

Andrew Pitonyak



OpenOffice.org Macros Explained

OpenOffice.org Macros Explained

OOME Fourth Edition

Last Modified

Wednesday, October 17, 2018 at 12:36:37 PM

Document Revision: 634

General Information

The contents of this document is Copyright 2011 - 2018 by Andrew D. Pitonyak.

I created a second edition that was never released, which is why I refer to this book as the fourth edition.

I have been repeatedly asked about the cover page so.... I am neither an artist nor a designer, the cover was created by Amanda Barham (see <http://www.amandabarham.com>). She did an awesome job.

Many people contributed to this document in one form or another. Feedback from community members helps greatly. For reasons I don't understand, my wife allows me to take time away from my family to work on this document, so, I suppose that the person to really thank is her (send her enough money that she can get a fully body massage and she may force me to work on this document).

While taking this document to the latest version, many errors, corrections, and suggestions were provided by Volker Lenhardt who is providing a full translation into German. This document has been greatly improved based on his work.

Thank you Volker Lenhardt (<https://www.uni-due.de/~abi070/ooo.html>).

Table of Contents

General Information	1
Table of Contents	i
1. Introduction and housekeeping	13
1.1. Comments from the author	13
1.2. Environment and comments	13
2. Getting Started	15
2.1. Macro storage	15
2.1.1. Library container	16
2.1.2. Libraries	17
2.1.3. Modules and dialogs	18
2.1.4. Key points	18
2.2. Creating new modules and libraries	18
2.3. Macro language	20
2.4. Create a module in a document	20
2.5. Integrated Debugging Environment	24
2.6. Enter the macro	27
2.7. Run a macro	28
2.8. Macro security	29
2.9. Using breakpoints	31
2.10. How libraries are stored	33
2.11. How documents are stored	34
2.12. Conclusion	34
3. Language Constructs	35
3.1. Compatibility with Visual Basic	36
3.2. Compiler options and directives	37
3.3. Variables	37
3.3.1. Constant, subroutine, function, label, and variable names	38
3.3.2. Declaring variables	38
3.3.3. Assigning values to variables	41
3.3.4. Boolean variables are True or False	41
3.3.5. Numeric variables	42
Integer variables	44
Long Integer variables	44
Currency variables	45
Single variables	45
Double variables	45
3.3.6. String variables contain text	46
3.3.7. Date variables	47
3.3.8. Create your own data types	48
3.3.9. Declare variables with special types	49
3.3.10. Object variables	50
3.3.11. Variant variables	50
3.3.12. Constants	51
3.4. The With statement	52

3.5. Arrays	53
3.5.1. Changing the dimension of an array	56
3.5.2. The unexpected behavior of arrays	58
3.6. Subroutines and functions	60
3.6.1. Arguments	61
Pass by reference or by value	62
Optional arguments	64
Default argument values	66
3.6.2. Recursive routines	66
3.7. Scope of variables, subroutines, and functions	67
3.7.1. Local variables defined in a subroutine or function	68
3.7.2. Variables defined in a module	69
Global	69
Public	69
Private or Dim	70
3.8. Operators	71
3.8.1. Mathematical and string operators	72
Unary plus (+) and minus (-)	72
Exponentiation (^)	73
Multiplication (*) and Division (/)	74
Remainder after division (MOD)	74
Integer division (\)	75
Addition (+), subtraction (-), and string concatenation (& and +)	76
3.8.2. Logical and bit-wise operators	77
AND	79
OR	80
XOR	80
EQV	81
IMP	82
NOT	82
3.8.3. Comparison operators	83
3.9. Flow control	83
3.9.1. Define a label as a jump target	84
3.9.2. GoSub	84
3.9.3. GoTo	85
3.9.4. On GoTo and On GoSub	85
3.9.5. If Then Else	86
3.9.6. IIf	87
3.9.7. Choose	87
3.9.8. Select Case	88
Case expressions	88
If Case statements are easy, why are they frequently incorrect?	89
Writing correct Case expressions	90
3.9.9. While ... Wend	92

3.9.10.	Do ... Loop	93
	Exit the Do Loop	93
	Which Do Loop should I use?	94
3.9.11.	For ... Next	95
3.9.12.	Exit Sub and Exit Function	96
3.10.	Error handling using On Error	96
3.10.1.	CVErr	98
3.10.2.	Ignore errors with On Error Resume Next	99
3.10.3.	Clear an error handler with On Error GoTo 0	99
3.10.4.	Specify your own error handler with On Error GoTo Label	100
3.10.5.	Error handlers — why use them?	102
3.11.	Conclusion	104
4.	Numerical Routines	105
4.1.	Trigonometric functions	106
4.2.	Rounding errors and precision	109
4.3.	Mathematical functions	111
4.4.	Numeric conversions	113
4.5.	Number to string conversions	119
4.6.	Simple formatting	119
4.7.	Other number bases, hexadecimal, octal, and binary	120
4.8.	Random numbers	123
4.9.	Conclusion	125
5.	Array Routines	126
5.1.	How big can an array be?	127
5.2.	Array() quickly builds a one-dimensional array with data	127
5.3.	DimArray creates empty multi-dimensional arrays	129
5.4.	Change the dimension of an array	130
5.5.	Array to String and back again	132
5.6.	Array inspection functions	133
5.7.	Conclusion	136
6.	Date Routines	137
6.1.	Compatibility issues	138
6.2.	Retrieve the current date and time	138
6.3.	Dates, numbers, and strings	138
6.4.	Locale formatted dates	140
6.5.	ISO 8601 dates	141
6.6.	Problems with dates	142
6.7.	Extract each part of a date	146
6.8.	Date arithmetic	152
6.9.	Assembling dates from components	152
6.10.	Measuring elapsed time over short intervals	154
6.11.	How fast does this run? A real-world example!	155
6.12.	Long time intervals and special dates	159
6.13.	Conclusion	160

7. String Routines	161
7.1. ASCII and Unicode values	163
7.2. Standard string functions	167
7.3. Locale and strings	170
7.4. Substrings	171
7.5. Replace	173
7.6. Aligning strings with LSet and RSet	174
7.7. Fancy formatting with Format	175
7.8. Converting data to a string	180
7.9. Advanced searching (regular expressions)	182
7.10. Conclusion	183
8. File Routines	184
8.1. Using URL notation to specify a file	185
8.2. Directory manipulation functions	186
8.3. File manipulation functions	187
8.4. File attributes, bitmasks, and binary numbers	191
8.5. Obtaining a directory listing	192
8.6. Open a file	194
8.7. Information about open files	197
8.8. Reading and writing data	199
8.9. File and directory related services	206
8.9.1. Path Settings and Singletons	206
What is a Singleton?	207
Back to path settings	207
8.9.2. Path Substitution	211
8.9.3. Simple File Access	213
8.9.4. Streams	214
8.9.5. Pipes	217
8.10. Conclusion	218
9. Miscellaneous Routines	219
9.1. Display and color	219
9.1.1. Determine the GUI type	219
9.1.2. Determine pixel size (in twips)	220
9.1.3. Use color functions	221
9.2. Flow control	224
9.2.1. Return an argument	224
9.2.2. Pause or end the macro	226
9.2.3. Dynamic Link Libraries	226
9.2.4. Calling external applications	228
9.2.5. Dynamic Data Exchange	229
9.3. User input and output	231
9.3.1. Simple output	231
9.3.2. Multi-line output	232
9.3.3. Prompting for input	235

9.4.	Error-related routines	236
9.5.	Miscellaneous routines	237
9.6.	Partition	241
9.7.	Inspection and identification of variables	242
9.8.	Routines you should not use and other curiosities	248
9.9.	Routines I do not understand	249
9.10.	New Compatibility Functions	250
9.11.	Conclusion	251
10.	Universal Network Objects	252
10.1.	Base types and structures	253
10.2.	UNO interface	255
10.3.	UNO service	257
10.3.1.	Setting a Read-Only Value	260
10.4.	Context	261
10.5.	Inspecting Universal Network Objects	261
10.6.	Using the type description manager	267
10.7.	Use Object or Variant	270
10.8.	Comparing UNO variables	271
10.9.	Built-in global UNO variables	272
10.10.	Creating UNO values for OOO internals	276
10.11.	Finding objects and properties	277
10.12.	UNO listeners	278
10.12.1.	Your first listener	279
10.12.2.	A complete listener: selection change listener	279
10.13.	Creating a UNO dialog	282
10.14.	Conclusion	286
11.	The Dispatcher	287
11.1.	The environment	287
11.1.1.	Two different methods to control LO	287
11.1.2.	Finding dispatch commands	289
	Use the WIKI for OpenOffice.org	289
	Read the source	289
	Probe the interface	290
	Read source code	292
11.2.	Writing a macro using the dispatcher	293
11.3.	Dispatch failure – an advanced clipboard example	293
11.4.	Conclusion	294
12.	StarDesktop	295
12.1.	The Frame service	295
12.1.1.	The XIndexAccess interface	296
12.1.2.	Find frames with FrameSearchFlag constants	297
12.2.	The XEventBroadcaster interface	298
12.3.	The XDesktop interface	299
12.3.1.	Closing the desktop and contained components	299

12.3.2.	Enumerating components using XEnumerationAccess	300
12.3.3.	Current component	301
12.3.4.	Current component (again)	301
12.3.5.	Current frame	302
12.4.	Load a document	303
12.4.1.	Named arguments	307
12.4.2.	Loading a template	309
12.4.3.	Enabling macros while loading a document	310
12.4.4.	Importing and exporting	311
12.4.5.	Filter names	312
12.4.6.	Loading and saving documents	318
12.4.7.	Error handling while loading a document	320
12.5.	Conclusion	320
13.	Generic Document Methods	321
13.1.	Service Manager	321
13.2.	Services and interfaces	322
13.3.	Getting and setting properties	324
13.4.	Document properties	326
13.4.1.	Document properties from a closed document	327
13.4.2.	Custom properties	328
13.4.3.	Deprecated document information object	328
13.5.	List events	329
13.5.1.	Registering your own listener	331
13.5.2.	Intercepting dispatch commands	331
13.6.	Link targets	334
13.7.	Accessing view data: XViewDataSupplier	336
13.8.	Close a document: XCloseable	337
13.9.	Draw Pages: XDrawPagesSupplier	338
13.9.1.	Draw and Impress	338
13.9.2.	Draw lines with arrows in Calc	341
13.9.3.	Writer	343
13.10.	The model	344
13.10.1.	Document arguments	345
13.11.	Saving a document	347
13.12.	Manipulating styles	350
13.12.1.	Style utilities	356
13.13.	Dealing with locale	360
13.14.	Enumerating printers	369
13.15.	Printing documents	369
13.15.1.	Printing Writer documents	373
13.15.2.	Printing Calc documents	375
13.15.3.	A Calc example with a Print listener	376
13.15.4.	Print examples by Vincent Van Houtte	379
13.16.	Creating services	388

