of type Variant). For example:

```
Dim anArray(1 To 3) As Integer
Dim varV As Variant
varV() = anArray()        ' Illegal
varV = anArray()          ' Legal
varV = anArray            ' Legal

Dim X As Integer
X% = varV()               ' Illegal
X% = varV                ' Legal
```

Remove the parentheses from the Variant variable.

Illegal use of parentheses

You called a sub or function and enclosed its argument list in parentheses. You can only do this under the following circumstances:

- The sub or function is the target of a Call statement. For example:

```
Call MySub()                          ' Legal
Call MyOtherSub("ABC", 4)             ' Legal
Call MyFunction()                     ' Legal
Call MyOtherFunction(123, "XXX")      ' Legal
```

- The sub or function has a single parameter that the caller is passing by value. For example:

```
MySub("ABC")                      ' Legal
MyFunction(anInt%)                ' Legal
```

- The target is a function that is included in a statement. For example:

```
X% = MyFunction(123, "XXX")       ' Legal
```

The following are illegal:

```
MySub()                           ' Illegal
MyFunction()                      ' Illegal
MyOtherSub("ABC", 4)              ' Illegal
MyOtherFunction(123, "XXX")       ' Illegal
```

Remove the parentheses from around the argument list or call the sub or function with the Call statement.

Class not specified on BIND into: <name>

You tried to assign a reference to a product object to a variable of type Variant with the Set...Bind statement and you omitted the class name of the object. For example, assuming a product class named ProdADT:

```
Dim P As New ProdADT("MyProdADT")
Dim varV As Variant
Set varV = Bind("MyProdADT")              ' Illegal because
                                          ' product class name
                                          ' is missing
Set varV = Bind ProdADT("MyProdADT")      ' Legal syntax
```

Insert the name of the product class after the Bind keyword.

Illegal Directive

Any of the following could have caused this error:

- You used an unrecognized directive. For example:

  ```
  %Else If   ' Illegal
  %ElseIf    ' Legal
  ```

- You nested a %Rem...%End Rem block inside another %Rem...%End Rem block.
- You used an %End Rem without a preceding %Rem.
- You used a %Else, %ElseIf, or %End If directive outside a %If...%End If block.
- You nested a %If...%End If block inside another %If...%End If block.

Unterminated %IF, %ELSEIF, or %ELSE directive

Your script contains a %If directive to which there is no corresponding %End If. For example:

```
%If WIN16
%ElseIf WIN32
' End of script. Error message appears here because there is
' no %End If.
```

Insert a %End If in the appropriate place in the script.

Illegal character after directive

Your script contains a %If directive in which the keyword %Else or the block terminator %End If is followed on the same line by a space or Tab and then one or more characters other than the comment character ('). For example:

```
%If WIN16
%ElseIf WIN32
```

```
%End If  Win16.      ' Illegal
%End If              ' Win16. (This is legal.)
```

Insert the comment character if a comment is intended, or remove the superfluous characters.

LIB name must be a string constant

The name that you specified in the Lib clause of a Declare statement is not a quoted literal or a string constant though that is what is required. Change the name to a quoted literal or string constant.

USE or USELSX name must be a string constant

The name that you specified in a Use or UseLSX statement is not a quoted literal or a string constant though that is what is required. For example, to use the module LSModule :

```
Use LSModule            ' Illegal

Use "LSModule"        ' Legal

Const myModuleName$ = "LSModule"

Use myModuleName$       ' Legal
```

Change the name to a quoted literal or string constant.

EVALUATE argument must be a string constant

The name you specified in an Evaluate function or statement is not a quoted literal or a string constant, though one was required.

Supply a quoted literal or string constant.

Illegal second parenthesized expression

You tried to refer to an element in a nested array, list, or collection. For example:

```
Dim anArray(1 To 3) As Variant
Dim anotherArray(1 To 3) As Integer
anotherArray(1) = 1
anotherArray(2) = 2
anotherArray(3) = 3
anArray(1) = anotherArray
Print anArray(1)(1)               ' Illegal
```

To refer to an element in a nested array, list, or collection, assign the inner array, list, or collection to a variable of type Variant:

```
Dim varV As Variant
varV = anArray(1)
Print varV(1)                     ' Legal
```

Statement is illegal in a subprogram

You used one of the following statements within a LotusScript procedure:

- Class
- Declare
- Function
- One of the Def*type* statements
- Option Base, Option Compare, Option Declare, or Option Public
- Property Get
- Property Set
- Sub
- Type
- Use
- UseLSX

You can only use these statements at the module level.

Move the statement to the module level.

Illegal use of UNICODE or LMBCS keyword

In a Declare statement, you included the Unicode or LMBCS keyword with an object reference argument. This is not allowed. For example:

```
Class MyClass
   ' ...
End Class
Dim X As New MyClass
Declare Function MyFunc Lib "C:\USER.DLL" (X As LMBCS MyClass) _
As Long
' Illegal
```

Instead of passing an object reference, pass a variable of a user-defined data type.

UNICODE and LMBCS strings must be declared BYVAL

In a Declare statement, you included the Unicode or LMBCS keyword with a string argument but did not include the ByVal keyword, which is required for passing string arguments. For example:

```
Declare Function MyFunc Lib "c:\USER.DLLl" (X As LMBCS String) _
As Long
' Illegal
```

```
Include the ByVal keyword in the Declare statement:
```

```
Declare Function MyFunc Lib "c:\USER.DLLl" (ByVal X As LMBCS _
String) As Long
```

Too many nested WITHs

You tried to nest a series of With statements to more than 16 levels. This is not allowed.

Illegal use of escape character in identifier: <name>

You included an escape character in one of the following contexts in which that character is not allowed:

- In a declared name (a variable, constant, procedure, class, or user-defined data type)
- In the name of an implicitly declared variable
- In a label definition or reference
- In the name of the reference variable in a ForAll statement

For example:

```
Dim fo~x As Integer   ' Illegal
```

Remove the escape character.

Illegal use of escape character

You included an escape character at the end of a line. This is not allowed. For example:

```
aString$ = "This is a tilde: "
anotherString$ = aString$~
' This is illegal
```

Remove the escape character.

Error in EVALUATE macro

The macro named in an Evaluate function or statement is not a valid macro in the product that you are using.

Correct the macro or remove the Evaluate function or statement.

Name previously referenced in this scope

You declared a variable in an outer scope. You then referred to this variable in an inner scope and then declared it in that scope. For example:

```
Dim X As Integer
Sub MySub
   X% = 5
   Dim X As Integer   ' Illegal because the preceding
                      ' assignment statement referred to the X
                      ' declared in outer scope
End Sub
```

Move the declaration of the variable in the inner scope so that it precedes the assignment statement, or remove the declaration of the variable in the inner scope. Moving the declaration of the variable in the inner scope creates a local variable that shadows the one in the outer scope, while removing the declaration lets you refer to the variable in the outer scope from within the inner scope.

Wrong number of arguments for event handler: <sub name>
In an On Event statement, the number of arguments you included in the Call clause does not match the number required by the product class event.

Check the product documentation for a description of the arguments defined for the event.

Property is read-only: <property name>
You attempted to apply a Property Set statement to a property of a product object but the product has defined that property as read-only. This means that you can retrieve but cannot modify the property's current value.

Remove the Property Set statement.

Missing array subscript or collection index for: <name>
Either of two conditions could have caused this error:

- You included empty parentheses in a reference to the return value of a function or property. This is not allowed. Assuming that the function or property returns a Variant containing an array, list, or reference to a collection, you can either remove the empty parentheses or insert the appropriate subscript or subscripts. Removing the parentheses makes the reference be to the entire array, list or collection, while including the subscript or subscripts makes the reference be to a single element in the array, list, or collection. For example:

```
Dim anArray(5) As Variant
Function MyFunction(someArray()) As Variant
   ' ...
   MyFunction = someArray
End Function
varV = MyFunction(anArray)()          ' Illegal.
varV = MyFunction(anArray)            ' Legal. Returns
                                      ' the contents
                                      ' of the array.
varV = MyFunction(anArray)(1)         ' Legal. Returns
                                      ' the first element
                                      ' of the array.
```

- You included empty parentheses in a reference to a class member function that returns an array, list, or collection.

  You can either remove the empty parentheses or insert the appropriate subscript or subscripts.

Missing argument to constructor for: <class name>

You used a Dim or Set statement to create a new instance of a user-defined class or product class and omitted one or more of the arguments that the class's constructor sub (Sub New) requires. For example:

```
Class MyClass
   Sub New(aString As String, anInt As Integer)
      ' ...
   End Sub
End Class
Dim X As New MyClass("ABC")     ' Illegal. MyClass's Sub New
                                ' expects two arguments: a
                                ' string and then an integer.
```

or:

```
Class MyClass
   Sub New(aString As String, anInt As Integer)
      ' ...
   End Sub
End Class
Dim X As MyClass
Set X = New MyClass("ABC")     ' Illegal. MyClass's Sub New
                               ' expects two arguments: a
                               ' string and then an integer.
```

Include all of the required arguments in the appropriate order in the Dim or Set statement.

Missing array bound for: <array name>

You used a ReDim statement to define the dimensions of a dynamic array but included an extra comma (,) in the bounds list. For example:

```
Dim anArray()
ReDim anArray(,1,2)    ' Illegal comma at beginning of bounds
                       ' list
ReDim anArray(1,2,)    ' Illegal comma at end of bounds list
ReDim anArray(1,,2)    ' Illegal comma immediately after
                       ' another comma
```

Remove the misplaced comma.

LEN argument must be a variable or string expression

You called the Len function and specified as its argument something other than a string expression or the name of a variable. For example:

```
Print Len(123)        ' Illegal because 123 is a numeric
                      ' constant
Print Len("123")      ' Legal. Returns the number of
                      ' characters in
                      ' the string "123" (3).
Dim X As Integer
Print Len(X%)         ' Legal. Returns the number of bytes
                      ' allocated to store an integer value in
                      ' memory (2).
```

Make the argument a string expression or the name of a variable.

Missing collection index for: <name>

You included empty parentheses in a reference to a collection. This is not allowed. You can either remove the empty parentheses or insert the appropriate subscript. Removing the parentheses makes the reference be to the entire collection, while including the subscript makes the reference be to a single element in the collection.

Missing list subscript for ISELEMENT argument: <list name>

You called the IsElement function and did not include the list tag, which is required. For example:

```
Dim myList List As Double
myList("Alex") = 12345
myList("Martin") = 23456
If IsElement(myList) = TRUE Then Print "Yes."
     ' Illegal
If IsElement(myList("Mary")) = TRUE Then Print "Yes."
     ' Legal
```

Specify a list tag when you call IsElement.

Cannot assign into collection item

You tried to assign a value to a collection item. You can retrieve items in a collection but you cannot assign values to them.

Remove the assignment statement.

Cannot forward declare CLASS or TYPE

You tried to use the Declare statement to declare a user-defined data type or class before defining it. This is not allowed. Declare may only be used to forward-declare functions, subs, and properties.

CLASS or TYPE declaration may not be inside a control block

You tried to include a Class or Type statement inside one or another of the following block statements: Do, For, ForAll, If...Then...Else...EndIf, Select Case, While. This is not allowed. For example:

```
If 1 = 1 Then
   Class MyClass   ' Illegal
      ' ...
   End Class
End If
```

Move the Class or Type statement to outside the block.

Procedure declaration may not be inside a control block

You tried to include a Function, Property Get, Property Set, or Sub statement inside one or another of the following block statements: Do, For, ForAll, If...Then...Else...EndIf, Select, While. This is not allowed. For example:

```
If 1 = 1 Then
   Sub MySub      ' Illegal
      ' ...
   End Sub
End If
```

Move the Function, Property Get, Property Set, or Sub statement to outside the block.

Product class does not have a New method: <class name>

You tried to assign a product object reference to a variable and used the keyword New but the product class does not have a New method. Use a Set...Bind statement instead.

Collection item is not an instance

You referred to an item in an indexed collection as though that item were an object, but it isn't. For example, if iColl is a collection of integers, the following statement would be illegal:

```
iColl(3).value = 4
```

Illegal on declarations in this scope: <*keyword*>
The following conditions could have caused this error:

- You used the keyword Dim, Public, Private, or Static when defining a member variable in a Type statement. For example:

```
Type MyType
   Public X As Integer          ' Illegal: Public keyword
                                ' is not allowed here.
End Type
```

  Remove the Dim, Public, Private, or Static keyword.

- You used the Dim keyword when defining a member variable in a Class statement. For example:

```
Class MyClass
   Dim X As Integer             ' Illegal: Dim keyword is
                                ' not allowed here.
End Class
```

  Remove the Dim keyword.

Wrong return type in event handler <*handler_name*>
The return type of the event does not match the return type of the function.

Event handler must be a FUNCTION
The event handler for an object is a function and the user-defined procedure is a sub.

Event handler must be a SUB
The event handler for an object is a sub and the user-defined procedure is a function.

Conflicting option
You specified conflicting options in an Option Compare statement or statements. You cannot specify any other options if you specify Binary. You cannot specify both Case and NoCase. You cannot specify both Pitch and NoPitch.

PROPERTY GET and SET arguments do not match: <*property_name*>
The corresponding parameters to Get and Set for a property are not of the same type.

Number of arguments do not match for PROPERTY GET and SET *<property_name>*
The number of parameters to Get and Set for a property are not the same.

Property signature does not match parent property: *<property name>*
You used a Declare statement or a Property statement to declare or define a property within the definition of a base class. In subsequently defining a derived class, you declared or defined a property of the same name as the base class's property but with a different signature.