13.17.	Document settings	390
13.18.	The coolest trick I know	392
13.19.	Converting to a URL in other languages	392
13.20.	Conclusion	392
14.	Writer Documents	394
14.1.	Basic building blocks	395
14.1.1.	Primary text content: the XText interface	395
14.1.2.	Text ranges: the XTextRange interface	396
14.1.3.	Inserting simple text	398
14.1.4.	Text content: the TextContent service	398
14.2.	Enumerating paragraphs	400
14.2.1.	Paragraph properties	401
	Insert a page break	405
	Set the paragraph style	405
14.2.2.	Character properties	406
14.2.3.	Enumerating text sections (paragraph portions)	410
14.3.	Graphics	412
14.4.	Paste HTML then embed linked graphics	415
14.5.	Cursors	417
14.5.1.	View cursors	417
14.5.2.	Text (non-view) cursors	419
14.5.3.	Using cursors to traverse text	421
	Keep the view cursor and text cursor in sync	423
14.5.4.	Accessing content using cursors	424
14.6.	Selected text	427
14.6.1.	Is text selected?	427
14.6.2.	Selected text: Which end is which?	429
14.6.3.	Selected text framework	430
14.6.4.	Remove empty spaces and lines: A larger example	431
	What is white space?	432
	Rank characters for deletion	432
	Use the standard framework	433
	The worker macro	434
14.6.5.	Selected text, closing thoughts	436
14.7.	Search and replace	436
14.7.1.	Searching selected text or a specified range	437
	Searching for all occurrences	438
14.7.2.	Searching and replacing	439
14.7.3.	Advanced search and replace	439
14.8.	Text content	443
14.9.	Text tables	443
14.9.1.	Using the correct text object	446
14.9.2.	Methods and properties	446
14.9.3.	Simple and complex tables	449

14.9.4.	Tables contain cells	452
14.9.5.	Using a table cursor	454
14.9.6.	Formatting a text table	457
14.10.	Text fields	458
14.10.1.	Text master fields	467
14.10.2.	Creating and adding text fields	469
14.11.	Bookmarks	473
14.12.	Sequence fields, references, and formatting	473
14.12.1.	Formatting numbers and dates	474
	List formats known to the current document	474
	Find or create a numeric format	475
	Default formats	475
14.12.2.	Create a field master	476
14.12.3.	Insert a sequence field	477
14.12.4.	Replace text with a sequence field	477
14.12.5.	Create a GetReference field	479
14.12.6.	Replace text with a GetReference field	480
14.12.7.	The worker that ties it all together	482
14.13.	Table of contents	483
14.14.	Conclusion	489
15.	Calc Documents	491
15.1.	Accessing sheets	492
15.2.	Sheet cells contain the data	494
15.2.1.	Cell address	495
15.2.2.	Cell data	495
15.2.3.	Cell properties	497
15.2.4.	Cell annotations	502
15.3.	Uninterpreted XML Attributes	504
15.4.	Sheet cell ranges	505
15.4.1.	Sheet cell range properties	506
	Validation settings	506
	Conditional formatting	509
15.4.2.	Sheet cell range services	510
	Retrieving cells and ranges	510
	Querying cells	510
	Finding non-empty cells in a range	511
	Using complex queries	512
	Query Precedents and Dependents	514
	Query Column Differences	514
15.4.3.	Searching and replacing	515
15.4.4.	Merging cells	515
15.4.5.	Retrieving, inserting, and deleting columns and rows	516
15.4.6.	Retrieving and setting data as an array	518
15.4.7.	Computing functions on a range	519

15.4.8.	Clearing cells and cell ranges	520
15.4.9.	Automatic data fill	520
15.4.10.	Array formulas	522
15.4.11.	Computing multiple functions on a range	523
15.4.12.	Cells with the same formatting	526
15.4.13.	Sorting	527
15.5.	Sheets	531
15.5.1.	Linking to an external spreadsheet	532
15.5.2.	Finding dependencies by using auditing functions	534
15.5.3.	Outlines	536
15.5.4.	Copying, moving, and inserting cells	537
15.5.5.	Copy data between documents	538
	Data functions	538
	Clipboard	538
	Transferable content	539
15.5.6.	Data pilot and pivot tables	539
	A data pilot example	540
	Generating the data	541
	Creating the data pilot table	542
	Manipulating data pilot tables	544
	Data pilot fields	544
	Filtering data pilot fields	545
	Tables	546
15.5.7.	Sheet cursors	547
15.6.	Calc documents	549
15.6.1.	Named range	549
15.6.2.	Database range	552
15.6.3.	Filters	553
15.6.4.	Protecting documents and sheets	556
15.6.5.	Controlling recalculation	556
15.6.6.	Using Goal Seek	556
15.7.	Write your own Calc functions	557
15.8.	Using the current controller	559
15.8.1.	Selected cells	559
	Enumerating the selected cells	560
	Selecting text	562
	The active cell	562
15.8.2.	General functionality	563
15.9.	Control Calc from Microsoft Office	564
15.10.	Accessing Calc functions	565
15.11.	Finding URLs in Calc	565
15.12.	Importing and Exporting XML Files Using Calc	566
15.12.1.	Read an XML file	567
15.12.2.	Write XML File	574

15.13.	Charts	584
15.14.	Conclusion	592
16.	Draw and Impress Documents	593
16.1.	Draw pages	594
16.1.1.	Generic draw page	596
16.1.2.	Combining shapes	597
16.2.	Shapes	600
16.2.1.	Common attributes	603
	Drawing text service	609
	MeasureShape	611
	Drawing line properties	612
	Filling space with a ClosedBezierShape	613
	Shadows and a RectangleShape	616
	Rotation and shearing	617
16.2.2.	Shape types	618
	Simple lines	618
	PolyLineShape	620
	PolyPolygonShape	621
	RectangleShape and TextShape	622
	EllipseShape	623
	Bezier curves	625
	ConnectorShape	628
	Creating your own glue points	631
	Adding arrows by using styles	631
	Insert a TableShape	633
16.3.	Forms	633
16.4.	Presentations	636
16.4.1.	Presentation draw pages	638
16.4.2.	Presentation shapes	639
16.5.	Conclusion	641
17.	Library Management	642
17.1.	Accessing libraries using Basic	642
17.2.	Libraries contained in a document	646
17.3.	Writing an installer	647
17.4.	Conclusion	648
18.	Dialogs and Controls	649
18.1.	My first dialog	649
18.1.1.	The Properties dialog	652
18.1.2.	Starting a dialog from a macro	654
18.1.3.	Assign an event handler	655
18.2.	Dialog and control paradigm	658
18.2.1.	Dialog and control similarities	659
18.2.2.	Dialog-specific methods	661
18.2.3.	The dialog model	662

18.3.	Controls	664
18.3.1.	Control button	666
18.3.2.	Check box	668
18.3.3.	Radio button	669
18.3.4.	Group box	671
18.3.5.	Fixed line control	671
18.3.6.	Combo box	671
18.3.7.	Text edit controls	673
	Currency control	675
	Numeric control	676
	Date control	676
	Time control	680
	Formatted control	681
	Pattern control	685
	Fixed text control	686
	File control	687
18.3.8.	Image control	689
18.3.9.	Progress control	689
18.3.10.	List box control	690
18.3.11.	Scroll bar control	691
18.4.	Using the Step property for multi-page AutoPilot dialogs	694
18.5.	The object inspector example	694
18.5.1.	Utility subroutines and functions	695
	Identifying and removing white space from a string	695
	Convert a simple object to a string	696
	Object inspection using Basic methods	700
	Sort an array	701
18.5.2.	Creating a dialog at run time	702
18.5.3.	Listeners	706
	Radio buttons	706
	Inspect selected	706
	Inspect previous	708
18.5.4.	Obtaining the debug information	708
18.5.5.	Getting property values	713
18.6.	Conclusion	715
19.	Sources of Information	716
19.1.	Help pages included with OpenOffice.org	716
19.2.	Macros included with OpenOffice.org	717
19.3.	Web sites	717
19.3.1.	Reference material	717
19.3.2.	Macro examples	717
19.3.3.	Miscellaneous	718
19.1.	http://www.openoffice.org/api/ or http://api.libreoffice.org	718
19.2.	Mailing lists and newsgroups	719

19.3. How to find answers	720
19.4. Conclusion	721

1. Introduction and housekeeping

First, there was the first edition of OpenOffice.org Macros Explained (OOME). A few years later I produced the second edition, updated to match OpenOffice.org (OOo) version 2.x, but the second edition was never released. Now, I feel that it is time for a fourth edition.

I use the term OOo generically even as opposed to strictly calling out Apache OpenOffice (AOO) or LibreOffice (LO).

There is, unfortunately, more capability than I have time or room to document; much is missing. You should use this book, therefore, as a reference with numerous examples, but, always remember that OOo is constantly changing and supporting more features.

The document contains buttons that trigger the macros in the text. This is fabulous while reading the original source, but it provides for undesirable artifacts in printed form; sorry about that.

1.1. Comments from the author

I am the primary author of this document. I do not make a living working with OOo, and nothing in here is related to my primary day job. In other words, I am simply another member of the OOo community who does this without remuneration.

I receive numerous requests for help because I am highly visible in the OOo community. Unfortunately, I am over-time-committed, and it is difficult to provide personal help to all. I enjoy helping people in my non-existent spare time, but, be certain to use existing material, mailing lists, and forums if possible. I occasionally provide solutions on a commission basis, but I lack the time for large tasks.

I appreciate comments and bug reports. If you have something interesting that you think should be included, let me know. If you want to write an entire section or chapter, do so. I will likely highly edit what you produce. Please, provide feedback and suggestions by sending an email to andrew@pitonyak.org.

1.2. Environment and comments

My primary work is performed using 64-bit Fedora Linux with the en-US locale. Gnome is my desktop environment, which affects screen shots. Little to no testing has been done by me in other environments. Although I use both LibreOffice (LO) and Apache OpenOffice (AOO), I mostly use LO at this time. I use the term OpenOffice, OO, or OOo to generically refer to either product.

AOO and LO are changing independently, and as time passes, the APIs, features, and user interfaces become less similar. The result is a macro that works in LO may fail in AOO, and vice versa. This also means that screen shots and menu paths differ more and more as time passes.

Rapid development means that changes occur faster than time permits me to produce documentation; I receive no remuneration for my time spent on this document.

The best way to find out what works on your chosen platform is testing and inspection. By inspection, I mean that you should inspect object instances to determine the supported properties, methods, constants, enumerations, structures, services, and interfaces.

2. Getting Started

In OpenOffice.org (OOo), macros and dialogs are stored in documents and libraries. The included integrated development environment (IDE) is used to create and debug macros and dialogs. This chapter introduces the basic concepts of starting the IDE and creating macros by showing the steps to produce a simple macro, which displays the text “Hello World” on the screen.

A macro is a saved sequence of commands or keystrokes stored for later use. An example of a simple macro is one that “types” your address. Macros support commands that allow a variety of advanced functions, such as making decisions (for example, if the balance is less than zero, color it red; if not, color it black), looping (while the balance is greater than zero, subtract 10), and even interacting with a person (asking the user for a number). Some of these commands are based on the BASIC programming language. (BASIC is an acronym for Beginner’s All-purpose Symbolic Instruction Code.) It is common to assign a macro to a keystroke or toolbar icon so that it can be quickly started.

A dialog – or dialog box – is a type of window used to have a “dialog” with a user. The dialog may present information to the user, or obtain input from the user. You can create your own dialogs and store them in a module with your macros.

The OpenOffice.org macro language is very flexible, allowing automation of both simple and complex tasks. Although writing macros and learning about the inner workings of OpenOffice.org can be a lot of fun, it is not always the best approach. Macros are especially useful when you have to do a task the same way over and over again, or when you want to press a single button to do something that normally takes several steps. Once in a while, you might write a macro to do something you can’t otherwise do in OpenOffice.org, but in that case you should investigate thoroughly to be sure OOo cannot do it. For instance, a common request on some of the OpenOffice.org mailing lists is for a macro that removes empty paragraphs. This functionality is provided with AutoFormat (select **Tools > Auto Correct > AutoCorrect Options > Options** tab and check *Remove blank paragraphs*). It is also possible to use regular expressions to search for and replace empty space. There is a time and a purpose for macros, and a time for other solutions. This chapter will prepare you for the times when a macro is the solution of choice.

2.1. Macro storage

In OpenOffice.org, routines that are logically related are stored in a module. For example, a module might contain routines for finding common mistakes that require editing. Logically related modules are stored in a library, and libraries are stored in library containers. The OpenOffice.org application can act as a library container, as can any OOo document. Simply stated, the OpenOffice.org application and every OpenOffice.org document can contain libraries, modules, and macros.

Container	A library container contains zero or more libraries.
Library	A library contains zero or more modules and dialogs.
Module	A module contains zero or more subroutines or functions.

2.1.1. Library container

An OOo document is a library container, as is the application itself. If a specific document requires a macro, it is useful to store that macro in the document. The advantage is that the macro stays with the document. This is also an easy way to send macros to others.

If several documents use the same macros, however, then every document will have a copy, and, if you change the macro, then you must change it in every document that contains the macro.

Macros contained in a document are visible only to that document. It is not easy, therefore, to call a macro in a document from outside of that document.

TIP

Do not (except in rare exceptions) store macros that will be called from outside a document in a document; because macros contained in a document are visible only to that document.

The application library container has two primary components, macros distributed with OOo, and macros that you create. The Basic Macros dialog shows your macros in a container named “My Macros”, and those distributed as “LibreOffice Macros” (see Figure 1). LibreOffice Macros are stored in a directory with the application, and My Macros are stored in your user directories.

Use **Tools > Macros > Organize Macros > LibreOffice Basic** to open the Basic Macros dialog (see Figure 1). The library containers are the highest level objects shown in the “Macro from” area.

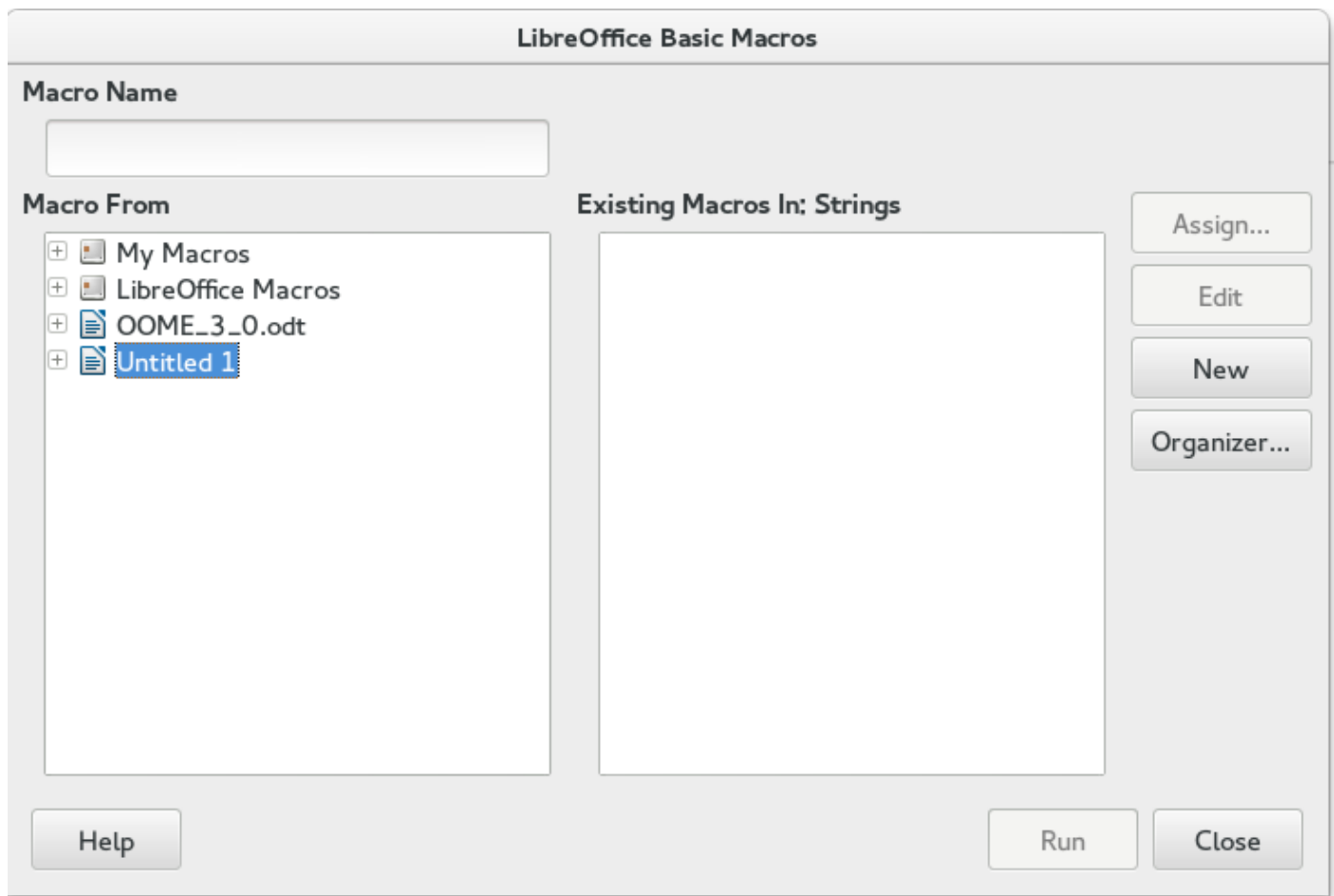


Figure 1. Use the Basic Macros dialog to create new macros and organize libraries.

2.1.2. Libraries

A *library container* contains one or more libraries, and a *library* contains one or more modules and dialogs. Double-click on a library container in Figure 1 to see the contained libraries. Double-click on a library to load the library and see the contained modules and dialogs.

The Basic Macros dialog uses a different icon to distinguish loaded libraries; in Figure 2, AndrewTestLib is loaded, the other libraries are not.

TIP The icons and colors that you see on your computer may be different than those shown in the screen shots. Different versions of OOo may use different icons and colors, and more than one icon set is supported. Use **Tools > Options > LibreOffice > View** to change the icon size and style.

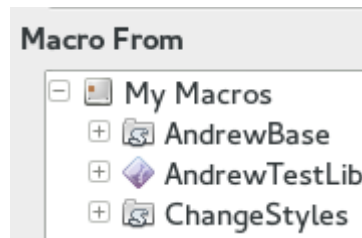


Figure 2. Loaded libraries are shown differently.

2.1.3. Modules and dialogs

A module is typically used to group similar functionality at a lower level than a library. The macros are stored in the modules. To create a new module, select a library and click New.

2.1.4. Key points

Things to consider:

- You can import libraries from one library container to another.
- Import a module by importing the library that contains the module. It is not possible to simply import a single module using the GUI.
- Use descriptive names for libraries, modules, and macros. Descriptive names reduce the likelihood of a name collision, which hampers library import.
- The Standard library is special; it is automatically loaded so the contained macros are always available.
- The Standard library is automatically created and *cannot* be imported.
- Macros contained in a library are not available until after the library is loaded. The Standard library is automatically loaded.
- The Basic Macro organizer dialog allows you to create new modules, but not new libraries.

The key points listed above have certain consequences; for example, I rarely store macros in the Standard library because I cannot import the library to another location. My usual usage for the standard library is for macros called from buttons in a document. The macros in the standard library then load the actual work macros in other libraries, and call them.

2.2. Creating new modules and libraries

The New button on the Basic Macros dialog always creates a new subroutine in the selected library (see Figure 1). A new module is created if the library does not yet contain a module.

From the Macros dialog, click the Organizer button to open the LibreOffice Basic Macro Organizer dialog (see Figure 3). The Modules and Dialogs tabs are almost identical. Use the Modules or Dialogs tab to create, delete, and rename modules or dialogs.