One of the following does not match:

- The data type
- The number of parameters
- The data type of one of the parameters
- The data structure of one of the parameters

Change the signature of the base class property or of the corresponding property in the derived class so that they match.

Type suffix character required on: *<name>*
A variable that was implicitly declared with a data type suffix character was used without the suffix character. When a variable is implicitly declared with a suffix character, all subsequent references must contain the suffix character. A reference without the suffix character is treated as an implicit declaration of an already declared variable. This is illegal (a variable can't be declared twice).

Append the suffix character to the variable name when you refer to it.

Must be a sub: *<procedure_name>*
A module-level procedure named "Initialize" or "Terminate" must be a sub. Initialize and Terminate are special subs at the module level. Initialize is executed when the module is loaded and Terminate when it is unloaded.

Appendix G
Run-time Error Messages

This chapter describes the run-time error messages in the LotusScript language.

User-defined error

A user-defined error occurred. You used the Error statement to create an error and assigned it a number that is not a LotusScript error number. You did not specify an error message in the Error statement, so LotusScript displays the default error message "User-defined error."

To display a more meaningful error message, provide the message string as the second argument to the Error statement.

RETURN without GOSUB

You executed a Return statement without having first transferred control in the procedure to a labeled statement using a GoSub statement or an On...GoSub statement.

Use a GoSub or On GoSub statement to transfer control to a labeled statement before executing a Return statement.

Illegal function call

The following conditions could have caused this error:

- You tried to pass a negative subscript to an array.
- You tried to pass an invalid argument to the ACos function, the ASin function, or the ATn2 function.
- You tried to pass an invalid argument to the Asc function. The empty string ("") is an invalid argument.
- You used an invalid string expression with the Val function.
- You tried to pass an invalid argument to the Chr function.
- You tried to use the Date function to set an invalid system date.
- You tried to use the Time function to set an invalid system time.
- You tried to pass an invalid argument to the DateNumber function or the TimeNumber function.
- You tried to pass NULL, EMPTY, the empty string (""), or a number less than 1 or greater than 255 to the Environ function.

- You tried to use a negative record number in the Get statement or the Put statement.
- You tried to pass an invalid start position or an invalid count to the Instr function or the InstrB function.
- You tried to pass an array to the Len function or the LenB function.
- You tried to pass a negative value to the Log function or the Sqr function.
- You tried to pass a negative value or a value greater than 65K as an argument to a function.
- You tried to pass an invalid window style argument (it must be an integer from 1 to 9 inclusive) to the Shell function.
- You tried to pass an invalid comparison argument (it must be 0 or 1) to the StrCompare function.
- You tried to activate a program using the ActivateApp statement, but the program was not found.
- The string specified in the SendKeys statement contained an unmatched parenthesis, an illegal key name, or an illegal repeat count; or the string was too long to be processed.

Overflow

The result of a numeric operation, value conversion, or assignment is outside the range of allowable values for the result data type.

Do one or both of the following:

- Change the numeric data type of one or more values being used in the operation, conversion, or assignment.
- Change the destination data type to accommodate the result. For example:

```
Dim N As Long

I% = 30000            ' Declare I implicitly as an Integer.
J% = 10000            ' J is also an Integer.
Print (I% + J%)       ' Overflow from numeric operation. The
                      ' number 40000 cannot be represented as
                      ' an Integer.
Print CInt(40000&)    ' Overflow from attempted conversion of
                      ' a Long value to an Integer value.
Invar% = 40000        ' Overflow from attempted assignment of
                      ' a large value to an Integer variable.
N = 40000             ' No error. N was declared a Long.
MN = 40000            ' No error. MN is implicitly declared a
```

```
                    ' Long by the assignment of a large
                    ' value to it.
```

Invalid ^ operator operands

In an expression whose operator is the exponentiation operator (^), the pair of operands is invalid.

The expression X **^** Y (the base X raised to the exponent Y) cannot be evaluated when

- X is 0, and Y is negative or 0: for example, 0 **^** -2
- X is negative, and Y is not an integer: for example, -1 **^** 2.2

Respecify the expression, or the computations leading up to it, to ensure that the operands will have legal values when the **^** operator is applied to them.

Out of memory

There is not enough system memory to perform an operation.

To free memory on your computer, end one or more other programs that are currently in memory, other than the Lotus software running LotusScript.

Subscript out of range

When accessing an array, either the number of subscripts does not match the given array's defined dimensions, or the size of one or more subscripts does not match the given array's bounds.

- If the subscripts are correct, redefine the array dimensions or bounds using the ReDim statement.
- If the array dimensions and bounds are correct, the subscripts must be respecified. For example:

```
Dim TiArr(-1 To 1)
Print TiArr(0, 0)        ' Error. Array is defined as having
                         ' one dimension, and is being
                         ' accessed as a two-dimensional
                         ' array.
Print TiArr(-2)          ' Error. The specified subscript
                         ' falls outside of the array's
                         ' defined bounds.
```

Expression out of range

You used a numeric expression whose value at run time is out of the legal range, in one of these contexts:

- As the numeric expression in an On...GoTo or On...GoSub statement.

  The value of the expression must be between 0 and 255 inclusive.
- As the designated error number in an Err or Error statement.

  The error number must be positive or 0.
- As the designated error number in an Error function call.

  The error number must be positive.

Respecify the expression in the statement or in the function call, to ensure that its value falls within the legal range.

Duplicate PUBLIC name in USE module: *<module name>*

You declared as Public a name that is also declared as Public in another loaded module, a module that was loaded in executing a Use statement.

Determine the duplicate Public name and change its declaration in one module or the other.

Division by zero

In a mathematical operation, there was an attempt to divide by zero. It is impossible to divide by zero.

Check the appropriate operand for a zero value before using it as a divisor.

Type mismatch

One of the following conditions could have caused this error:

- You attempted an operation on operands with conflicting data types.
- You assigned a value to a variable that has a different data type, and LotusScript cannot convert it automatically.
- You are passing a value as an argument that has a different declared data type, and LotusScript cannot convert it automatically.
- You used a string as the initial value, or as the To or Step value, in a For statement.

Use the correct data type.

Out of string space

There is too little available memory for string storage, either at compile time or at run time.

If your program includes many strings, or very long strings, either eliminate some strings, or restructure your program to limit the set of strings that must be kept in memory at any one time.

If your program includes a great many names, you may need to restructure it similarly. LotusScript creates and stores a string for each name. The string's length is the number of characters in the name. For example, if your program includes a definition of a type with several thousand members, string storage space may be exhausted.

No RESUME

You are using an On Error statement in a procedure, but have not included a Resume statement.

Insert one or more Resume statements at the appropriate points in the script.

RESUME without error

You tried to execute a Resume statement outside of an error-handling routine. You cannot resume execution if an error has not occurred.

Insert the Resume statement within an error-handling routine.

Out of stack space

One of the following conditions could have caused this error:

- You wrote a recursive function that never reaches its base case, and therefore never terminates itself.

  Rewrite the function so that it reaches its base case.

- You declared too many local variables in a procedure.

  Remove a sufficient number of variable declarations in the procedure to free up stack space by rewriting it as several smaller procedures. If you are using fixed arrays, declare them as dynamic.

Sub or function not defined

You declared a sub or function with a Declare statement, and then tried to call it before defining it with a Sub or Function statement.

Define the function or sub.

Error in loading DLL

The dynamically linked library (DLL) specified in a Declare statement could not be found.

If the Declare statement does not specify the path of the DLL, LotusScript seeks the DLL as follows, in order:

- In the working directory
- In the directories on the search path specified by the DOS environment variable PATH

If the Declare statement specified the path of the DLL, correct the DLL name or the path.

If the Declare statement did not specify the path of the DLL, do one of the following:

- Specify the path of the DLL in the Declare statement.
- Move the DLL to the working directory; or change the working directory to the directory that contains the DLL.
- Add the location of the DLL to the DOS environment variable PATH.

Bad DLL calling convention

You are using a C-callout function to call a DLL entry point with a different calling convention than the one used to implement the DLL entry point.

Define the correct argument-passing protocol in the C-callout function to implement the DLL entry point.

Internal error

An internal error occurred.

Record the error message and contact Lotus Software Support.

Bad file name or number

You tried to access a file that does not exist, or you specified a file number that is currently not assigned to a file. For example, using Print # to print to a file that has not first been opened generates this error.

- Check the spelling of the file name and correct it if it is wrong.
- Open the file first with the specified file number.
- Check the file number and correct it if it is wrong.

File not found

You referred to a file that cannot be found.

If you are using DOS, you can correct this problem in the following way:

If you do not specify the path of the file, LotusScript seeks the file as follows, in order:

- In the working directory
- In the directories on the search path specified by the DOS environment variable PATH

If the file specification included the path, correct the file name or the path.

If the file specification did not include the path, then do one of the following:

- Specify the path.
- Move the file to the working directory; or change the working directory to the directory that contains the file.
- Add the location of the file to the DOS environment variable PATH.

Bad file mode

You used an Open statement to try open a file in a mode that is incompatible with the file's access type. For example, opening a file for Output that has Read access causes this error.

If you intended to open this file, change either the file's access type, or change the For clause specification in the Open statement.

File already open

You used the Open statement on a file that is already open.

Use Close to close the open file, or remove the Open statement that attempts to open it.

Device I/O error

The following conditions could have caused this error:

- You tried to write to a read-only disk.

  Change the disk's read-only access to read-write access.

- You tried to open a file on a protected diskette.

  Make sure there is a diskette in the drive and that it has read-write access.

File already exists

You tried to create a file with the same name as a file that already exists on disk.

Specify a different file name.

Bad record length

You tried to give a record length for a file that is incompatible with that specified in the Open statement for the file, or you specified a negative record length.

For record-oriented I/O with random files, use the Len = *reclen* clause of the Open statement to define the record length. When opening the file for reading, *reclen* should be the same as when the file was opened for writing.

Disk full

You tried to save a file on a disk that did not have enough room for the file.

Save the file on another disk.

Input past end of file

One of the following conditions could have caused this error:

- You tried to read past the end of the file.

  Use the EOF function to check for the end-of-file.

- An Input statement tried to read in more values than are present in the last record in the file.

  Adjust the number of values being read so that the number read from the last record is less than or equal to the number of values in the last record.

Bad record number

You tried to read from a file using a record number that is either invalid (negative) or out-of-bounds (larger than the number of records in the file).

If you are using a Get statement, make sure that the record numbers are within the bounds of the file. Numbering of records begins at 1.

Bad file name

You specified a file using an invalid DOS file name.

Specify a correct file name using DOS file naming rules.

Too many files

You have too many files open in LotusScript.

Close some open files before attempting this file operation.

Device unavailable

You specified an invalid drive.

Specify a drive that exists on your system.

Permission denied

One of the following conditions could have caused this error:

- You tried to access a file that is currently locked by another program.

  Close the file in the other program.

- You tried to write to a file that has been write-protected. In DOS, this attribute can be changed with the Attrib command. In Windows, this attribute can be changed with the File Properties command.

  To correct this problem in Windows, open the File Manager (or Windows Explorer), choose File Properties to remove the read-only attribute from the file, then return to LotusScript and try again to write to the file.

Disk not ready

The disk drive door is not closed.

Close the disk drive door.

Cannot rename with different drive

You used a Name statement to try to rename a file to a different drive than the one where it is currently stored. You cannot change the drive on which a file is stored when you rename the file.

Rename the file without changing the drive where the file is currently stored.

Path/file access error

One of the following conditions could have caused this error:

- You tried to access a file that is currently locked by another program.

  Close the file in the other program.

- You tried to access a file or directory that is protected.

  If you are using DOS, you can correct this problem by using the Attrib command to change the attributes of the file or directory.

  If you are using Windows, you can correct this problem by opening the File Manager or Windows Explorer and choosing File Properties.

Path not found

You specified a path that cannot be found.

Check the path to make sure you specified it correctly and that it exists, and then respecify it.

Object variable not set

You tried to access an instance of a LotusScript class or product class, but either of the following was true:

- The object reference variable you used does not hold a reference to any object. (Its value is NOTHING.)

  Use the Set statement to assign the variable a reference to an object.

- The object reference variable has been deleted.

  Remove the statement that refers to the deleted variable.

FOR loop not initialized

One of the following conditions could have caused this error:

- You used the GoTo statement to transfer control to a For statement.

  You cannot use the GoTo statement to transfer control to a For statement (though you *can* use it to exit a For loop).

- The count variable that you specified in a For statement does not have a valid initial value.

  Make sure the count variable of the For statement is properly initialized. For example, in the statement For I = X, I and X must be of the same data type.

Invalid pattern string

You used an invalid pattern string with the Like operator.

Record the error message number and contact Lotus Software Support.

Invalid use of NULL

You tried to convert a NULL value to another value type. NULL cannot be converted to another value type.

For example, the function call CInt (NULL) is an invalid use of NULL. This function call attempts to convert NULL to an integer explicitly.

Implicit conversion of NULL is also invalid, as in the following sequence of statements:

```
S = NULL
For I = 1 To 5 Step S
Next
```

In a For statement, the step value must be numeric. LotusScript attempts to convert the value in S to a number when executing the For statement above. This is an invalid use of NULL.

Use the IsNull function to determine if a value is NULL.

Cannot destroy active instance

You attempted to delete an instance of a class that is still in use in your program.

Record the error message number and contact Lotus Software Support.

File not writable

You tried to write to a file that is marked read-only on disk.

- Save the file under a different file name.
- In Windows, open the File Manager (or Windows Explorer), choose File Properties to remove the read-only attribute from the file, and then return to LotusScript to save the file.
- In DOS, use the DOS Attrib command to remove the read-only attribute from the file.

File not readable

You used an Open statement to try to open a file that cannot be read at this time. It may currently be locked by another program, or it could be corrupted and therefore cannot be opened.