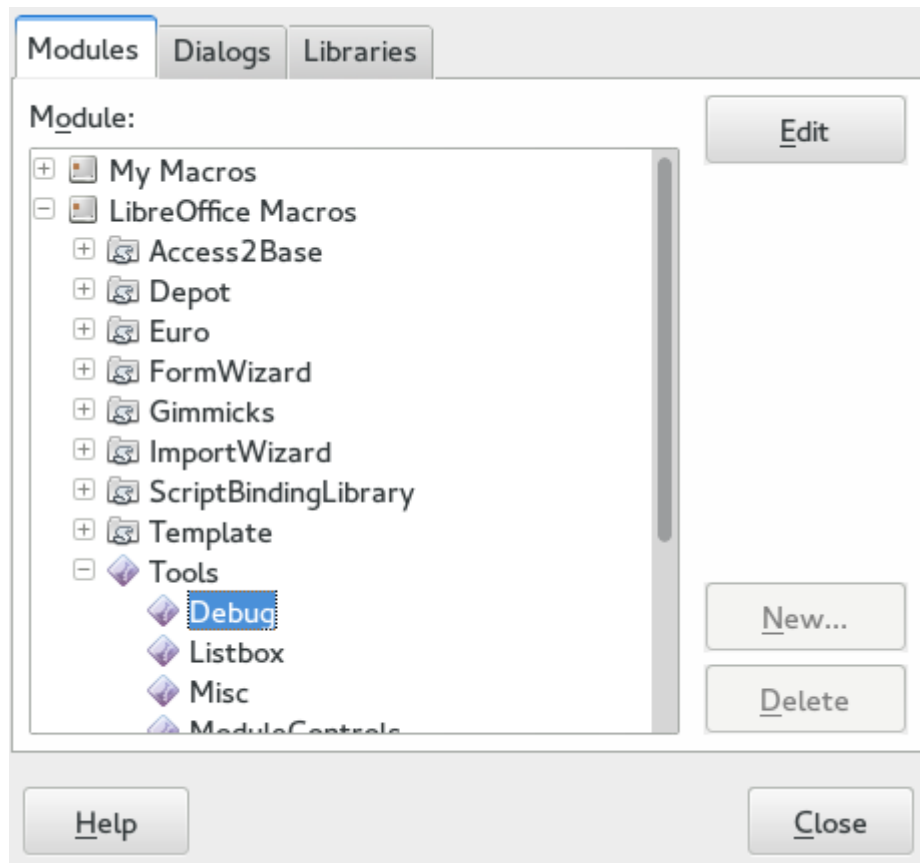


Figure 3. Modules tab of the Basic Macro Organizer dialog.

Use the Libraries tab (see Figure 4) to create, delete, rename, import, and export libraries.

TIP

When I rename the library contained in this document (for example, from OOME_30 to OOME_40), which I do for every major release, this breaks every button that calls a macro in the library. This also breaks every macro contained in the library that explicitly references the library. I fix this by unzipping the ODT file, changing all references in a text editor, then zipping the file again.

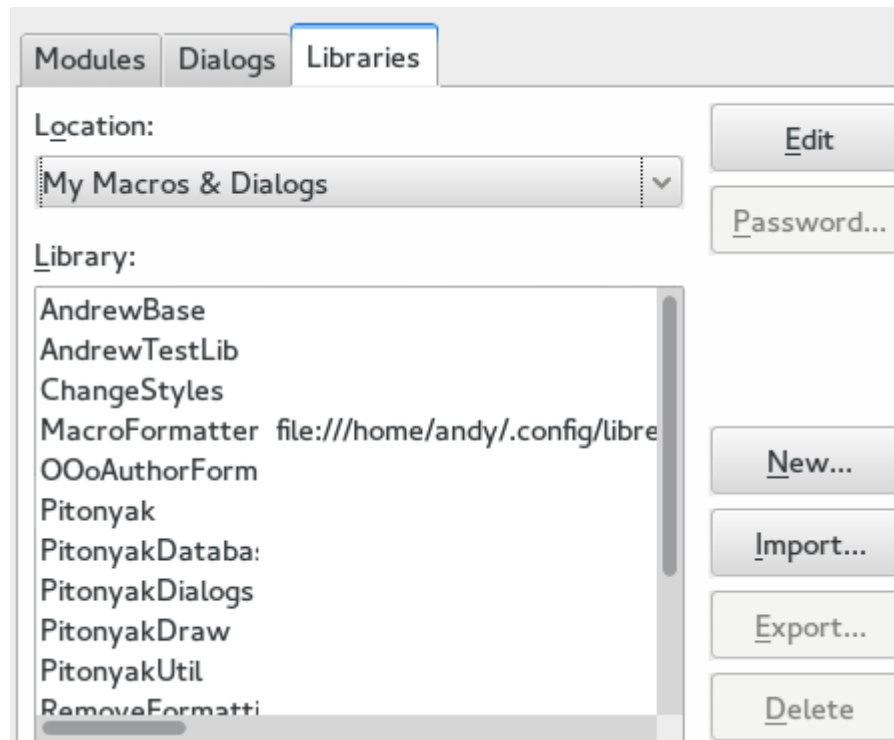


Figure 4. Libraries tab of the OOo Macro Organizer dialog.

The first step is to select the desired library container from the Location drop-down. To rename a library, double click on the library and then edit the name in place.

TIP

Renaming modules and libraries in the Basic Macro Organizer can be frustrating. To rename a library, double or triple click on the library name then wait a few seconds. Try again. Try one more time. Click on another library. Double or triple click on the library name; you get the idea.

The easiest method to change a module name is to right click on the module name in the tabs at the bottom of the IDE and choose rename (see Figure 11).

2.3. Macro language

The macro language is based on the BASIC programming language. The standard macro language is officially titled StarBasic, but, it is also referred to as Basic. Many different programming languages can be used for automation. There is easy support for macros in Basic, JavaScript, Python, and BeanShell. In this document, my primary concern is Basic.

2.4. Create a module in a document

Each document is a library container able to contain macros and dialogs. When a document contains the macros that it uses, possession of the document implies possession of the macros. This is a convenient distribution and storage method. Send the document to another person or location, and the macros are available and usable.

- 1) To add a macro to a document, the document must be open for editing. Start by opening a new text document, which will be named “Untitled 1” — assuming that no other untitled document is open.
- 2) Use **Tools > Macros > Organize Macros > LibreOffice Basic** to open the Basic Macros dialog (see Figure 1).
- 3) Click the Organizer button to open the OOO Macro Organizer dialog, then click on the Libraries tab (see Figure 4).
- 4) Select “Untitled 1” from the location drop-down.

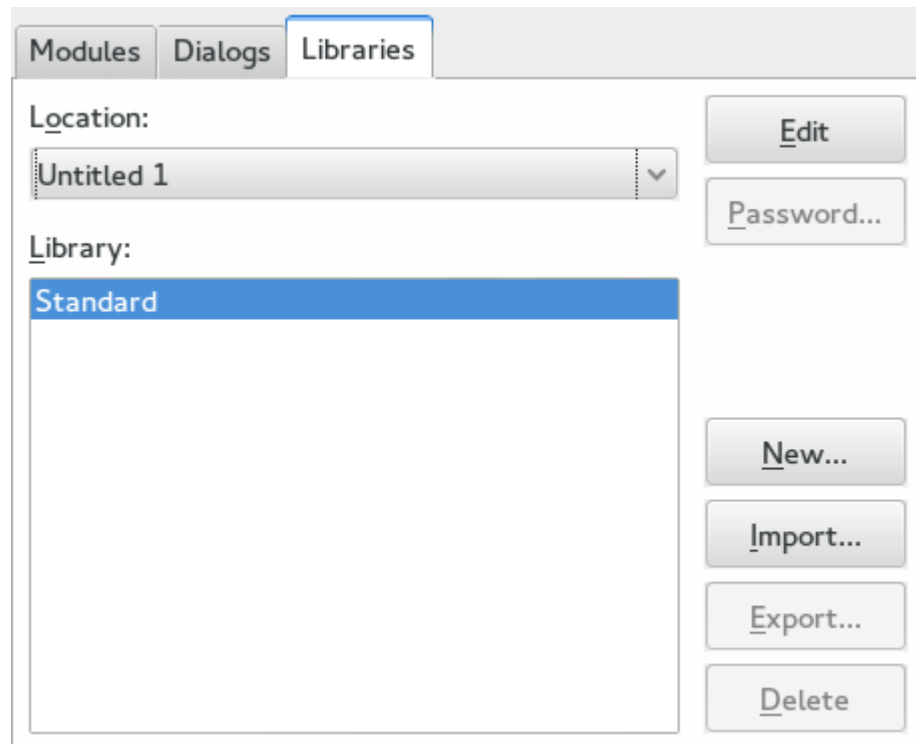


Figure 5. Libraries tab of the Basic Macro Organizer dialog.

- 5) Click New to open the New Library dialog.

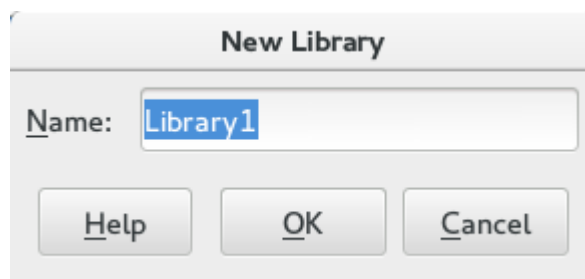


Figure 6. New library dialog.

- 6) The default name is Library1, which is not very descriptive. Choose a descriptive name and click OK. The new library is shown in the list. For this example, I named the library “HelloWorld”.

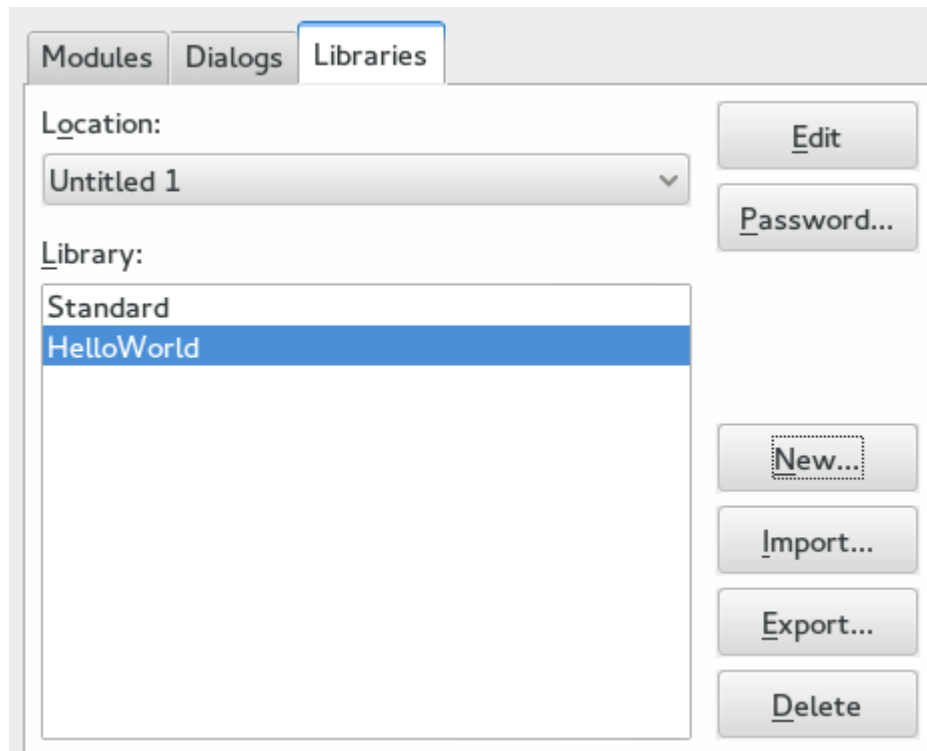


Figure 7. The new library is shown in the list.

- 7) In the Modules tab, select the HelloWorld library. LO created the module named “Module1” when the library was created.

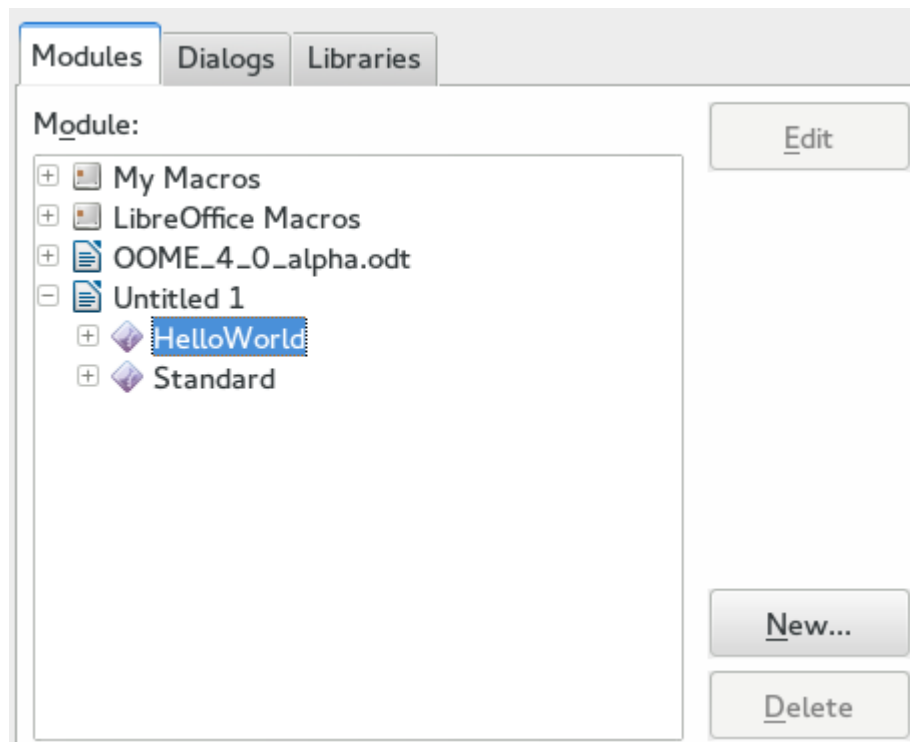


Figure 8. The new library is shown in the list.

- 8) Click New to open the New Module dialog. The default name is Module2, because Module1 already exists.



Figure 9. New module dialog.

- 9) Use a descriptive name and click OK.

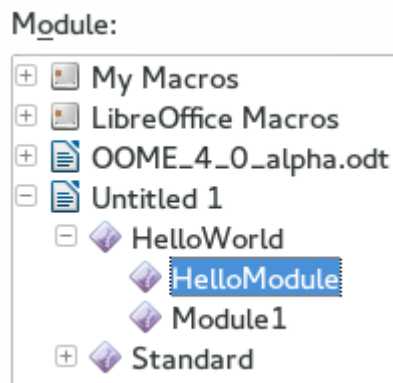


Figure 10. The new module is shown in the list.

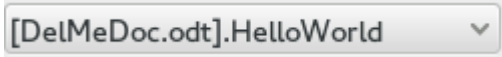
- 10) Select HelloWorld and click Edit.

- 11) At this point, I saved the document and named it “DelMeDoc” because I intended to delete the document when I was finished with the example. Select a name that works well for you. If you reopen the dialog shown in Figure 10, the document name will be shown rather than “Untitled 1”.

At this point, the Integrated Debugging Environment (IDE) is opened to edit the macro.

2.5. Integrated Debugging Environment

Use the Basic Integrated Debugging Environment (IDE) to create and run macros (see Figure 11). The IDE places significant functionality in a little space. The toolbar icons are described in Table

1. The top left corner just above the editing window  contains a drop-down list that shows the current library. The portion in the square brackets identifies the library container, and the portion following identifies the library. This provides a quick method to choose a library. You can also quickly choose a library container, library, or module using the Object Catalog on the left side of the display.

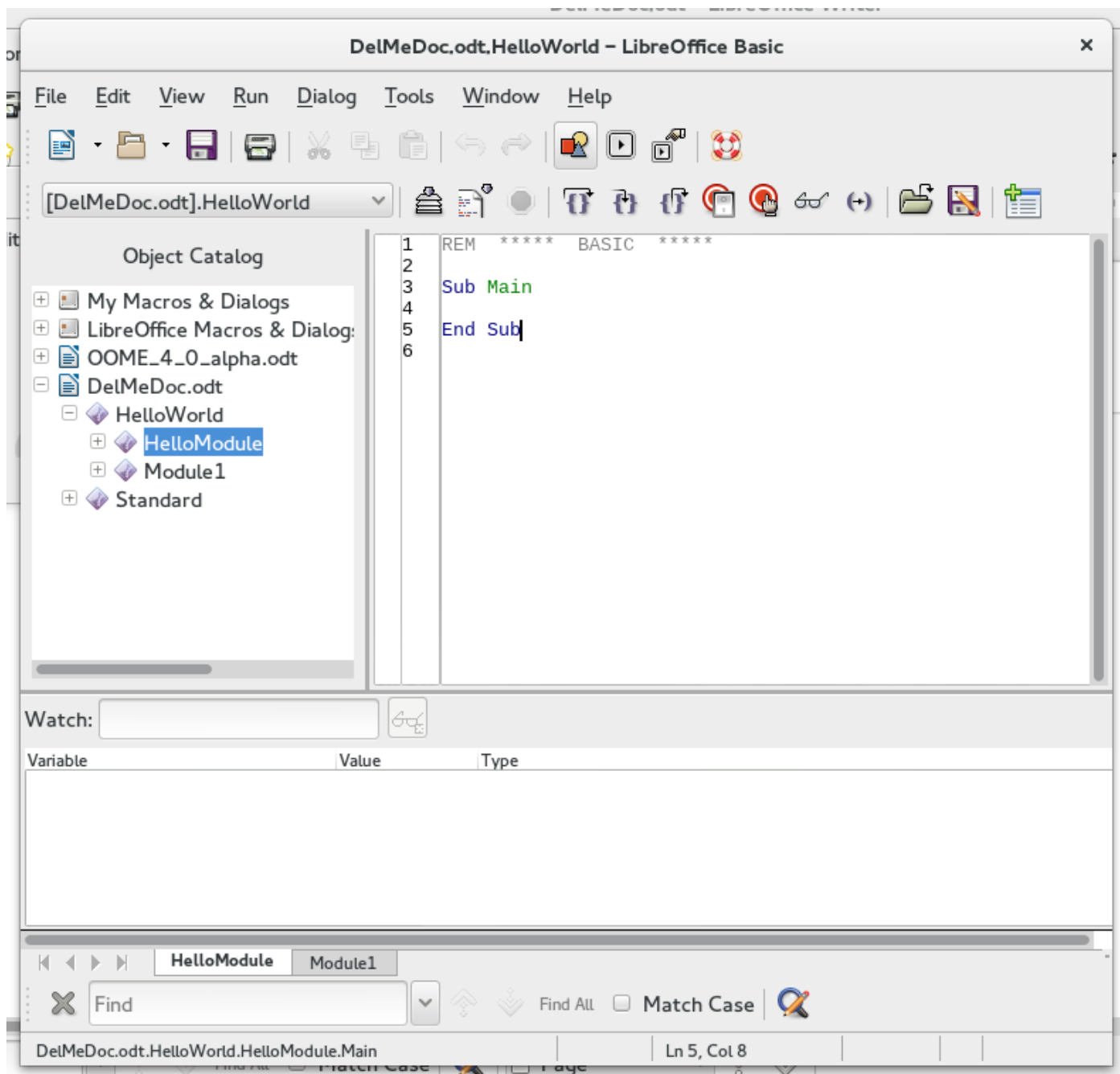


Figure 11. Basic Integrated Debugging Environment.

Rest your mouse cursor on a toolbar icon for a few seconds to read the text that appears; this provides a hint at what that icon does.

Table 1. Toolbar icons in the Basic IDE.

Icon	Key	Description
	Ctrl+N	Create a new document.
	Ctrl+O	Open an existing document.