If the file is currently locked by another program, access the other program and close the file there.

Illegal file number

You specified a file number outside of the range 1 to 255 in an Open statement.

Specify a file number between 1 and 255.

File not open

In a statement or function that requires an open file, you specified a file that is not open.

Use the Open statement to open the file.

Conflicting modes supplied

You used an Open statement to try to open a file in a mode that is incompatible with its access type. For example, opening a file for Output that has Read access causes this error.

If you intended to open this file, change either the file's access type, or change the For clause specification in the Open statement.

Unable to open file

The following conditions could have caused this error:

- You tried to open a file that was not found.

  Verify that you specified the correct file name.

- You tried to open a file that is currently locked by another active program on your system.

  Close the file in the other program.

- You tried to open a file that has been corrupted.

  Recreate the file.

Illegal operation for file mode

You tried to perform an operation on a file that is illegal for the file's mode. For example, using the Get statement on a sequential file generates this message.

Use the FileAttr function to determine the file's mode before performing this operation on the file.

Data too big for record

You tried to write data into a record that is too small for the amount of data you are writing.

Write less data into the record, or create another file with a larger record size to hold the data.

Bad attribute

You supplied an illegal file attribute number using the FileAttr function.

Supply a legal attribute number (either 1 or 2) that specifies the type of information you want.

Cannot set attribute for file

You tried to supply a legal file attribute using the FileAttr function, but could not do so because the file is write-protected or is being used by another program.

Verify whether or not you can access the file, or close the file in the other program.

List item does not exist

You used a list tag that does not exist in a list.

Before accessing a list element with that tag, use the IsElement function to test if the element exists in the list.

Cannot find module *<module name>*

You tried to access a Public name in a module that is not loaded. At compile time, that module was accessed indirectly: it was made available by a Use statement in another loaded module, not the current module.

Insert a Use statement in the current module to make the other module available before accessing the Public name.

Cannot find external name *<name>*

The currently executing module contains a Use statement whose target module contains a Public name to which the currently executing module refers. That name has been changed in the target module since the currently executing module was compiled.

Restore the original name in the target module, or change the name in the currently executing module to the new name.

Type mismatch on external name *<name>*

The currently executing module contains a Use statement whose target module contains a Public name to which the currently executing module refers. The data type of the name in the target module has been changed since the currently executing module was compiled.

Restore the name's original data type in the target module, or change the name's data type in the currently executing module to the new data type.

Module already loaded

In the Use statement, you named a module that is already present.

If the named module is the one you want to load, remove the Use statement. Otherwise, change the name in the Use statement.

Invalid module file
You tried to use a module that is incompatible with this release of LotusScript.

Verify that the script source language is compatible with this release of LotusScript, and recompile the module.

Compiler error
The function signature of an external C-callout function has been corrupted.

Record the error message number and contact Lotus Software Support.

Opcode <*opcode name*> not implemented
A required operation code has not been implemented.

Record the error message number and contact Lotus Software Support.

Named product object does not exist
You tried to use a product object that does not exist.

Refer to an existing product object; or, in the LotusScript product from which you invoked LotusScript, define the object you are trying to use.

ADT error: Control procedure missing
The Lotus software from which you invoked LotusScript is missing a procedure needed to manage product objects.

Record the error message number and contact Lotus Software Support.

Bad argument to external function
In a Declare statement, you declared an external function using an invalid argument.

Replace the invalid argument with a valid one.

Unsupported argument type to external function
In a Declare statement, you declared an external function using an unsupported type for an argument.

Replace the argument type with a valid one.

Unsupported return type for external function
In a Declare statement, you declared an external function using an unsupported type for the return value.

Replace the return type with a valid one.

External function not found

The dynamically linked library (DLL) named in a Declare statement for an external function was found, but the declared function was not found in the DLL.

If the function name was misspelled in the Declare statement, correct it.

If the function name was correct but the wrong DLL was specified, specify the correct DLL.

Event handler not attached

You tried to remove an event handler from a product object with an On Event statement, but the handler is not bound to the object.

Remove the On Event statement or specify the correct handler.

Module in use

You tried to unload the currently running module.

Remove any attempt to unload the currently running module.

Illegal circular USE: *<module name>*

The module currently being compiled contains a Use statement whose target module contains a Use statement whose target module refers to the current module (possibly by transitivity) in another Use statement. Use statements cannot be circular: if module A uses module B, then B, or any module that B uses, may not use A.

Reconfigure the set of Use statements to remove the circularity.

Too many calls into module

You have exceeded the allowable maximum number of nested calls to functions or subs within a single module.

Record the error message number and contact Lotus Software Support.

LISTTAG argument not a list element

Within a ForAll loop that iterates over the elements of an array, you supplied the reference variable as the argument to the ListTag function. You can apply ListTag only to the ForAll reference variable for a list, not an array. For example:

```
Dim anArray(10)
ForAll X In anArray
   Print ListTag(X)             ' Illegal. ListTag can only
                                ' refer to a list, not to an
                                ' array.
End ForAll
```

Remove the incorrectly used ListTag function.

Illegal REDIM of fixed array

You used the ReDim statement to resize an existing fixed array. You can only use ReDim to declare a dynamic array, or to resize an existing dynamic array.

Remove the ReDim statement.

Array size exceeds maximum limit

The total storage space in memory of the dynamic array exceeds the allowable maximum of 64K. For example:

```
ReDim MyArr(-20000 To 20000) As Integer
' This declares an array with 40,001 elements of 2 bytes each.
' The declared array size is greater than 64K.
```

Use the ReDim statement to decrease the array size.

Illegal LIKE pattern

The pattern specified for a Like operation is illegal for one of the following reasons:

- You specified a range of characters in square brackets, but the second character is earlier in the collating sequence than the first character. For example:

  ```
  "a" Like "[e-a]"
  ```

  Reverse the order of the characters in the illegal range specification.

- You specified an open square bracket without a close square bracket. For example:

  ```
  "a" Like "[abc"
  ```

  Supply the close square bracket. If you want to specify an open square bracket as a character to match, enclose it in square brackets:

  ```
  "[" Like "[[]"
  ```

Error in constant expression evaluation

An error occurred in evaluating a constant expression. The error is explained in one of the following messages:

Division by zero

Illegal function call

Illegal Like pattern

Invalid ^ operator operands

Invalid use of NULL

Out of string space

Overflow

Operation not supported on this platform

You tried to use a LotusScript function, statement, or directive that your operating system does not support. For example, the CreateObject statement is not supported under OS/2 or UNIX.

Remove the unsupported function call, statement, or directive.

Type suffix does not match actual data type

You referred to a variable, constant, function, or property with a data type suffix character that does not match its declared data type. For example:

```
Class MyClass
  Public X As Integer
End Class
Dim varV As Variant
Set varV = New MyClass
Print varV.X$           ' Illegal because X was declared as
                        ' an Integer.
```

Change the suffix character to match the declared data type, or remove the suffix character.

Instance member does not exist

You referred to a nonexistent member of a class. For example:

```
Class MyClass
'...
End Class
Dim varV As Variant
Set varV = New MyClass
Print varV.Something    ' Illegal because Something is not
                        ' defined as a member of MyClass.
```

Define the member within the class, or remove the reference.

Variant does not contain an object

You referred to a variable of type Variant as though it contained an object reference, but no such reference has been assigned to it. For example:

```
Dim varV As Variant
varV.Something        ' Illegal.
```

Remove the reference or insert a statement before it that assigns an object reference to the Variant.

Variant does not contain a container

You referred to a variable of type Variant as though it held an array, list, or collection but it does not hold one of these. For example:

```
Dim varV As Variant
varV(1) = 5           ' Illegal.
```

Remove the reference or insert a statement before it that assigns a list, fixed array, or reference to a collection to the Variant.

Wrong number of arguments for method

You called a function or sub that is a member of a user-defined class and passed it either too few or too many arguments. For example:

```
Class MyClass
   Function MyFunction(A As Integer) As Integer
   '...
   End Function
End Class
Dim varV As Variant
Set varV = New MyClass
Dim X As Integer
X% = varV.MyFunction           ' Illegal: too few arguments.
X% = varV.MyFunction(5,10)     ' Illegal: too many arguments.
```

Supply the correct number and type of arguments.

Name used as a method is not a method

You referred to something as though it were a member function or sub of a class when no such function or sub has been defined for that class. For example:

```
Class MyClass
'...
End Class
Dim varV As Variant
Set varV = New MyClass
Print varV.Something("ABC")     ' Illegal: Something is not
                                ' defined as a sub or function
                                ' in MyClass.
```

Remove the reference or define the sub or function as a member of the class.

Illegal use of sub

You defined a sub as a member of a class and then referred to that sub as though it were a member function, property, or variable. For example:

```
Class MyClass
   Sub MySub
   '...
   End Sub
```

```
End Class
Dim varV As Variant
Set varV = New MyClass
X = varV.MySub          ' Illegal: a sub doesn't have a
                        ' return value.
varV.MySub = 5          ' Illegal: you can't assign a sub a
                        ' value.
```

Remove the reference or redefine the sub as the appropriate type of class member.

Illegal use of function

You defined a function as a member of a class and specified its return type as something other than Variant or object reference. You then referred to that function as though its return type were an object reference or a Variant holding an array, list, or object reference. For example:

```
Class MyClass
  Function MyFunction(X As Integer) As Integer
  '...
  End Function
End Class
Dim varV As Variant
Set varV = New MyClass
Print varV.MyFunction.F(1)            ' Illegal.
Print varV.MyFunction.Something       ' Illegal.
```

Remove the reference or change the function's return type to Variant.

Illegal use of property

You defined a property as a member of a class and then referred to that property in an inappropriate way. For example:

```
Class MyClass
  Property Set MyProp As Integer
  '...
  End Property
  Property Get MyProp As Integer
  '...
  End Property
End Class
Dim varV As Variant
Set varV = New MyClass
varV.MyProp           ' Illegal: a reference to a property
                      ' must occur in a statement that assigns
                      ' or retrieves the property's value.
X% = varV.MyProp(1)   ' Illegal: integer variables can't be
                      ' subscripted.
```

Remove the reference or correct its syntax.

Illegal use of read-only property

You tried to assign a value to a property of a product object, but the product has defined that property to be read-only. This means that you can retrieve but cannot assign that property's value.

Remove the assignment statement.

List reference must contain exactly one subscript

You declared a list variable as a class member. When you subsequently referred to that list, you either omitted a subscript or included more than one subscript. A reference to a list must include one, and only one, subscript. For example:

```
Class MyClass
  Public myList List As Integer
End Class
Dim varV As Variant
Set varV = New MyClass
Print varV.myList(1,1)   ' Illegal: too many subscripts.
Print varV.myList()      ' Illegal: missing subscript.
```

Supply one, and only one, subscript.

Illegal DELETE

You tried to use the Delete statement to delete a member of an object rather than the object itself. The Delete statement requires a plain object name. For example:

```
Class MyClass
  Public X As Integer
End Class
Dim varV As Variant
Set varV = New MyClass
Delete varV.X            ' Illegal.
Delete varV              ' Legal.
```

Remove the Delete statement or change its argument to an unqualified object name.

Not a product object

Where a reference to a product object was expected in an On Event statement or an Evaluate function or statement, you used a reference to a user-defined object. This is not allowed. For example:

```
Class MyClass
'...
End Class
Sub MySub
'...
End Sub
Dim varV As Variant
Set varV = New MyClass
On Event Click From varV Call MySub   ' Illegal.
On Event Click From varV Remove MySub ' Illegal.
X = Evaluate("mymacro",varV)          ' Illegal.
```

Remove the function or statement, or change it so that it refers to a product object.

Event does not exist

The event that you specified in an On Event statement is not defined for the specified product object. For example, suppose that ProdADT is a product object for which NotAnEvent is not a defined event:

```
Sub MySub(Source As ProdADT)
'...
End Sub
Set prodObjRef = New ProdADT("astring")
On Event NotAnEvent From prodObjRef Call MySub     ' Illegal.
```

Remove the On Event statement or specify an event defined for the object.

Event handler argument count mismatch

The event you specified in an On Event statement requires a different number of parameters than are found in the specified event handler's signature. For example, assume that the Moved event defined for the product class Walden requires three parameters (an object reference and two integers):

```
Sub GoodSub(Source As Walden, X As Integer, Y As Integer)
'...
End Sub
Sub BadArgNum(Source As Walden, X As Integer)
'...
End Sub
Dim objRefVar As New Walden("ABC")
On Event Moved From objRefVar Call GoodSub         ' Legal.
On Event Moved From objRefVar Call BadArgNum       ' Illegal:
BadArgNum
```

```
' has only 2 parameters,
' but Moved requires 3.
```

Change the event handler's signature to make it have the required number of parameters.

Event handler argument type mismatch

The event you specified in an On Event statement requires that one or more of the event handler's parameters be different in data type from what appears in the event handler's signature. For example, assume that the Moved event defined for the product class Walden requires three parameters (an object reference and two integers):

```
Sub GoodSub(Source As Walden, X As Integer, Y As Integer)
'...
End Sub
Sub BadArgType(Source As Walden, X As Integer, Y As String)
'...
End Sub
Dim objRefVar As New Walden("ABC")
On Event Moved From objRefVar Call GoodSub        ' Legal.
On Event Moved From objRefVar Call BadArgType     ' Illegal:
      ' BadArgType's third parameter should be an Integer.
```

Change the event handler's signature so that its parameters are of the required data types.