Icon	Key	Description
	Ctrl+S	Save the current library. If the library is in a document, then the document is saved.
	Ctrl+P	Print the macro to the printer.
	Ctrl+X	Cut to clipboard.
	Ctrl+C	Copy to clipboard.
	Ctrl+V	Paste the clipboard.
	Ctrl+Z	Undo the last operation.
	Ctrl+Y	Redo the last operation that was undone.
		Open the Object catalog (see Figure 12). Select the macro and double-click on the macro.
		Open the Basic Macros dialog (see Figure 2). Select a macro and click edit or run. This is a short-cut for Tools > Organize Macros > LibreOffice Basic .
		Select a module. This opens the OOO Macro Organizer dialog with the Modules tab selected (see Figure 3). Select a module and click Edit.
		Open the help, which contains numerous useful examples for Basic.
		Click the Compile icon to check the macro for syntax errors. No message is displayed unless an error is found. The Compile icon compiles only the current module.
	F5	Run the first macro in the current module. While stopped (from a breakpoint, or single stepping, this continues the execution). To run a specific macro, use  to open the Basic Macros dialog, select the desired macro, and click Run; or select the macro in the Object Catalog on the left.
	Shift+F5	Stop the currently running macro.
	Shift+F8	Step over the current statement. When a macro stops at a breakpoint, this executes the current statement. Can also be used to start a macro running in single step mode.
	F8	Step into. This is the same as Step over except if the current statement calls another macro, it single steps into that macro so that you can watch that macro execute.
	Ctrl+Shift+F8	Step out runs the macro to the end of the current subroutine or function.
	Shift+F9	Toggles a breakpoint on / off at the current cursor position in the IDE. An icon () is displayed to the left of the line to show that a breakpoint is set for that line. You can also double-click in the breakpoint area to toggle a breakpoint on / off.
		Open the manage breakpoints dialog (see Figure 17), which allows you to selectively turn breakpoints on or off, and also, to prevent a breakpoint from triggering until it has been reached a certain number of times.
	F7	Select a variable and click the Watch icon to add the variable to the watch window. You can also enter the variable name into the Watch input line and press enter.
		Find parentheses.
		Insert Basic source. Open a dialog to select a previously saved basic file and insert it.

Icon	Key	Description
		Save the current module as a text file on disk. LO stores modules on disk in a special encoded format. Files saved using this method are standard text files. This is an excellent way to create a backup of a macro or to create a text file that can be easily sent to another person.
		Import a dialog from another module.

Module names are listed along the bottom of the IDE. Module navigation buttons  are to the left of the module names. The buttons display the first, previous, next, and last module in the current library. Right click on a module name to:

- Insert a new module or dialog.
- Delete a module or dialog.
- Rename a module; this is the easiest way that I know to rename a module or dialog.
- Hide a module or dialog.
- Open the Basic Organization dialog.

Use  to open the Objects catalog (see Figure 12), select a macro and double-click on the macro to edit that macro.

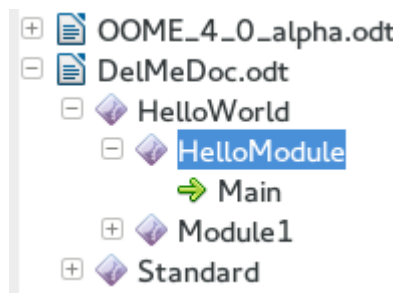


Figure 12. Objects catalog.

2.6. Enter the macro

Change the text in the IDE to read as shown in Listing 1. Click the run icon.

Listing 1. Hello World macro

```
REM ***** BASIC *****
Option Explicit

Sub Main
    Print "Hello World"
End Sub
```

Table 2. Line by line description of Listing 1.

Line	Description
REM ***** BASIC *****	Basic comment, this line is ignored. A comment may also begin with a single quote character.
Option Explicit	Tells the basic interpreter that it is an error to use a variable that is not explicitly defined. Misspelled variables are likely to be caught as an error at compile time.
Sub Main	Indicates that this is the beginning of the definition of a subroutine named Main.
Print "Hello World"	The Print command.
End Sub	End the Main subroutine.

The macro in Listing 1 is text that a human can read. The computer must translate macro text into something that the computer can use. The process of converting the human readable macro into something that the computer can read is called compiling. In other words, a macro is written as lines of text that are compiled to prepare a runnable program.

2.7. Run a macro

The Run icon always runs the first macro in the current module. As a result, a different approach is required if more than one macro is in a module. The following options can be used:

- Place the cursor in the macro to run and click the Run icon.
- Use the first macro to call the desired macro. I like this method during development. I keep a main macro as the first thing that does nothing. During development, I change the main macro to call the macro of interest. For general use, I let the main macro at the top call the most frequently run macro.
- Use the Macro dialog (see Figure 2) to run any routine in the module.
- Add a button to your a document or toolbar that runs the macro.
- Assign the macro to a keystroke. Use **Tools > Customize** to open the Customize dialog. Select the Keyboard tab. Macro libraries are at the bottom of the Category list. This is also available with **Tools > Macros > Organize Macros > LibreOffice Basic**, select the macro, and click the **Assign** button to launch the Customize dialog. Various tabs in this dialog allow you to assign the macro to execute as a menu item, from a toolbar icon, or a system event.

To use the Macro dialog to run any subroutine in a module, follow these steps:

1. Select **Tools > Macros > Organize Macros > LibreOffice Basic** to open the Macro dialog (see Figure 2).
2. Find the document that contains the module in the “Macro from” list.
3. Double-click a library to toggle the display of the contained modules.
4. Select the module to display the contained subroutines and functions in the “Existing macros in: <selected module name>” list.
5. Select the desired subroutine or function to run — for example, HelloWorld.
6. Click the Run button to run the subroutine or function.

2.8. Macro security

Depending on how OOO is configured, you may not be able to run macros in a document. When I open a document containing a macro, the warning in Figure 13 is displayed. If you do not expect a macro or do not trust the source, then choose to disable macros. Remember, I am able to write a macro that destroys your computer.



Figure 13. The opened document contains a macro.

Use **Tools > Options > LibreOffice > Security** to open the Security tab of the Options dialog.



Figure 14. Options dialog, security tab.

Click the Macro Security button to open the Macro Security dialog. Choose a level that fits your comfort level. Medium security uses the confirmation dialog in Figure 13, which is relatively quick and unobtrusive.



Figure 15. Macro Security dialog, Security Level tab.

You can set trusted locations and certificates that allow documents to load without confirmation based on either where they are stored, or the certificate used to sign a document.

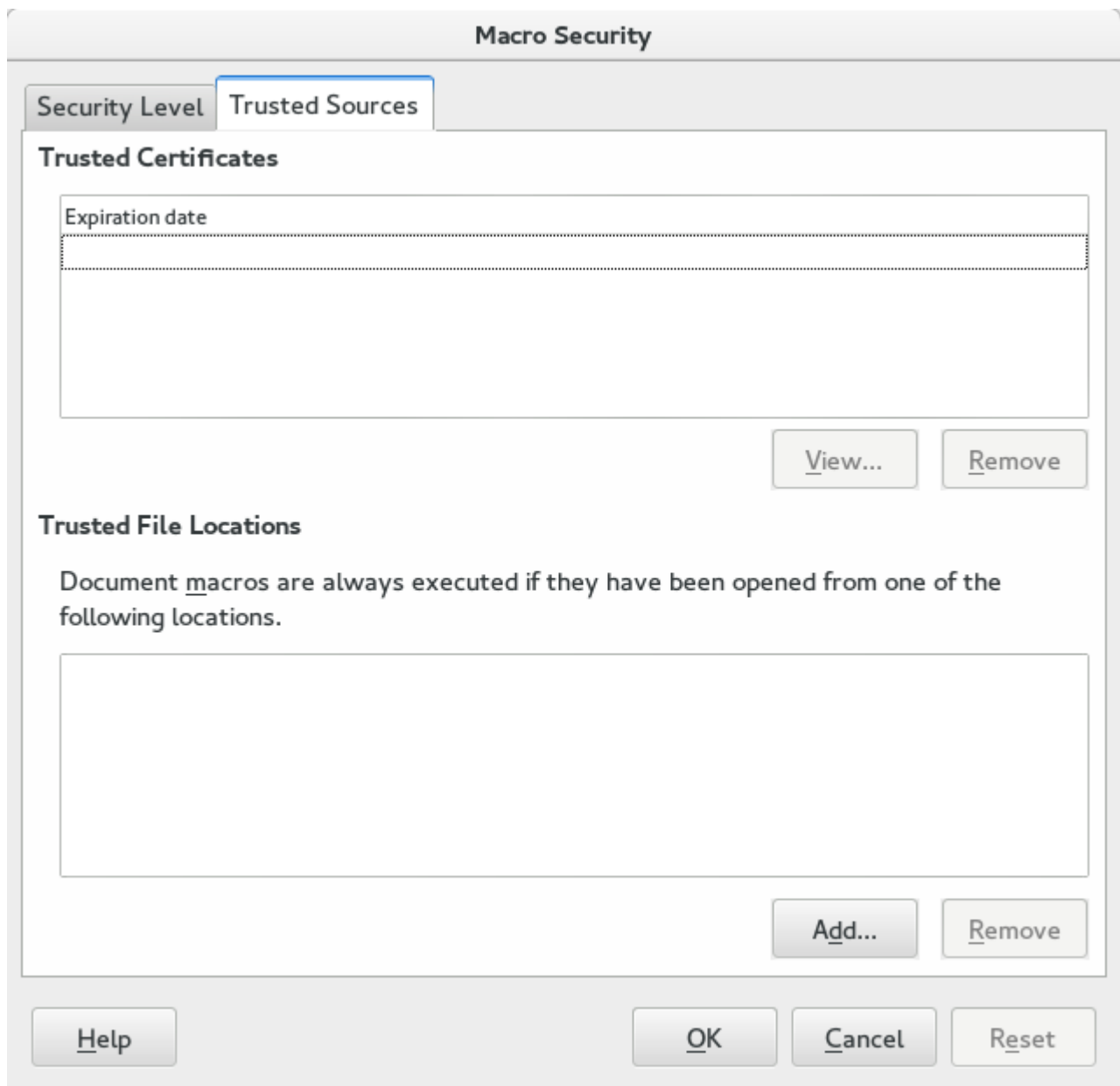


Figure 16. Macro Security dialog, Trusted Sources tab.

2.9. Using breakpoints

If you set a breakpoint in the code, the macro stops running at that point. You can then inspect variables, continue running the macro, or single-step the macro. If a macro fails and you don't know why, single-stepping (running one statement at a time) allows you to watch a macro in action. When the macro fails, you'll know how it got there. If a large number of statements run before the problem occurs, it may not be feasible to run one statement at a time, so you can set a breakpoint at or near the line that causes the problem. The program stops running at that point, and you can single-step the macro and watch the behavior.

The breakpoint On/Off icon sets a breakpoint at the statement containing the cursor. A red stop sign marks the line in the breakpoint column. Double-click in the breakpoint column to toggle a breakpoint at that statement. Right-click a breakpoint in the Break-point column to activate or deactivate it.

Use the Manage Breakpoints icon to load the Manage Breakpoints dialog. All of the active breakpoints in the current module are listed by line number. To add a breakpoint, enter a line number in the entry field and click New. To delete a breakpoint, select a breakpoint in the list and click the Delete button. Clear the Active check box to disable the highlighted breakpoint without deleting it. The Pass Count input box indicates the number of times a breakpoint must be reached before it is considered active. If the pass count is four (4), then the fourth time that the statement containing the breakpoint is to be run, it will stop rather than run. This is extremely useful when a portion of the macro does not fail until it has been called multiple times.

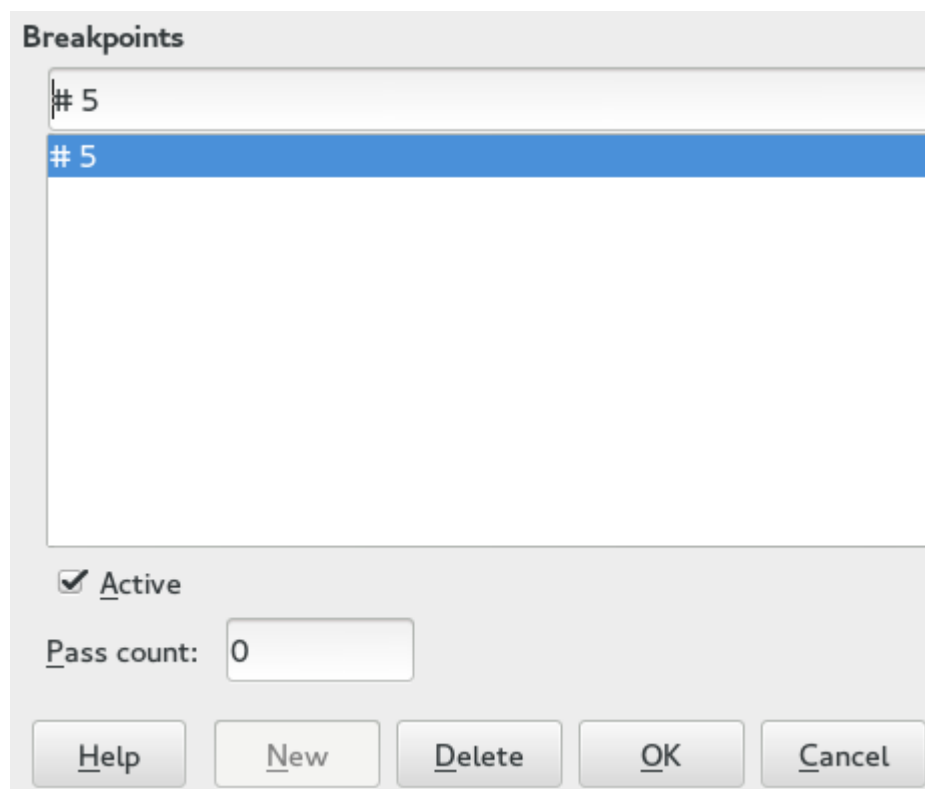


Figure 17. Manage Breakpoints dialog.

There are two things that cause a breakpoint to be ignored: a pass count that is not zero, and explicitly marking the breakpoint as “not active” in the Manage breakpoints dialog. Every breakpoint has a pass count that is decremented toward zero when it is reached. If the result of decrementing is zero, the breakpoint becomes active and stays active because the pass count stays at zero thereafter. The pass count is not restored to its original value when the macro is finished or restarted.

2.10. How libraries are stored

Macro libraries are stored as XML files that are easily editable using any text editor. In other words, it is easy for you to poke around and damage your files. This is advanced material that you may want to ignore. If you do not understand XML and why the file contains > rather than >, perhaps you should not edit the files. Although manually editing your external libraries is generally considered foolish, I have had at least one instance where this was required, because a version of OOo was unable to load a module that contained a syntax error.

Each library is stored in a single directory, and each module and dialog is contained in a single file. The global libraries that are included with OpenOffice.org are stored in a shared basic directory under the directory in which OOo is installed. Examples:

```
C:\Program Files\OpenOffice3.2\share\basic      'A Windows installation
/opt/openoffice.org/basis3.2/share/basic        'A Linux installation
```

OpenOffice.org stores user-specific data in a directory under the user's home directory. The location is operating system specific. Use **Tools > Options > OpenOffice.org > Paths** to view where other configuration data is stored. Here are some examples where my basic macros are stored:

```
C:\Documents and Settings\andy\Application Data\OpenOffice.org\3\user\basic      'Windows XP
C:\Users\pitonyaka\AppData\Roaming\OpenOffice.org\3\user\basic                    'Windows 7
/opt/libreoffice6.0/share/basic                                                    'Linux
/home/andy/.config/libreoffice/4/user/basic                                       'Linux
```

Each library is stored in its own directory off the basic directory. The user directory contains two files and one directory for each library (see Table 3).

Table 3. Files and some directories in my user/basic directory.

Entry	Description
dialog.xlc	XML file that references every dialog file known to this user.
script.xlc	XML file that references every library file known to this user.
Standard	Directory containing the Standard library.
Pitonyak	Directory containing a the library named Pitonyak.
PitonyakDialogs	Directory containing the library named PitonyakDialogs.

The files dialog.xlc and script.xlc contain a reference to all of the dialogs and libraries known to LO. If these two files are overwritten, LO will not know about your personal libraries even if they exist. You can, however, either manually edit the files, or, even easier, use the Basic Macro Organizer dialog to import the libraries (because you can import a library based on the directory).

The directory containing a library contains a single file for each module and dialog in the library. The directory also contains the files dialog.xlb and script.xlb, which references the modules.

2.11. How documents are stored

The standard OOo formats use a standard ZIP file to store all of the components. Any program that can view and extract ZIP files can be used to inspect a document — however, some programs will require you to change the file extension to end with ZIP.

After unzipping an OOo document, you will find files that contain the primary content, styles, and settings. The extracted document also contains three directories. The META-INF directory references all of the other files, embedded pictures, code libraries, and dialogs. The Dialogs directory contains all of the embedded dialogs, and the Basic directory contains all of the embedded libraries.

The point to all this is that in an emergency, you can manually inspect a document's XML and potentially fix problems. You can also mess up the file so that it is no longer usable, so, use a copy not the original.

2.12. Conclusion

Macros and dialogs are stored in modules, modules are stored in libraries, and libraries are stored in library containers. The application is a library container, as is every document. The IDE is used to create and debug macros and dialogs.

You have just completed one of the most difficult steps in writing macros for OpenOffice.org: writing your first macro! You are now ready to explore, try other macro examples, and create a few of your own.

3. Language Constructs

The macro language is similar to the one in Microsoft Office because they are both based on BASIC. Both macro languages access the underlying implementation structures, which differ significantly and are therefore incompatible. This chapter emphasizes the portions of the language that do not access the underlying implementation.

This chapter shows how to assemble different components to produce a macro that will compile and run. In a word: syntax. Correct syntax does not imply that the macro does what you want, only that the pieces are put together in a correct way. The phrases “Can I drive?” and “May I drive?” are both syntactically correct. The first phrase is about ability, and the second phrase is about permission. In speech, these two questions may be understood to have the same meaning. The computer, on the other hand, does exactly what you ask, rather than what you mean.

The primary components that syntactically constitute an OpenOffice.org macro are statements, variables, subroutines, and functions, and flow-control constructs. A statement is a small, runnable portion of code that is usually written as a single line of text. Variables are containers that hold values that can change during macro execution. Subroutines and functions separate a macro into functional named groups of statements, allowing for better organization. Flow control directs which statements run and how many times.

OOo runs one line of a macro at a time. Each line of a macro is delimited exactly the way it sounds; by a new line (see Listing 2).

Listing 2. Two line macro

```
Print "This is line one"
Print "This is line two"
```

Lines that are too long may use more than one line by appending an underscore (_) to the end of the line (see Listing 3). The underscore must be the last character on the line for it to act as a line-continuation character. The underscore has no special meaning when it isn't the last character of a line, allowing it to be used inside strings and in variable names. When used as a continuation character, spaces may precede the underscore and are in some cases required to separate the underscore from what precedes it. For example, splitting the line “a+b+c” after the b requires a space between the b and the underscore, or the underscore is considered part of the variable name. Spaces that inadvertently follow a continuation character may cause a compile-time error. Unfortunately, the error does not state that something follows the underscore, but that the next line is invalid.

Listing 3. Append an underscore to the end of a line to continue on the next line.

```
Print "Strings are concatenated together " & _
    "with the ampersand character"
```

TIP

When anything follows a line-continuation character, the next line is not taken as its continuation. Sometimes, when I copy code listings from web sites and paste them into the IDE, an extra space is added at the end of a line, which breaks line continuation.

You can place multiple statements on a single line by separating them with colons. This is usually done for improved code readability. Each of the combined statements act as a single line of code while debugging the macro in the Integrated Development Environment (IDE). Listing 4, therefore, acts like three separate statements while using the IDE to single-step through the macro.

Listing 4. *Set the variables x, y, and z to zero.*

```
x = 0 : y = 0 : z = 0
```

You should add remarks, also called comments, to all of the macros that you write. While writing a macro, remember that what is clear today may not be clear tomorrow, as time passes and new projects arise and memory fades all too quickly. You start a comment with either the single quotation character, or the keyword REM. All text on the same line following a comment indicator is ignored. Comments are not considered runnable statements; they are ignored while single-stepping a macro.

Listing 5. *Add comments to all of the macros that you write.*

```
REM Comments may start with the keyword REM.  
ReM It is not case-sensitive so this is also a comment.  
' All text following the start of the comment is ignored  
X = 0 ' A comment may also start with a  
    ' single quote  
z = 0 REM All text following the start of the comment is ignored
```

TIP

Keywords, variables, and routine names in Basic are not case-sensitive. Therefore, REM, Rem, and rEm all start a comment.

Nothing can follow a line-continuation character, and this includes comments. All text following a comment indicator is ignored — even the continuation character. The logical result from these two rules is that a line-continuation character can never occur on the same line as a comment.

3.1. Compatibility with Visual Basic

With respect to syntax and BASIC functionality, Basic is very similar to Visual Basic. The two Basic dialects are nothing alike when it comes to manipulating documents, but the general command set is very similar. Steps were taken to improve the general compatibility between the two dialects.