Not a PUBLIC member

You referred to a variable, property, function, or sub that was defined as a Private member of a class. Private members are not visible outside of the class to which they belong. For example:

```
Class MyClass
  X As Integer                 ' X is Private by default.
  Private Function Z As Integer
  '...
  End Function
End Class
Dim varV As Variant
Set varV = New MyClass
varV.X% = 10                   ' Illegal: X is Private.
anInt% = varV.Z%               ' Illegal: Z is Private.
```

Remove the reference or, if possible, change the definition of the class member from Private to Public.

Missing argument

You called a member sub or function of a product class and omitted one or more of the arguments that it expected. For example, assume a product class Walden that has a member sub Move that has two integer parameters:

```
Dim varV As Variant
Set varV = New Walden("ABC")
varV.Move 5                    ' Illegal: Walden's Move method
                               ' has two parameters, not one.
```

Supply the required number of arguments in the call, or remove the calling statement.

Operation is disallowed in this session

The product from which you are running LotusScript has disabled the function, statement, or directive that you attempted to use.

Remove the function call, statement, or directive.

Attempt to access an uninitialized dynamic array

Either of the following situations could have produced this error:

- You tried to assign an uninitialized dynamic array to a Variant:

  ```
  Dim anArray() As Integer
  Dim varV As Variant
  varV = anArray               ' Illegal.
  ```

  Use the ReDim statement to assign bounds to the array before assigning the array to the Variant.

- You tried to pass an uninitialized dynamic array to the LBound or UBound function:

  ```
  Dim anArray() As Integer
  LB% = LBound(anArray)        ' Illegal.
  ```

  Use the ReDim statement to assign bounds to the array before calling the LBound or UBound function.

Error loading USE or USELSX module

The target that you specified in a Use or UseLSX statement cannot be found or is invalid (possibly because of a version discrepancy).

Supply the name of an existing file of the appropriate format or remove the Use or UseLSX statement.

Wrong number of collection indices

You used more than a single subscript in referring to a member of a collection. For example, assuming a collection class IntegerCollection:

```
Dim IntCol As New IntegerCollection("astring",10)
'...
Dim varV As Variant
Set varV = IntCol
Print varV(1,1)       ' Illegal.
Print varV(1)         ' Legal.
```

Use one, and only one, subscript when referring to a collection member.

Not a collection object

You referred to a product object as though it were a collection, but it isn't a collection. For example, assuming the product class ProdADT, which is not a collection class:

```
Dim varV As Variant
Set varV = New ProdADT("abc")
ForAll X In varV              ' Illegal.
'...
End ForAll
```

Remove the reference or replace its target with the name of a collection.

Collection item not found

You tried to refer to a nonexistent member of a collection. For example, assuming a collection class IntegerCollection:

```
Dim varV As Variant
Dim IntCol As New IntegerCollection("astring",10)
Print IntCol(3)       ' Illegal because the collection doesn't
                      ' have any members.
```

Add members to the collection before trying to refer to them; specify an index that identifies a member; or remove the reference.

Underflow

An internal error occurred.

Record the error message and contact Lotus Software Support.

SET required on class instance assignment

You attempted to assign an object reference to a variable but omitted the Set keyword. (An object reference can be a reference to a user-defined object, a product object, an OLE automation object, or the constant NOTHING). The Set keyword is required in object reference assignments. For example:

```
Class MyClass
'...
End Class
Dim varV As Variant
Dim otherVarv As Variant
Dim X As New MyClass
Set varV = X
otherVarV = varV        ' Illegal.
otherVarV = New varV    ' Illegal.
Set otherVarV = varV    ' Legal.
Set otherVarV = New varV ' Legal.
```

Include the Set keyword in the assignment statement or remove the statement.

Invalid Collection item

You attempted to access a member of a collection, but the product was unable to comply with your request correctly.

Record the error message and contact Lotus Software Support.

Automation-Object error

An error occurred when you tried to refer to an OLE Automation object.

Check the syntax of the statement that caused the error, and check the documentation for the OLE Automation object to which you tried to refer.

Automation-Object cannot create

You called CreateObject or GetObject but LotusScript could not interpret the argument or arguments in the call.

Make sure that the arguments designate a valid application and class and, if appropriate, a valid path.

Automation-Object file name error

The path that you specified in a call to GetObject is invalid.

Specify a valid path or remove the GetObject statement.

Automation-Object member not found

You referred to an undefined member of an OLE Automation object, or you attempted to assign a value to an OLE Automation object property that is read-only.

Check the documentation for the OLE Automation object to ascertain its members and their status.

Automation-Object argument count

You called a method of an OLE Automation object and included too few or two many arguments. The number of arguments must be the same as the number of parameters defined for the method.

Check the documentation for the OLE Automation object to ascertain the method's parameters.

Automation-Object argument type mismatch

You called a method of an OLE Automation object and included one or more arguments whose data type differs from the corresponding parameters in the method's definition. The data type of each argument must be the same as the data type of the corresponding parameter.

Check the documentation for the OLE Automation object to ascertain the data type of each of the method's parameters.

ForAll container invalid or modified

You tried to assign a value to the target in a ForAll block. For example:

```
Dim anArray(3) As Integer
Dim varV As Variant
varV = anArray
ForAll X In varV
  '...
  varV = 4      ' Illegal.
End ForAll
```

Remove the assignment statement.

Out of system stack space

You entered an expression that LotusScript is unable to evaluate because the expression contains too many elements. For example, an expression consisting of hundreds of values separated by arithmetic operators would cause this error because the result of each individual arithmetic operation has to be saved on the stack until they can all be combined to calculate the value of the expression as a whole, and there isn't enough room on the stack to save them all.

Break the expression up into smaller pieces handled by multiple statements.

Illegal REDIM

You used a ReDim statement in a context in which it is inappropriate:

- In referring, with the Preserve keyword, to a variable of type variant that doesn't already contain an array. For example:

```
Dim varV As Variant
varV = 5
ReDim Preserve varV(1 To 3)      ' Illegal
```

  Remove the keyword Preserve if you want varV to hold an array, or remove the Redim statement.

- You referred to a member variable of a class as though it were an array, though it isn't. For example:

```
Class AClass
   Public X As Integer
End Class
Dim varV As Variant
Set varV = New AClass
ReDim varV.X(1 To 3)             ' Illegal, because X
                                 ' isn't an array.
```

  Declare X as a dynamic array or remove the ReDim statement.

- You referred to a member variable or property of a class as though it held or returned a dynamic array rather than a fixed array. For example:

```
Class AClass
   Public X(1 To 2) As Integer
End Class
Dim varV As Variant
Set varV = New AClass
ReDim varV.X(1 To 3)             ' Illegal, because X is a
                                 ' fixed array.
```

  Define X as a dynamic array or remove the ReDim statement.

Error creating product object

You tried to create an instance of a product class but the product encountered an error condition (such as Out of Memory) and was unable to create the object.

Record the error message and contact Lotus Software Support.

Error accessing product object property

You tried to refer to a property of an instance of a product class but the product encountered an error condition when you tried to do so.

Record the error message and contact Lotus Software Support.

Error accessing product object method

You tried to refer to a method (member sub or function) of an instance of a product class but the product encountered an error condition when you tried to do so.

Record the error message and contact Lotus Software Support.

Error accessing product object

You tried to delete an instance of a product class but the product encountered an error condition when you tried to do so.

Record the error message and contact Lotus Software Support.

Error in EVALUATE macro

When you tried to execute an Evaluate function or statement, the product containing the macro to which the function or statement refers encountered an error condition.

Record the error message and contact Lotus Software Support.

Event handler return type mismatch

The return type of the event does not match the return type of the function when attaching an event function to an object through a variant.

Event handler procedure type mismatch

The event handler for an object is a sub and the user-defined procedure is a function, or vice-versa, when attaching an event handler to an object through a variant.

Wrong number of arguments for PROPERTY

The number of parameters do not match when accessing an object property through a variant.

Illegal use of MEMBER

An argument list is specified when accessing an object member variable through a variant.

PROPERTY SET not defined
A set operation is attempted through a variant on an object property that does not define Property Set.

PROPERTY GET not defined
A get operation is attempted through a variant on an object property that does not define Property Get.

String too large
A string is generated at run-time that exceeds the size limit of 32,000 characters.

Variable is read-only
A set operation is attempted on a product variable that is read-only.

Unknown class instance
An product object is returned for a class not registered with LotusScript.

Cannot assign into collection item
An attempt is made to write to a member when accessing a collection object through a variant.

Wrong number of array subscripts
An array access through a variant has the wrong number of subscripts.

Index

Symbols

- (minus sign)
  LotusScript, 4-5, 4-8, 4-13
(pound sign)
  LotusScript, 4-31
%If directive
  LotusScript, 9-2, 12-146
%Include directive
  LotusScript, 9-2, 12-153
%Rem directive
  LotusScript, 12-249
& (ampersand)
  LotusScript, 4-31
& (string concatenation) operator
  LotusScript, 4-31
* (asterisk)
  LotusScript, 4-5, 4-8
. (dot notation)
  LotusScript, 12-34, 12-314
. (period)
  LotusScript, 8-13, 11-4
/ (slash)
  LotusScript, 4-5, 4-9
[ ] (brackets)
  LotusScript, 11-4
^ (caret)
  LotusScript, 4-5, 4-7
~ (tilde) escape character
  LotusScript, 2-4
+ (addition) operator
  LotusScript, 4-5, 4-11, 4-31
< (less than sign), 4-33
  LotusScript, 4-5
<> (not equal operator), 4-33
  LotusScript, 4-5
= (assignment) operator
  LotusScript, 12-194
= (equal sign), 4-33
  LotusScript, 4-5, 4-31
> (greater than sign), 4-33
  LotusScript, 4-5, 4-31
>= or => (greater than or equal to operator), 4-33
  LotusScript, 4-5, 4-31

A

Abs function
  LotusScript, 12-1
Absolute values
  LotusScript, 12-1
Access keyword
  LotusScript, 12-199
Access modes
  changing, 6-13
Access types for files
  LotusScript, 12-101
Acos function
  LotusScript, 12-2
ActivateApp function
  LotusScript, 11-9
ActivateApp statement
  LotusScript, 12-3
Actual parameters
  LotusScript, 5-13
Addition operator (+)
  LotusScript, 4-5, 4-11
ADT errors, G-14
Agents, types of
  HTTP, 10-1
Agents, working with
  synchronization, 10-3
Alias keyword
  LotusScript, 12-62
Aliases
  list of, D-1
Ampersand (&)
  LotusScript, 4-31
And operator
  LotusScript, 4-5, 4-24
ANSI characters
  LotusScript, 12-15, 12-33, 12-298
Any keyword
  LotusScript, 12-62
AppActivate alias
  LotusScript, D-1
AppActivate statement
  LotusScript, 12-3
Append keyword
  LotusScript, 12-225
Applications
  determining use, 11-1
  interacting with programs, 11-9
  interacting with SendKeys statement, 12-270
  LotusScript, 12-3, 12-279
Arccosine
  LotusScript, 12-2
Arcsine
  LotusScript, 12-16
Arctangent
  LotusScript, 12-16, 12-17
Argument passing
  data type converting, 3-2
Arguments
  LotusScript, 12-43, 12-277, 12-279
  passing, 5-4, 5-12, 11-15
Arithmetic operators. *See* Operators
ArrayAppend function
  LotusScript, 12-4
ArrayGetIndex function
  LotusScript, 12-9
ArrayReplace function
  LotusScript, 12-10
Arrays
  arguments to C functions, 11-20
  bounds list, 3-30
  data type, 3-2, 3-34
  DataType function, 3-41
  deleting, 12-87
  Dim statement, 3-31
  dimension, 3-29
  dynamic, 3-30
  elements, 3-36
  Erase statement, 3-40
  fixed, 3-32
  index, 3-29
  IsArray function, 3-41
  limiting, A-3
  LotusScript, 3-29, 12-73, 12-125, 12-171, 12-229, 12-246, 12-319
  lower bounds, 3-30
  NotesOutlineEntry class, 12-184
  passing, 11-20
  ReDim statement, 3-39
  sizing, 3-39

subscript, 3-29
TypeName function, 3-41
upper bounds, 3-30
ArrayUnique function
LotusScript, 12-13
As keyword
external C call, 12-62
forward reference, 12-66
LotusScript, 12-34, 12-73, 12-238, 12-246, 12-299
Name statement, 12-215
Asc function
LotusScript, 12-15
ASin function
LotusScript, 12-16
Assignment operator (=)
LotusScript, 12-194
Assignment operators. *See* Operators
Assignment to variables
LotusScript, 12-194, 12-273
Asynchronous agents
enabling, 10-7
ATn function
LotusScript, 12-16
ATn2 function
LotusScript, 12-17
Atomic update, 10-1
Attributes
files, 12-79
Automatic data type conversions
LotusScript, 3-5

B

Backslash
LotusScript, 4-5
Bars
vertical (|), 2-3
Base classes, 8-7
methods, 8-8
properties, 8-8
referring to a member, 8-26
Base keyword
Option Base statement, 12-229
Base of numbers
LotusScript, 12-19, 12-140, 12-216
BAT files
LotusScript, 12-277, 12-279
Beep statement
LotusScript, 12-18
Bin function
LotusScript, 12-19
Binary files
accessing, 6-9
LotusScript, 6-1, 6-9, 12-129, 12-158, 12-225, 12-242
opening, 6-7
reading, 6-7, 6-12
variable length record, 6-7
writing, 6-10
Binary keyword
LotusScript, 12-225, 12-230
Binary numbers
LotusScript, 2-2, 12-19
Binary operations
LotusScript, 3-2
Bind keyword
LotusScript, 12-273
Bitwise operators. *See* Operators
Blank spaces
LotusScript, 12-204, 12-263, 12-282, 12-283, 12-307, 12-313
statement construction rules, 2-1
Block statements
LotusScript, 9-4, 9-6, 9-8
Boolean data type
LotusScript, 12-20
Boolean operators. *See* Operators
Boolean values
LotusScript, 3-50
Bounds for arrays
limiting, A-3
LotusScript, 12-184, 12-229, 12-319
Bounds lists
LotusScript, 3-30
Bracket notations
LotusScript, 11-4, 12-21
Branching statements
LotusScript, 9-10 to 9-13, 12-138, 12-139, 12-222, 12-255
Built-in constants
EMPTY, 3-14
FALSE, 3-14
LotusScript, 3-13, 11-7
NOTHING, 3-14
NULL, 3-14
PI, 3-14
TRUE, 3-14
Built-in functions
ActivateApp, 11-9
CDat, 3-51
Command, 11-1
CreateObject, 11-12
DataType, 3-41
Date, 3-52
DateNumber, 3-52
DateValue, 3-52
Day, 3-52
described, 5-1
Environ, 11-9
EOF, 6-11
Erl, 7-2
Err, 7-2
Error, 7-3
Error$, 7-3
FileDateTime, 3-52
Format, 3-52
FreeFile, 6-10
GetObject, 11-12
Hour, 3-52
Input, 6-10
InputBox, 11-6
IsArray, 3-41
IsDate, 3-52
LBound, 3-36
LOF, 6-10
LotusScript, 3-26
MessageBox, 11-6
Minute, 3-52
Month, 3-52
Now, 3-52
Second, 3-52
Seek, 6-12
Shell, 11-9
Time, 3-52
TimeNumber, 3-52
Timer, 3-52
TimeValue, 3-52
Today, 3-52
TypeName, 3-41
WeekDay, 3-52
Year, 3-52
Yield, 11-9
Byte-oriented functions
LotusScript, 12-160, 12-163, 12-166, 12-167, 12-187, 12-190, 12-191, 12-210, 12-257
ByVal keyword, 5-4
Declare statement, 11-20
forward reference, 12-66
LotusScript, 12-62, 12-299

C

C functions, 11-13
calling, 11-13
calling convention, 11-13
declaring, 11-14
external, 12-62
passing arguments to, 11-15
return value, 11-25, 12-62

C language, 11-13
Call keyword
LotusScript, 12-23, 12-220
Call statement
LotusScript, 5-19, 12-23
Calling
C functions, 11-13
Sub Delete, 8-25
Sub New, 8-25
Case keyword
LotusScript, 12-185, 12-230, 12-268
Case sensitivity
LotusScript, 12-164, 12-166, 12-167, 12-230, 12-288
CBool function
LotusScript, 12-25
CCur function
LotusScript, 12-27
CDat function
LotusScript, 3-52
CDbl function
LotusScript, 12-30
Character codes
LotusScript, 12-15, 12-33, 12-298, 12-321, 12-325
Character extraction
LotusScript, 12-186 to 12-188, 12-208, 12-210, 12-211, 12-256 to 12-258
Character oriented functions, 12-188
InStrC function, 12-168
LotusScript, 12-193, 12-211, 12-258, 12-291 to 12-294
Characters
case, 12-230, 12-320
LotusScript, 12-185, 12-230
special, 2-10
ChDir statement
LotusScript, 12-31
ChDrive statement
LotusScript, 12-32
Chr function
LotusScript, 12-33
CInt function
LotusScript, 12-33
Class constructor
LotusScript, 12-304
Class destructor
LotusScript, 12-302
Class members
public and private, 8-13
referring to, 8-13
scope, 8-13, 8-14
Class statement
LotusScript, 12-34
Classes
array and list, 8-28
base classes, 8-7
benefits, 8-8
bracket notation, 11-4
class library, 8-8
Collection classes, 11-5
creating an object, 11-2
creating object, 8-30
declaring, 8-29
defining a variable, 8-9
deleting an object, 8-17, 11-5
derived classes, 8-7
dot (.) notation, 8-13, 11-4
dotdot (..) notation, 8-26
events, 11-4
inheritance, 8-7
instance, 8-7
LotusScript, 12-83
Me keyword, 8-13
methods, 8-7, 11-4
object member, 8-13
object reference, 3-2, 8-13, 8-29
overriding a method, 8-21
overriding a property, 8-21
properties, 11-4
scope, 8-13, 8-14
user-defined, 3-2
ClassName property
JavaClass class, 11-36
ClearJavaError method
JavaSession class, 11-74
CLng function
LotusScript, 12-38
Close statement
LotusScript, 12-39
CodeLock function
LotusScript, 12-39
CodeLockCheck function
LotusScript, 12-42
CodeUnlock function
LotusScript, 12-43
Collection classes
LotusScript, 11-5
Collections
for an array, 3-29
of lists, 3-42
Columns
printing output, 12-307
COM files
LotusScript, 12-277, 12-279
Command function
LotusScript, 11-1, 12-43
Command line arguments
LotusScript, 12-43, 12-277, 12-279
Comments
LotusScript, 9-2, 12-249
Compare keyword
LotusScript, 12-230
Comparing
LotusScript, 12-288
Comparison
LotusScript, 12-230, 12-288
Comparison operator. *See* Operators
Compile times
errors, 1-13, 7-1
Compiled files
LotusScript, 8-8
Compiled modules
loading, 1-12
Compiled scripts, 1-13
Compiler directives
LotusScript, 9-2, 12-146, 12-153, 12-249
placing, 2-1
Compiler limits, A-4
Concatenation operator (&)
LotusScript, 4-31
Concatenation operators. *See* Operators
Conditional statements
LotusScript, 12-142, 12-144 to 12-146
Const statement
LotusScript, 12-44
Constants
built-in, 3-13
data type suffix characters, 3-17
defined, 3-13
EMPTY, 3-14
FALSE, 3-14
in LSCONST.LSS, 3-15
in LSPRVAL.LSS, 3-15
LotusScript, 12-44
LSCONST.LSS file, 3-13
naming rules, 2-4
NOTHING, 3-14
NULL, 3-14
PI, 3-14
platform identification, 12-146
product-specific, 3-15
scope, 3-9
testing data type, 3-17
TRUE, 3-14
user-defined, 3-15

Constructor (New sub)
LotusScript, 12-34
Constructor sub. *See* Sub New
Container classes. *See* Collection classes
Container variable
LotusScript, 9-27
Context switch, 10-1
Conversions
changing case, 12-185, 12-320
Converting data types
LotusScript, 12-25, 12-27, 12-30, 12-33, 12-38, 12-50, 12-53
overview, 3-2
Converting numbers
LotusScript, 12-19, 12-140, 12-216
Converting strings
LotusScript, 12-289, 12-326
Copying files
LotusScript, 12-102
Cos function
LotusScript, 12-46
Cosine
LotusScript, 12-46
Count property
JavaMethodCollection class, 11-51
JavaPropertyCollection class, 11-68
CreateLock function
LotusScript, 12-47
CreateObject function
LotusScript, 11-12, 12-48
CreateObject method
JavaClass class, 11-36
CSng function
LotusScript, 12-50
CStr function
LotusScript, 12-50
CurDir function
LotusScript, 12-51
CurDrive function
LotusScript, 12-52
Curly braces ({ })
LotusScript, 2-3
Currency
LotusScript, 12-52
Currency conversions
LotusScript, 12-27
Currency data types, 3-1
default value, 3-19
LotusScript, 12-52
Current errors
LotusScript, 7-2
Current property
JavaMethodCollection class, 11-53
JavaPropertyCollection class, 11-69
CVar function
LotusScript, 12-53
CVDate alias
LotusScript, D-1

D

Data
limiting, A-1, A-2
Data type suffix characters
constants, 3-17
LotusScript, 3-16, 12-69, 12-73
omitting, 3-17
Data types
array, 3-2, 3-34
constants, 3-17
converting, 3-2, 12-19, 12-27
currency, 3-1, 12-52
date/time, 3-51
default, 3-17
determining, 3-17
Double data type, 3-1
integer, 3-1
list, 3-2
long, 3-1
LotusScript, 3-1, 12-20, 12-54, 12-56, 12-69, 12-83, 12-84, 12-170, 12-202, 12-281, 12-297, 12-317, 12-327
numeric limits, A-1
object reference, 3-2
scalar, 3-1
single, 3-1
string, 3-1
user-defined, 3-2, 12-314
user-defined in C language function calls, 11-24
variables, 3-19
variant, 8-32
variants, 3-2
DataType function
LotusScript, 3-17, 3-41, 3-53, 12-54
Date and time handling
converting, 12-59, 12-60, 12-310, 12-312, 12-329
measuring, 12-311
reading, 12-57, 12-61, 12-103, 12-141, 12-212, 12-213, 12-216, 12-264, 12-309, 12-312, 12-335
setting, 12-58, 12-310
testing, 12-171
Date function
LotusScript, 3-52, 12-57
Date statement
LotusScript, 3-52, 12-58
Date values
valid range, 3-51
DateNumber function
LotusScript, 3-52, 12-59
DateSerial alias
LotusScript, D-1
DateSerial function
LotusScript, 12-59
DateTime
data type, 3-51
DateValue function
LotusScript, 3-52, 12-60
Day function
LotusScript, 3-52, 12-61
Debugger. *See* Script Debugger
Debugging
a script, 1-15
Decimals
LotusScript, 2-2
Declarations
LotusScript, 9-2
scope, 3-10
Declare keyword
external C call, 12-62
forward reference, 12-66
LotusScript, 12-233
Declare statement
LotusScript, 5-4, 11-14
Declaring
a dynamic array, 3-38
a list, 3-42
a property, 5-23
a sub, 5-19
fixed array, 3-34
object reference, 8-29
user-defined, 8-3
Declaring variables
explicitly, 3-19
implicitly, 3-24
LotusScript, 12-73, 12-233
Default data type
LotusScript, 3-17
Default values
variables, 3-21
DefByte statement
LotusScript, 12-69
DefCur statement
LotusScript, 12-69

Defining
a function, 5-2
a property, 5-23
a sub, 5-19
an error, 7-1, 7-7
member variables, 8-3, 8-9
Defining functions
LotusScript, 12-125
Definition statements
LotusScript, 9-2
DefInt statement
LotusScript, 12-69
DefLng statement
LotusScript, 12-69
DefSng statement
LotusScript, 12-69
Deftype statement
LotusScript, 3-26
DefVar statement
LotusScript, 12-69
Delete
LotusScript, 12-87
Delete statement
LotusScript, 8-17, 12-71
Delete sub, 5-22
calling, 8-25
Deleting, files
LotusScript, 12-183
Delimiters
LotusScript, 2-10
Derived classes, 8-7, 8-20
defining a member, 8-21
using Sub New, 8-23
DestroyLock, 10-8
DestroyLock function
LotusScript, 12-73
Destructor
LotusScript, 12-34, 12-71, 12-302
Destructor sub. *See* Sub Delete
Dialog boxes
LotusScript, 12-161, 12-204
Differences
Macintosh platform, B-5
OS/2 platform, B-1
Dim statement, 8-29
dynamic array, 3-38
fixed array, 3-34
for a list, 3-42
LotusScript, 3-23, 3-31, 12-73
Dimensions
for an array, 3-29
Dir function
LotusScript, 12-79
Directives
LotusScript, 9-2, 12-146, 12-153
Directories
LotusScript, 12-51, 12-79
Directories and files
LotusScript, 12-31, 12-32, 12-51, 12-52, 12-79, 12-102, 12-183, 12-213, 12-215, 12-225, 12-259
Disjunction (Or) operator
LotusScript, 4-25
Disk drives
LotusScript, 12-32, 12-52, 12-79
Division operator (/)
floating-point division (/), 4-5
integer division, 4-5
LotusScript, 4-9
Divisions
remainder, 4-5, 4-10
DLL files
Declare statement, 11-13
LotusScript, 12-62, 12-323
using, 11-13
Do keyword
Do statement, 12-81
Do loops
LotusScript, 9-15
Do statement
LotusScript, 9-15
DoEvents function and statement
LotusScript, 12-337
DominoAsynchronousAgents, 10-7
Dot (.) notation
LotusScript, 8-13, 11-4, 11-75, 12-34, 12-83, 12-314
Dotdot (..) notation
LotusScript, 8-26
Double data types
LotusScript, 3-1, 12-84
Double precision numbers
LotusScript, 3-1
Drives
LotusScript, 12-32, 12-52, 12-79
Dynamic arrays
DataType function, 3-41
declaring, 3-38
Dim statement, 3-38
LotusScript, 3-30, 12-73, 12-246
ReDim statement, 3-39
TypeName function, 3-41
Dynamic Link Libraries
LotusScript, 11-13

E

Early termination statements
LotusScript, 12-84, 12-98
Editor. *See* Script Editor
Elapsed time
LotusScript, 12-311
Elements
array, 3-36
array data type, 3-41
Else keyword
LotusScript, 12-142, 12-144, 12-145, 12-268
ElseIf keyword
LotusScript, 12-145
Empty strings
LotusScript, 2-4
Empty values
LotusScript, 3-14, 12-73, 12-176, 12-327
Encapsulation
LotusScript, 8-13
End of File
LotusScript, 12-86
End statement
LotusScript, 9-30, 12-84
Environ function
LotusScript, 11-9, 12-85
Environment variables
LotusScript, 11-9
EOF function
LotusScript, 6-11, 12-86
Equals operator
LotusScript, 4-5, 4-13
Equals operator (=)
LotusScript, 4-31
Eqv operator
LotusScript, 4-5, 4-28
Erase statement
LotusScript, 3-40, 12-87
Erasing
an object, 11-5
Erl function
LotusScript, 7-2, 12-88
Err function
LotusScript, 7-2, 12-89
Err statement
LotusScript, 12-91
Error function
LotusScript, 7-3, 12-92
Error handling
LotusScript, 7-1, 7-2, 7-6, 9-3, 12-88, 12-89, 12-91 to 12-93, 12-217, 12-254

Error keyword
LotusScript, 12-217
Error line numbers
returning, 7-2
Error messages, 9-3, 12-92, F-57, G-1, G-6, G-23, G-25 to G-29
compile-time, F-52
defining, 7-6
file, 7-7
LotusScript, F-1 to F-58
returning, 7-3
run-time, G-28, G-29
Error number
LotusScript, 12-89, 12-91
Error numbers
defining, 7-6
LotusScript, 7-1
resetting, 7-2
returning, 7-2
Error statement
LotusScript, 12-93
Error statements
LotusScript, 7-2, 7-7
Error$ function
LotusScript, 7-3
ErrorMsg property
JavaError class, 11-43
Evaluate function and statement
LotusScript, 12-95
Event handling
LotusScript, 12-220
Event keyword
LotusScript, 12-220
Events
defined, 1-12
for Lotus product classes, 11-4
Examples
LotusScript, 12-2, 12-25, 12-55, 12-85, 12-131, 12-143, 12-184, 12-207, 12-235, 12-265, 12-308, 12-326
Exclusive Or (Xor) operator
LotusScript, 4-27
Exclusive Or operator
LotusScript, 4-5
EXE files
LotusScript, 12-277, 12-279
Execute function and statement
LotusScript, 12-96
Executing
a script, 1-15
a sub, 5-19
a user-defined function, 5-9
Exit statement
flow, 9-32
LotusScript, 12-98
Exp function
LotusScript, 12-100
Explicit data type conversions
LotusScript, 3-4
Explicitly declaring variables
LotusScript, 3-19, 12-69, 12-73
Exponentiation operator (^)
LotusScript, 4-5, 4-7
Exporting
library function, 11-13, 12-62
Expression
compiling as a temporary module, 12-96
Expressions
described, 4-1
Expressions and operators
LotusScript, 4-1
Extended character sets
LotusScript, 12-149, 12-151
External declarations, 11-13
library function, 12-62
External functions
library function, 11-13
LotusScript, 12-62

F

False values
LotusScript, 3-14, 3-50
File access
LotusScript, 12-199
File attributes
LotusScript, 12-132, 12-275
File information
LotusScript, 12-101, 12-103, 12-275
File number, unused
LotusScript, 12-124
File operations
summary, 6-1
FileAttr function
LotusScript, 12-101
FileCopy statement
LotusScript, 12-102
FileDateTime function
LotusScript, 3-52, 12-103
FileLen function
LotusScript, 12-103
Files
binary, 6-1, 6-7, 6-9, 12-129, 12-242
closing, 6-13, 12-39, 12-253
compiling a script, 1-13
fixed length, 6-12
formatting data, 12-331
getting information, 12-201
limits on operations, A-3
locking, 12-199
LotusScript, 12-39, 12-86, 12-129, 12-155, 12-158, 12-160, 12-163, 12-196, 12-198, 12-236, 12-242, 12-253, 12-265, 12-266, 12-283, 12-307, 12-331, 12-333
LotusScript constants, 11-7
LSCONST.LSS, 3-13
LSCONST.LSS file, 11-7
LSO, 8-8, 1-13
naming, 1-14
opening, 6-4, 6-10, 12-124
random, 6-1, 6-4, 6-8, 12-129, 12-242
reading, 6-3, 6-6, 6-7
reading from, 6-10, 12-129, 12-155, 12-160, 12-163, 12-196
sequential, 6-1, 6-2, 6-8, 12-155, 12-196, 12-236, 12-333
using with data type, 8-5
variable length, 6-11
writing, 6-3, 6-5, 6-7
writing to, 6-10, 12-199, 12-236, 12-242, 12-333
Files, positioning
LotusScript, 12-86, 12-198, 12-265, 12-266
Fix function
LotusScript, 12-104
Fixed arrays
bounds list, 3-34
DataType function, 3-41
declaring, 3-34
Dim statement, 3-34
dimension, 3-34
LotusScript, 12-73
lower bounds, 3-34
reinitializing, 3-40
size, 3-34
TypeName function, 3-41
upper bounds, 3-34
Fixed length records
LotusScript, 6-12
Fixed length strings
LotusScript, 3-21
Floating-point numbers
LotusScript, 3-1, 12-84, 12-281

Flow control statements
LotusScript, 9-1
Flow of execution
LotusScript, 9-1
For keyword
LotusScript, 12-98, 12-105, 12-225
For loops
LotusScript, 9-19
For statement
functioning, 12-105
LotusScript, 12-105
managing flow, 9-19
nesting, 9-21
ForAll keyword
LotusScript, 12-98
ForAll statement
container variable, 9-27, 12-107
functioning, 12-107
LotusScript, 12-107
managing flow, 9-25
Formal parameters
LotusScript, 5-13
Format function
LotusScript, 3-52, 12-110
Forward references
LotusScript, 12-66
Fraction function
LotusScript, 12-123
FreeFile function
LotusScript, 6-10, 12-124
From keyword
LotusScript, 12-220
FullTrim function
LotusScript, 12-125
Function statement
LotusScript, 12-125
Functions
block terminator, 5-1
calling C function, 11-13
declaring C function, 11-14
defining, 5-2
described, 5-1
executing, 5-9
forward reference, 12-66
in a class, 8-10
in run-time errors, 7-2
LotusScript, 11-1, 12-23, 12-62
maximum argument, A-4
naming rules, 2-4
overriding, 8-21
passing arguments to C functions, 11-15
predefined, 3-26
recursive, 5-11
return value, 3-26, 5-7
signature, 5-1
terminating, 9-30
user-defined, 5-9
with multiple arguments, 5-11
with no arguments, 5-9
with one argument, 5-10

G

Get keyword
LotusScript, 12-129, 12-225, 12-238
Get statement
LotusScript, 6-12, 12-129
GetAttr alias
LotusScript, D-1
GetAttr function
LotusScript, 12-132
getClass method
JavaSession class, 11-75
getClassMethods method
JavaClass class, 11-38
getClassProperties method
JavaClass class, 11-39
GetFileAttr function
LotusScript, 12-132
getFirst method
JavaMethodCollection class, 11-55
JavaPropertyCollection class, 11-70
getLastJavaError method
JavaSession class, 11-76
GetMethod method
JavaClass class, 11-40
getNext method
JavaMethodCollection class, 11-56
JavaPropertyCollection class, 11-71
getNth method
JavaMethodCollection class, 11-57
JavaPropertyCollection class, 11-72
GetObject function
LotusScript, 11-12, 12-134
GetProperty method
JavaClass class, 11-41
GetThreadInfo function
LotusScript, 12-136
getValue method
JavaProperty class, 11-64
GoSub keyword
LotusScript, 12-138, 12-222
GoSub statement
LotusScript, 9-13
GoTo keyword
LotusScript, 12-139, 12-142, 12-217, 12-223
GoTo statement
LotusScript, 9-10, 12-139
Greater than operator (>)
LotusScript, 4-5, 4-31
Greater than or equal to operator
LotusScript, 4-5, 4-13, 4-31

H

Handling
an error, 7-2, 7-7
Hex function
LotusScript, 12-140
Hexadecimal numbers
LotusScript, 12-140
numeric construction, 2-2
Hidden files
LotusScript, 12-79, 12-132, 12-275
Hiding
data, 8-13
Hiragana input mode
LotusScript, 12-149, 12-151
Host operating system differences, B-1, B-2, B-5
Hour function
LotusScript, 3-52, 12-141
HTTP agents
multi-threading, 10-1
serial, 10-2
threaded, 10-2

I

Identifiers
construction rules, 2-4
for variables, 3-19
maximum length, A-4
reserved for LotusScript, 2-6
If (%If directive)
LotusScript, 12-146
If...GoTo statement
LotusScript, 12-142
If...GoTo...Else statement
LotusScript, 9-10
If...Then...Else statement
LotusScript, 9-4, 12-144

If...Then...Elseif statement
LotusScript, 9-6, 12-145
IMESetMode function
LotusScript, 12-149
IMEStatus function
LotusScript, 12-151
Imp operator
LotusScript, 4-5, 4-29
Implicitly declaring variables
LotusScript, 3-24
Implode function
LotusScript, 12-152
Include (%Include directive)
LotusScript, 12-153
Inclusive Or (Or) operator
LotusScript, 4-25
Indexes
for a list, 3-42
for an array, 3-29
Inheritance
LotusScript, 8-7, 8-8, 8-20
Initialize sub
LotusScript, 5-21, 12-303
Initializing
values, 12-314
Input # statement
LotusScript, 6-11, 12-155
Input function
LotusScript, 6-10, 12-158
Input keyword
Input function, 12-158
LotusScript, 12-196, 12-225
Input mode
LotusScript, 12-149, 12-151
InputB function
LotusScript, 12-160
InputBox function
LotusScript, 11-6, 12-161
InputBP function
LotusScript, 12-163
Instances
of a class, 12-34
Instances of a class. *See* Objects
InStr function
LotusScript, 12-164
InStrB function
LotusScript, 12-166
InStrBP function
LotusScript, 12-167
InStrC function
LotusScript, 12-168
Int function
LotusScript, 12-169
Integer data type
LotusScript, 3-1, 12-170
Integer division operator (\)
LotusScript, 4-5, 4-10
Interacting
programs with applications, 11-9
with the user, 11-6
International functions
LotusScript, 12-149, 12-151, 12-188, 12-193, 12-211, 12-258, 12-289, 12-291, 12-292, 12-293, 12-294
Intrinsic functions. *See* Built-in functions
Invoke method
JavaMethod class, 11-50
Is keyword
LotusScript, 12-268
Is operator
LotusScript, 4-38, 8-16
IsA operator
LotusScript, 4-39
IsArray function
LotusScript, 3-41, 12-171
IsDate function
LotusScript, 3-52, 12-171
IsDefined function
LotusScript, 12-172
IsElement function
LotusScript, 12-174
IsEmpty function
LotusScript, 12-176
IsList function
LotusScript, 12-177
IsNull function
LotusScript, 12-177
IsNumeric function
LotusScript, 12-178
IsObject function
LotusScript, 12-179
IsScalar function
LotusScript, 12-180
IsUnknown function
LotusScript, 12-181
Iteration. *See* Loops
Iterative statements
LotusScript, 9-15, 9-19, 9-25, 9-30

J

JavaClass class
ClassName property, 11-36
CreateObject method, 11-36
getClassMethods method, 11-38
getClassProperties method, 11-39
GetMethod method, 11-40
GetProperty method, 11-41
LS2J classes, 11-35
JavaConnect
Arguments processing, 11-77
Data type mappings, 11-77
Java reference types, 11-77
Java Virtual Machine (JVM), 11-73
limitations, 11-77
JavaError class
ErrorMsg property, 11-43
LS2J classes, 11-42
StackTrace property, 11-44
JavaMethod class
Invoke method, 11-50
JClass property, 11-47
LS2J classes, 11-46
MethodName property, 11-47
Modifier property, 11-49, 11-62
Signature property, 11-50
JavaMethodCollection class
Count property, 11-51
Current property, 11-53
getFirst method, 11-55
getNext method, 11-56
getNth method, 11-57
LS2J classes, 11-51
JavaObject class
LS2J classes, 11-58
JavaProperty class
getValue method, 11-64
JClass property, 11-61
LS2J classes, 11-60
Modifier property, 11-62
PropertyName property, 11-61
setValue method, 11-66
Type property, 11-63
JavaPropertyCollection class
Count property, 11-68
Current property, 11-69
getFirst method, 11-70
getNext method, 11-71
getNth method, 11-72
LS2J classes, 11-67
JavaSession class
ClearJavaError method, 11-74
getClass method, 11-75
getLastJavaError method, 11-76
LS2J classes, 11-73
JClass property
JavaMethod class, 11-47
JavaProperty class, 11-61

Join function
LotusScript, 12-182
Jumps (branching)
LotusScript, 12-138, 12-139, 12-222, 12-255

K

Katakana input mode
LotusScript, 12-149, 12-151
Keystrokes
sending, 12-270
Keywords
LotusScript, 2-6
Me, 8-13
new, 8-29
Preserve, 3-39
Kill statement
LotusScript, 12-183

L

Labels
for statements, 9-3
placing, 2-1
rules for construction, 2-5
Language limits
function and sub argument, A-4
source language statement, A-4
LBound function
LotusScript, 3-36, 12-184
LCase function
LotusScript, 12-185
Left function
LotusScript, 12-186
Left-align strings
LotusScript, 12-203
LeftB function
LotusScript, 12-186
LeftBP function
LotusScript, 12-187
LeftC function
LotusScript, 12-188
Len function
LotusScript, 12-188
Len keyword
Len function, 12-188
LotusScript, 12-225
LenB function
LotusScript, 12-190
LenBP function
LotusScript, 12-191
LenC function
LotusScript, 12-193
Length of files
LotusScript, 12-103
Length of strings
LotusScript, 12-188, 12-190, 12-191, 12-193
Lengths
files, 12-201
Less than operator
LotusScript, 4-13
Less than or equal to operator
LotusScript, 4-13
Let statement
LotusScript, 12-194
Lib keyword
LotusScript, 12-62
Libraries
script, 1-13
Like operator
LotusScript, 4-34
Limits
array size, A-3
array variable, A-3
compiled program structure, A-4
file operations, A-3
numeric data, A-1
string data, A-2
Line continuation character (_)
LotusScript, 2-1
Line Input # statement
LotusScript, 6-12, 12-196
Line numbers
LotusScript, 12-88
Line widths
LotusScript, 12-331
List data type
LotusScript, 3-2, 3-42
List element
deleting, 12-87
List keyword
forward reference, 12-66
LotusScript, 12-73, 12-299
List tags
case sensitivity, 3-43
Lists
declaring, 3-42
deleting, 12-87
list tag, 3-42
LotusScript, 12-73, 12-174, 12-177, 12-197
ListTag function
LotusScript, 12-197
Literal strings
construction of, 2-3
LMBCS keyword
Declare statement, 11-20
LMBCS strings
LotusScript, 12-62
LOC function
LotusScript, 12-198
Local variables
LotusScript, 5-13
Lock keyword
Lock statement, 12-199
LotusScript, 12-225
LockID, 10-8
Locking functions
LotusScript, 12-39, 12-42, 12-43, 12-47, 12-73
Locks
creating and destroying, 10-8
thread safe code, 10-7
LOF function
LotusScript, 6-10, 12-201
Log function
LotusScript, 12-201
Logical operators. *See* Operators
Long data type
LotusScript, 3-1, 12-202
Loop control variables
For statement, 9-19
ForAll statement, 9-27
Loop keyword
LotusScript, 9-15, 12-81
Loops
Do loop, 9-15
For loop, 9-19
ForAll loop, 9-25
LotusScript, 12-81, 12-330
terminating, 9-32
While loop, 9-30
Lotus product classes
bracket notation, 11-4
Collection class, 11-5
creating an object, 11-2
deleting an object, 11-5
dot (.) notation, 11-4
events, 11-4
methods, 11-4
properties, 11-4
Lotus products
determining use, 11-1
interacting, 11-9
LotusScript
described, 1-9
error messages file, 7-7
keywords, 2-6
REXX integration, C-1

LotusScript constants. *See* Built-in constants
LotusScript data types
LotusScript, 12-20, 12-52, 12-84, 12-170, 12-202, 12-281, 12-297, 12-327
LotusScript statements
Call, 5-19
Date, 3-52
Declare, 5-4, 11-14
Deftype, 3-26
Delete, 8-17
Dim, 3-23, 8-29
Do, 9-15
End, 9-30
Erase, 3-40
Err, 7-6
Error, 7-6
Exit, 9-32
For, 9-19
ForAll, 9-25
Get/Set, 12-238
getting, 6-12
GoSub, 9-13
GoTo, 9-10
If...GoTo...Else, 9-10
If...Then...Else, 9-4
If...Then...Elseif, 9-6
Input #, 6-11
Line Input #, 6-12
On...GoSub, 9-13
On...GoTo, 9-12
Open, 6-10
Option Base, 3-35
Print, 11-6
Print #, 6-12
Property Get, 5-22
Property Set, 5-22
Put, 6-10
ReDim, 3-39
Return, 9-13
Seek, 6-12
Select Case, 9-8
SendKeys, 11-9
Set, 8-29
Time, 3-52
using, 1-13, 8-8
While, 9-30
With, 8-15
Write #, 6-11
Yield, 11-9
Lower bounds
fixed array, 3-34
LotusScript, 3-30
LS2J
ADT, 11-30
classes, 11-34
data type mappings, 11-77
dot notation, 11-30
error handling, 11-32
example, 11-81
installation, 11-28
Java errors, 11-32
Java precision, 11-77
Java security, 11-27
Java Virtual Machine (JVM), 11-28
limitations, 11-34
method invoking, 11-30
Notes agent, 11-28
Script Libraries, 11-29
string mapping, 11-77
system requirements, 11-28
Use statement, 11-29
using, 11-28
LS2J classes
JavaClass class, 11-35
JavaError class, 11-42
JavaMethod class, 11-46
JavaMethodCollection class, 11-51
JavaObject class, 11-58
JavaProperty class, 11-60
JavaPropertyCollection class, 11-67
JavaSession class, 11-73
LSCONST.LSS file
LotusScript, 11-7
viewing, 3-15
LSERR.LSS file
LotusScript, 7-7
LSet statement
LotusScript, 12-203
LSO files, 1-13
LotusScript, 8-8
LSPRVAL.LSS file, 3-15
LSS files
LotusScript, 12-153
LSX files
LotusScript, 11-27, 12-323
LTrim function
LotusScript, 12-204

M

Macintosh limitations
LotusScript, 11-75
Macintosh platform differences, B-5
Margins
script, 2-1
Matching
strings, 4-34
Mathematical functions
LotusScript, 12-1, 12-2, 12-16, 12-17, 12-46, 12-100, 12-104, 12-123, 12-169, 12-201, 12-245, 12-259, 12-261, 12-277, 12-280, 12-286, 12-309
Me keyword
LotusScript, 8-13, 12-34
Member variables
defining, 8-3
referring to, 8-4
Members
of a class, 12-34
of a type, 12-314
Members of classes
scope, 8-14
MessageBox function
LotusScript, 11-6
MessageBox function and statement
LotusScript, 12-204
MethodName property
JavaMethod class, 11-47
Methods
defining, 8-9
for Lotus product classes, 11-4
overriding, 8-21
referring to, 11-4
Mid function
LotusScript, 12-208
Mid statement
LotusScript, 12-209
MidB function
LotusScript, 12-210
MidB statement, 12-210
MidBP function
LotusScript, 12-210
MidC function
LotusScript, 12-211
Minus sign (-)
LotusScript, 4-5, 4-13
Minute function
LotusScript, 3-52, 12-212

Mod operator
LotusScript, 4-5, 4-10
Modifier property
JavaMethod class, 11-49, 11-62
JavaProperty class, 11-62
Module level variables
LotusScript, 5-13
Modules
creating, 1-13
limit on symbols, A-4
loading, 1-12
LotusScript, 12-303, 12-306, 12-322, 12-323
using, 1-13
Month function
LotusScript, 3-52, 12-213
Msgbox alias
LotusScript, D-1
Multiplication operator (*)
LotusScript, 4-5, 4-8
Multi-processing
enabling, 10-7
Multi-threading, 10-1

N

Name conflicts
LotusScript, 3-10
Name statement
LotusScript, 12-215
Named constants
LotusScript, 3-13
Names
construction rules, 2-4
of variables, 3-19
Natural logarithm
LotusScript, 12-201
Negation operator (-)
LotusScript, 4-8
Nested For loops
LotusScript, 9-21
New keyword, 8-29
LotusScript, 12-34, 12-73, 12-273
New sub, 5-22, 8-11, 8-14
calling, 8-25
LotusScript, 12-304
Next keyword, 12-217
LotusScript, 9-19, 12-254
NoCase keyword
LotusScript, 12-230
NoPitch keyword
LotusScript, 12-230
Not equal to operator
LotusScript, 4-13
Not operator
LotusScript, 4-5, 4-23
Nothing values
LotusScript, 12-34, 12-71, 12-73, 12-273
Now function
LotusScript, 3-52, 12-216
Null values
LotusScript, 3-14, 12-177, 12-327
Number handling
limits, A-1
LotusScript, 12-1, 12-104, 12-123, 12-169, 12-178, 12-188, 12-261, 12-277
numeric construction, 2-2
Numbers
rounding, 12-169
Numeric conversions
implicit converting, 3-2
LotusScript, 12-19, 12-25, 12-27, 12-30, 12-33, 12-38, 12-50, 12-53, 12-140, 12-216, 12-287, 12-326
Numeric operators. *See* Operators

O

Object arguments to C functions
arrays, 11-20
Object linking and embedding
LotusScript, 11-12
Object reference variables, 8-2, 8-8
declaring, 8-29
variant, 8-32
Object references, 8-13
as an argument, 8-26
LotusScript, 3-2
testing, 8-16
Objects
as an argument, 8-26
bracket notation, 11-4
creating, 8-29, 11-2, 12-48, 12-304
declaring a variable, 8-29
deleting, 8-17, 11-5, 12-71, 12-302
Is operator, 4-38
LotusScript, 12-21, 12-34, 12-73, 12-83, 12-179, 12-181, 12-220, 12-273, 12-332
memory management, 8-18
methods, 8-7
object reference, 8-29
opening, 12-134
passing, 11-20
referring to, 11-3
referring to members, 8-15
Oct function
LotusScript, 12-216
Octal numbers
function, 12-216
numeric construction, 2-2
OLE automation
LotusScript, 11-12
OLE objects
LotusScript language, 12-48, 12-134, 12-179, 12-181
naming rules, 2-4
On Error statement, 7-2, 7-7, 7-12, 7-14
LotusScript, 12-217
On Event statement
LotusScript, 12-220
On...GoSub statement
LotusScript, 9-13, 12-222
On...GoTo statement
LotusScript, 9-12, 12-223
One's complement
LotusScript, 4-5
Open statement
LotusScript, 6-10, 12-225
MIME charset names, E-1
Opening files
LotusScript, 6-10
random, 6-4
sequential, 6-2
Operating system differences, B-1, B-2, B-5
Operators, 4-5
addition, 4-5
And, 4-5, 4-24
arithmetic, 4-2
assignment, 4-2
bitwise, 4-2
Boolean, 4-2
comparison, 4-13
concatenation (& or +), 4-2
described, 4-1
division (/), 4-9
equal sign, 4-5
Eqv, 4-5, 4-28
exponentiation (^), 4-5
floating-point division (/), 4-5
greater than (>), 4-5
greater than or equal to, 4-5
Imp, 4-5, 4-29
integer division, 4-5
integer division (\), 4-10
Is, 8-16, 4-38
IsA, 4-39

less than (<), 4-5
less than or equal to, 4-5
Like operator, 4-31
logical, 4-2
LotusScript, 4-2
Mod, 4-5
multiplication (*), 4-5
negative (-), 4-8
Not, 4-5, 4-23
not equal to, 4-5, 4-31
numeric, 4-5
Or, 4-5, 4-25
precedence, 4-3
relational, 4-2, 4-13
string, 4-31
string concatenation (&), 4-31
subtraction, 4-5
unary minus (-), 4-5
unary plus (+), 4-5
Xor, 4-5, 4-27
Option Base statement
LotusScript, 3-35, 12-229
Option Compare statement
LotusScript, 12-230
Option Compare Text alias
LotusScript, D-1
Option Declare statement
LotusScript, 12-233
Option Explicit statement
LotusScript, 12-233
Option Public statement
LotusScript, 12-233
Or operator
LotusScript, 4-5, 4-25
OS information
LotusScript, 12-85
OS/2 platform differences, B-1
Output
LotusScript, 12-18, 12-204, 12-234
Output keyword
LotusScript, 12-225
Overriding
properties and methods, 8-21

P

Parameters. *See* Arguments
Parentheses
LotusScript, 12-23
Passing arguments, 5-12
C functions, 11-15
LotusScript, 5-4, 11-15
Passing arrays, 11-20
Passing objects, 11-20
Passing strings
LotusScript, 11-18
Passing types, 11-20
Pattern matching
LotusScript, 4-34
Pausing in scripts
LotusScript, 12-281
PI values
LotusScript, 3-14
PIF files
LotusScript, 12-277, 12-279
Pitch keyword
LotusScript, 12-230
Pitch sensitivity
LotusScript, 12-230
Platform differences, B-1, B-5
UNIX, B-2
Platforms
identification, 12-146, 12-172
Plus sign (+)
LotusScript, 4-5, 4-31
Positioning
in a file, 12-198, 12-265, 12-266
Precedence
of an operator, 4-3
Predefined functions. *See* Built-in functions
Preserve keyword
LotusScript, 3-39, 12-246
Print # statement
LotusScript, 6-12, 12-236
Print keyword
LotusScript, 12-225
Print statement
LotusScript, 11-6, 12-234
Private class members
LotusScript, 8-13
Private keyword
external C call, 12-62
forward reference, 12-66
LotusScript, 12-34, 12-44, 12-73, 12-238, 12-299
Procedures, 5-1
maximum argument, A-4
properties, 5-22
sub, 5-17
terminating, 9-30
Process
definition of, 10-1
Processing
an error, 7-2
Product objects
LotusScript, 12-21, 12-71, 12-73, 12-83, 12-273
Products
determining use, 11-1
interacting, 11-9
Product-specific constants
LotusScript, 3-15
Programs
interacting with SendKeys statement, 12-270
LotusScript, 12-3, 12-277, 12-279
Properties
declaring, 5-23
defining, 5-23, 8-9
for Lotus product classes, 11-4
naming rules, 2-4
overriding, 8-21
overview, 5-22
redefining, 8-21
referring to, 11-4
Property Get statement
LotusScript, 5-22
Property Get/Set statements
LotusScript, 12-238
Property keyword
forward reference, 12-66
Property Set statement
LotusScript, 5-22
PropertyName property
JavaProperty class, 11-61
Public class members
LotusScript, 8-13
Public keyword
external C call, 12-62
forward reference, 12-66
LotusScript, 12-34, 12-44, 12-73, 12-233, 12-238, 12-299
Put keyword
LotusScript, 12-225
Put statement, 12-242
Put statement
LotusScript, 6-10, 12-242

Q

Quotation marks
LotusScript, 2-3

R

Random files
accessing, 6-8
defining a record type, 6-5
LotusScript, 6-1, 6-8, 12-129, 12-225, 12-242
opening, 6-4

reading, 6-6, 6-12
writing, 6-5
Random keyword
LotusScript, 12-225
Random numbers
LotusScript, 12-259
Randomize statement
LotusScript, 12-245
Read keyword
LotusScript, 12-225
Reading from files
binary, 6-7
LotusScript, 12-129, 12-155, 12-158, 12-160, 12-163, 12-196
random, 6-6
sequential, 6-3
Read-only files
LotusScript, 12-132, 12-275
Records
fixed length, 6-12
variable length, 6-11
Recovering storage
in a dynamic array, 3-40
Recursion
limiting, A-4
Recursive functions
LotusScript, 5-11
Redefining
a method, 8-21
a property, 8-21
ReDim statement
LotusScript, 3-39, 12-246
Reference
external, 11-18
Reference, argument passing by
LotusScript, 3-2
References, forward, 12-66
Referring to
a method, 11-4
an object, 11-3
bracket notation, 11-4
class members, 8-14
member variables, 8-4
members of an object, 8-15
objects, 8-13
properties, 11-4
Reinitializing
a fixed array, 3-40
Relational operators. *See* Operators
Rem keyword
LotusScript, 12-249
Rem statement
LotusScript, 12-249
Remainder of division
LotusScript, 4-10
Remainders
determining, 4-5
Remarks. *See* Comments
Remove keyword
LotusScript, 12-220
Replace function
LotusScript, 12-251
Reserved words
LotusScript, 2-6
Reset statement
LotusScript, 12-253
Resizing
arrays, 3-39
Resume 0 keyword
LotusScript, 7-13
Resume keyword, 7-13
LotusScript, 12-217, 12-254
Resume Next keyword
LotusScript, 7-12
Resume statement
LotusScript, 7-2, 7-7, 12-254
Return statement
LotusScript, 9-13, 12-255
Return values
for a function, 5-7
LotusScript, 11-25, 12-62, 12-66
of functions, 3-26
REXX, C-1
Right function
LotusScript, 12-256
Right-aligning strings
LotusScript, 12-262
RightBP function
LotusScript, 12-257
RightC function
LotusScript, 12-258
RmDir statement
LotusScript, 12-259
Rnd function
LotusScript, 12-259
Round function
LotusScript, 12-261
RSet statement
LotusScript, 12-262
RTrim function
LotusScript, 12-263
Rules
constructing a script, 2-1
for scripts and statements, 2-1
Run statement
LotusScript, 12-264
Run-time errors, 1-13
LotusScript, 7-1, 7-2, 7-7

S

Scalar data types
currency, 3-1
double, 3-1
integer, 3-1
long, 3-1
LotusScript, 3-1
single, 3-1
string, 3-1
Scalar values
LotusScript, 12-180
Scope
of class members, 8-14
of constants, 3-18
of declarations, 3-10
Screen
printing, 12-234
Script Debugger, 1-15
Script Editor, 1-12
Script modules
creating, 1-13
using, 1-13
Scripts
comments, 9-2
compiler directive, 9-2
compiling, 1-13
constructing rules, 2-1
debugging, 1-15
declaration, 9-2
defined, 1-12
definition statements, 9-2
flow of execution, 9-1
labels, 9-3
limiting, A-4
pausing, 12-281
viewing, 1-12
writing, 1-12
Searching
in strings, 12-164, 12-166 to 12-168
Second function
LotusScript, 3-52, 12-264
Seeding
the random number generator, 12-245
Seek function
LotusScript, 6-12, 12-265
Seek statement
LotusScript, 6-12, 12-266
Select Case statement
LotusScript, 9-8, 12-268

SendKeys statement
LotusScript, 11-9, 12-270
Separators
LotusScript, 2-10
Sequential files
accessing, 6-8
LotusScript, 6-1, 6-8, 12-155, 12-158, 12-196, 12-225, 12-236, 12-333
opening, 6-2
reading, 6-3, 6-10
writing, 6-3, 6-12
Set keyword
LotusScript, 12-238, 12-273
Set statement
LotusScript, 8-29, 12-273
SetAttr alias
LotusScript, D-1
SetAttr function
LotusScript, 12-275
SetFileAttr function
LotusScript, 12-275
setValue method
JavaProperty class, 11-66
Sgn function
LotusScript, 12-277
Shadowing
LotusScript, 3-10
variables, 5-13
Shared keyword
LotusScript, 12-225
Shared libraries
external C call, 11-13, 12-62
Shell function
LotusScript, 11-9, 12-277, 12-279
Shellid function
LotusScript, 11-9, 12-277, 12-279
Signature property
JavaMethod class, 11-50
Sin function
LotusScript, 12-280
Single data type
LotusScript, 3-1, 12-281
Slash (/)
LotusScript, 4-5
Slash (/) notation
LotusScript, 11-75
Sleep statement
LotusScript, 12-281
Space function
LotusScript, 12-282
Spaces
LotusScript, 12-204, 12-263, 12-282, 12-283, 12-307, 12-313
Spc function
LotusScript, 12-283
Special characters
LotusScript, 2-10
Split function
LotusScript, 12-284
Sqr function
LotusScript, 12-286
Square roots
LotusScript, 12-286
StackTrace property
JavaError class, 11-44
Statement labels
LotusScript, 9-3
Statements
comments, 9-2
compiler directive, 9-2
construction rules, 2-1
continuation character (_), 2-1
declaration, 9-2
defining, 9-2
flow of execution, 9-1
in run-time errors, 7-6
separation character (:), 2-1
Statements, LotusScript. *See* LotusScript statements
Static keyword
for data type, 12-73
forward reference, 12-66
LotusScript, 12-238, 12-299
Stepping through applications, 1-15
Stop statement, 1-15
LotusScript, 12-286
Str function
LotusScript, 12-287
StrCom function
LotusScript, 12-288
StrComp alias
LotusScript, D-1
StrCompare function
LotusScript, 12-288
StrConv function
LotusScript, 12-289
String arguments to C functions, 11-18
LotusScript, 11-20
String concatenation operator (&)
LotusScript, 4-31
String conversions
LotusScript, 12-287, 12-289
String data type
LotusScript, 3-1, 12-297
String function
LotusScript, 12-298
String handling
changing case, 12-185, 12-320
comparing, 12-230, 12-288
concatenation operator (&), 4-31
conversion, 12-287, 12-289, 12-326
extracting, 12-186 to 12-188, 12-208, 12-210, 12-211, 12-256, 12-257, 12-258
length, 12-188, 12-190, 12-191, 12-193
limits, A-2
LotusScript, 12-325
pattern matching, 4-34
position in string, 12-164, 12-166 to 12-168
replacing, 12-209
searching, 12-291 to 12-294
setting to string variable, 12-203, 12-262
spaces, 12-282
string construction rules, 2-3
String function, 12-298
trimming, 12-204, 12-263, 12-313
String operators. *See* Operators
Strings
fixed length, 3-21
passing, 11-18
variable length, 3-21
StrLeft function
LotusScript, 12-291
StrLeftBack function
LotusScript, 12-292
StrRight function
LotusScript, 12-293
StrRightBack function
LotusScript, 12-294
StrToken function
LotusScript, 12-295
Sub Delete, 5-22, 8-11, 8-17
calling, 8-25
LotusScript, 12-302
Sub Initialize
LotusScript, 5-21, 12-303
Sub keyword
external C call, 12-62
forward reference, 12-66
LotusScript, 12-299
Sub New, 8-11, 8-14
calling, 8-25
LotusScript, 5-22, 12-304
Sub Terminate, 5-22
LotusScript, 12-306
Subprograms. *See* Subs
Subs, 5-17

declaring, 5-19
defining, 5-19
deleting, 12-34, 12-71
described, 5-17
executing, 5-19
in a class, 8-10
LotusScript, 12-23, 12-138, 12-299
maximum argument, A-4
overriding, 8-21
signature, 5-17
terminating, 9-30
Subscripts
for a list, 3-42
for an array, 3-29
Subtraction operator (-)
LotusScript, 4-5, 4-13
Suffix characters
constants, 3-17
for data type, 12-73
LotusScript, 12-69
omitting, 3-17
Symbolic constants
LotusScript, 12-44
Symbols
limiting, A-4
Synchronization, 10-1, 10-3
LotusScript, 12-281
Synchronization functions, 10-3
System date
LotusScript, 12-57, 12-58
System files
LotusScript, 12-79, 12-132, 12-275

T

Tab function
LotusScript, 12-307
Tabs
in scripts, 2-1
Tag names
in a list, 12-197
Tan function
LotusScript, 12-309
Temporary module
LotusScript, 12-96
Terminate sub
LotusScript, 5-22, 12-306
Terminating
a loop, 9-32
Terminating functions
LotusScript, 9-30
Terminating procedures
LotusScript, 9-30
Terminating subs
LotusScript, 9-30
Termination statements
Exit, 12-98
LotusScript, 12-84
Testing
for data type, 3-17
Text keyword
LotusScript, 12-230
Then keyword
LotusScript, 12-144, 12-145
Thread safe code, 10-7
Threading functions
LotusScript, 12-136
Threads
common problems, 10-7
definition of, 10-1
Tilde escape character (~)
LotusScript, 2-4
Time function
LotusScript, 3-52, 12-309
Time slice
giving up, 12-281
multi-threading, 10-1
Time statement
LotusScript, 3-52, 12-310
TimeDate
values, 3-51
TimeNumber function
LotusScript, 3-52, 12-310
Timer function
LotusScript, 3-52, 12-311
TimeSerial alias
LotusScript, D-1
TimeSerial function
LotusScript, 12-310
TimeValue function
LotusScript, 3-52, 12-312
To keyword
Lock and Unlock statements, 12-199
LotusScript, 12-73, 12-246, 12-268
Today function
LotusScript, 3-52, 12-312
Trigonometric functions
LotusScript, 12-2, 12-16, 12-17, 12-46, 12-280, 12-309
Trim function
LotusScript, 12-313
Trimming spaces
LotusScript, 12-204, 12-263, 12-313
Trimming spaces from strings
LotusScript, 12-125
True values
LotusScript, 3-14, 3-50
Type arguments to C functions, 11-20
Type property
JavaProperty class, 11-63
Type statement
LotusScript, 12-314
TypeName function
LotusScript, 3-17, 3-41, 12-317
Types
naming rules, 2-4
passing, 11-20

U

UBound function
LotusScript, 12-319
UCase function
LotusScript, 12-320
UChr function
LotusScript, 12-321
Unary minus operator (-)
LotusScript, 4-5
Unary plus operator (+)
LotusScript, 4-5
Uni function
LotusScript, 12-321
Unicode characters
LotusScript, 12-62, 12-321, 12-325
Unicode keyword
LotusScript, 11-20
UNIX platform differences, B-2
Unknown values
LotusScript, 12-181
Unlock statement
LotusScript, 12-199, 12-322
Until keyword
LotusScript, 9-15, 12-81
Upper bounds
fixed array, 3-34
LotusScript, 3-30
Use statement, 1-13
LotusScript, 8-8, 12-322
UseLSX statement
LotusScript, 12-323
User interactions
LotusScript, 11-6
User-defined classes
described, 8-1
LotusScript, 3-2
User-defined constants
LotusScript, 3-15
User-defined data types, 8-8, 11-24
declaring, 8-3
defining, 8-2

described, 8-1
LotusScript, 3-2, 12-314
naming rules, 2-4
using, 8-5
User-defined variables, 11-24
UString function
LotusScript, 12-325

V

V_IUNKNOWN value
LotusScript, 12-181
Val function
LotusScript, 12-326
Values, 11-18, 11-20
Boolean, 3-50
Default data type, 3-17
EMPTY, 3-14
FALSE, 3-14
for literal numbers, 2-2
literal string, 2-3
NOTHING, 3-14
NULL, 3-14
PI, 3-14
TRUE, 3-14
Variable length records
LotusScript, 6-11
Variable length strings
LotusScript, 3-21
Variables
data type, 3-19
declaring, 8-9
declaring data type, 8-3
declaring explicitly, 3-19
declaring implicitly, 3-24
declaring object reference, 8-29
declaring two or more at once, 3-23
default value, 3-21
defining, 8-3
environment, 11-9
for a list, 3-42
for an array, 3-29
in a loop control expression, 9-19
initializing, 8-14
lifetime, 3-9
local, 5-13
LotusScript, 12-73, 12-194, 12-203, 12-233, 12-262, 12-273
module level, 5-13
naming, 3-19
naming rules, 2-4
object reference, 8-8
scope, 3-9
shadowing, 5-13
string, 3-21
user-defined in C language function calls, 11-24
variants, 3-48
Variant data type, 8-32
data type converting, 3-2
LotusScript, 3-2, 3-48, 12-54, 12-327
Variants
date and time value, 3-51
referring to, 3-55
valid date range, 3-51
VarType alias
LotusScript, D-1
VarType function
LotusScript, 12-54
Vertical bars (|)
LotusScript, 2-3
Volume labels
LotusScript, 12-79, 12-132, 12-275

W

Waiting
LotusScript, 12-281
Web agents
multi-threads, 10-1
serial, 10-2
threaded, 10-2
WeekDay function
LotusScript, 3-52, 12-329
Wend keyword
LotusScript, 9-30
While keyword
LotusScript, 12-81
While loops
LotusScript, 9-30
While statement
LotusScript, 9-30, 12-330
White space
in LotusScript, 12-125, 12-263
in scripts, 2-1
removing, 12-204
Width # statement
LotusScript, 12-331
Wildcards
file name, 12-79
Like operator, 4-34
Windows
LotusScript, 12-3, 12-270, 12-277, 12-279
With statement
LotusScript, 8-15, 12-332
Write # statement
LotusScript, 6-11, 12-333
Write keyword
LotusScript, 12-225
Writing to files
binary, 6-7
LotusScript, 12-199, 12-236, 12-242, 12-333
random, 6-5
sequential, 6-3

X

Xor operator
LotusScript, 4-5, 4-27

Y

Year function
LotusScript, 3-52, 12-335
Yield function
LotusScript, 11-9, 12-337
Yield statement
LotusScript, 11-9, 12-337