The compiler option “Option Compatible” causes the module with the option to be more compatible with Visual Basic. This option affects only the module in which it is contained. Because a macro calls different modules during its execution, both the old and new behavior may be used, depending upon the existence of “Option Compatible” in each called module. Setting the option in one module and then calling another module has no effect in the called module.

A run-time function, CompatibilityMode(True/False), allows the behavior of run-time functions to be modified during the execution of a macro. This provides the flexibility to enable the new run-time behavior, perform some operations, and then disable the new run-time behavior.

CompatibilityMode(False) overrides Option Compatible for the new runtime behavior. Hopefully, some method of probing the current mode will be provided.

Visual basic allows any Latin-1 (ISO 8859-1) character as a valid variable name, LO does not. Setting “Option Compatible” allows “ä” to be considered a valid variable name. This is just one of many changes that use “Option Compatible.” The CompatibilityMode() function neither enables nor disables the new extended identifier names because CompatibilityMode() is not called until run time and variable names are recognized at compile time.

Both Visual Basic and LO Basic support the rmdir() command to remove a directory. VBA can remove only empty directories, but Basic can recursively remove an entire directory tree. If CompatibilityMode(True) is called prior to calling rmdir(), Basic will act like VBA and generate an error if the specified directory is not empty. This is just one of many changes that use CompatibilityMode().

StarBasic is much more forgiving than VBA. It is easier, therefore, to convert simple macros from VBA to LO Basic. A few examples, in LO Basic, “set” is optional during assignment. Therefore, “set x = 5” works in both VBA and Basic, but “x = 5” fails in VBA and works with Basic.

Another example is that array methods are far more stable and forgiving in OOo than in VBA; for example, the functions to determine array bounds (LBound and UBound) work fine with empty arrays, whereas VBA crashes. When I need to write a VBA macro, therefore, I spent a lot of time working around bugs that do not exist in VBA.

3.2. Compiler options and directives

A compiler converts a macro into something that the computer is able to run. Compiler behavior can be controlled through commands such as “Option Explicit” at the top of a module before all variables, subroutines, and functions. A compiler option controls the compilers behavior for the module containing the option.

Table 4. Compiler options and directives.

Option	Description
Def	Give a default type to undeclared variables based on the variables name.
Option Base	Control whether the first array index is 0 or 1; assuming it is not specified.
Option Compatible	Cause Star Basic to act more like VB.
Option Explicit	Force all variables to be defined. While the macro runs, if a variable that has not yet been defined is used, an error occurs.

3.3. Variables

Variables are containers that hold values. OpenOffice.org supports different types of variables designed to hold different types of values. This section shows how to create, name, and use

variables. Although LO Basic does not force you to declare variables, you should declare every variable that you use. The reasons for this are developed throughout this section.

3.3.1. Constant, subroutine, function, label, and variable names

Always choose meaningful names for your variables. Although the variable name “var1” requires little thought during creation, “FirstName” is more meaningful. Some variable names are not particularly descriptive but are commonly used by programmers anyway. For example, “i” is commonly used as a shortened version of “index,” for a variable that is used to count the number of times a repetitive task is executed in a loop. Basic imposes restrictions on variable names, including the following:

- *Officially*, a variable name cannot exceed 255 characters in length. I tested names with more than 300 characters with no problems, but I don’t recommend this!
- The first character of a variable name must be a letter: A-Z or a-z. If `Option Compatible` is used, then all characters defined as letters in the Latin-1 (ISO 8859-1) character set are accepted as part of an identifier name.
- The numbers 0-9 and the underscore character (`_`) may be used in a variable name, but not as the first character. If a variable name ends with an underscore, it won’t be mistaken for a line-continuation character.
- Variable names are not case sensitive, so “FirstName” and “firstNAME” both refer to the same variable.
- Variable names may contain spaces but must be enclosed in square brackets if they do. For example, `[First Name]`. Although this is allowed, I considered it poor programming practice. Using square brackets also allows a variable name to start with a number or to contain Latin-1 characters.

TIP These restrictions also apply to constant, subroutine, function, and label names.

3.3.2. Declaring variables

Some programming languages require that you explicitly list all variables used. This process is called “declaring variables.” Basic does not require this. You are free to use variables without declaring them.

Although it is convenient to use variables without declaring them, it is error-prone. If you mistype a variable name, it becomes a new variable rather than raising an error. Place the keywords “Option Explicit” before any runnable code in every module to cause Basic to treat undeclared variables as run-time errors. Comments may precede Option Explicit because they are not runnable. Although it would be even better if this became a compile-time error, Basic does not resolve all variables and routines until run time.

Listing 6. *Use Option Explicit before the first runnable line in a module.*

```
REM ***** BASIC *****  
Option Explicit
```

TIP

Use “Option Explicit” at the top of every module that you write; it will save you a lot of time searching for errors in your code. When I am asked to debug a macro, the first thing I do is add “Option Explicit” at the top of each module.

The scope for options is the module which contains the option. In other words, setting option explicit in one module, then calling another module, will not cause undeclared variables in the called module to cause a runtime error; unless the called module also has “option explicit”.

You can declare a variable with or without a type. A variable without an explicit type becomes a Variant, which is able to take on any type. This means that you can use a Variant to hold a numeric value and then, in the next line of code, overwrite the number with text. Table 5 shows the variable types supported by Basic, the value that each type of variable contains immediately after declaration (“initial value”), and the number of bytes that each type uses. In Basic, a variable's type can be declared by appending a special character to the end of the name when it is declared. The Post column in Table 5 contains the supported characters that can be post-fixed to a variable's name when the variable is declared.

Table 5. *Supported variable types and their initial values.*

Type	Post	Initial	Bytes	Convert	Description
Boolean		False	1	CBool	True or False
Currency	@	0.0000	8	CCur	Currency with 4 decimal places
Date		00:00:00	8	CDate	Dates and times
Double	#	0.0	8	Cdbl	Decimal numbers in the range of $\pm 1.79769313486232 \times 10^{308}$
Integer	%	0	2	CInt	Integer from -32,768 through 32,767
Long	&	0	4	CLng	Integer from -2147483648 through 2147483647
Object		Null	varies		Object
Single	!	0.0	4	CSng	Decimal numbers in the range of $\pm 3.402823 \times 10^{38}$
String	\$	""	varies	CStr	Text less than 2GB characters
Variant		Empty	varies	CVar	May contain any data type

TIP

Internally, a string uses a 32 bit integer to hold the length. I expect, therefore, that a string can contain 2,147,483,647 (2GB - 1) characters.

Although Basic supports a Byte variable type, you can't directly declare and use one. The function CByte, as discussed later, returns a Byte value that may be assigned to a variable of type Variant. With OOO version 2.0, you can declare a variable of type Byte, but the variable is assumed to be an externally defined object of type Byte rather than an internally defined Byte variable.

Use the DIM keyword to explicitly declare a variable before use (see Table 6). You can declare multiple variables on a single line, and you can give each variable a type when it is declared. Variables with no declared type default to type Variant.

Table 6. *Declaring simple variables.*

Declaration	Description
Dim Name	Name is type Variant because no type is stated.
Dim Name As String	Name is type String because the type is explicitly stated.
Dim Name\$	Name\$ is type String because Name\$ ends with a \$.
Dim Name As String, Weight As Single	Name is type String and Weight is type Single.
Dim Width, Length	Width and Length are type Variant.
Dim Weight, Height As Single	Weight is type Variant and Height is type Single.

TIP When multiple variables are declared in a single line, the type for each variable must be listed separately. In the last line of Table 5, Weight is a Variant, even though it looks like it may be of type Single.

Much of the available literature on macro programming uses a variable naming scheme based on Hungarian notation. With Hungarian notation, you can determine a variable's type from its name. In practice, everyone does this differently and with differing levels of adherence. This is a stylistic decision that some people love and some people hate.

Basic uses Def<type> statements to facilitate the use of Hungarian notion. The Def statements, which are local to each module that uses them, provide a default type for an undeclared variable based on its name. Normally, all undeclared variables are of type Variant.

The Def statement is followed by a comma-separated list of character ranges that specify the starting characters (see Listing 7).

Listing 7. *Declare untyped variables starting with i, j, k, or n to be of type Integer.*

```
DefInt i-k,n
```

Table 7 contains an example of each supported Def statement. Def statements, like Option statements, are placed in the module before any runnable line or variable declaration. The Def statement does not force a variable with a specific first letter to be of a certain type, but rather provides a default type other than Variant for variables that are used but not declared. I have never seen the Def statement used and I recommend that you do not use the Def statement.

TIP If you use “Option Explicit,” and you should, you must declare all variables. This renders the Def<type> statements useless because they affect only undeclared variables.

Table 7. Examples of supported Def statements in OpenOffice.org.

Def Statement	Type
DefBool b	Boolean
DefDate t	Date
DefDbl d	Double
DefInt i	Integer
DefLng l	Long
DefObj o	Object
DefVar v	Variant

3.3.3. Assigning values to variables

The purpose of a variable is to hold values. To assign a value to a variable, type the name of the variable, optional spaces, an equals sign, optional spaces, and the value to assign to the variable, like so:

```
x = 3.141592654
y = 6
```

The optional keyword Let may precede the variable name but serves no purpose other than readability. A similar optional keyword, Set, meant for Object variables, serves no purpose other than readability. These keywords are rarely used.

3.3.4. Boolean variables are True or False

Boolean variables have two valid values: True or False. They are internally represented by the Integer values -1 and 0, respectively. Any numeric value assigned to a Boolean that does not precisely evaluate to 0 is converted to True. The macro in Listing 8 introduces a few new concepts. A string variable, s, accumulates the results of the calculations, which are displayed in a dialog (see Figure 18). Adding CHR\$(10) to the string causes a new line to be printed in the dialog. Unfortunately, accumulating the results in a string provides a more complicated macro than using simple statements such as "Print CBool(5=3)", but the results are easier to understand (see Figure 18). In the Write version of the document, a button is frequently available that allows you to run the macro immediately.

Listing 8. Demonstrate conversion to type boolean.

```
Sub ExampleBooleanType
    Dim b as Boolean
    Dim s as String
    b = True

    b = False
    b = (5 = 3) REM Set to False
    s = "(5 = 3) => " & b
    b = (5 < 7) REM Set to True
    s = s & CHR$(10) & "(5 < 7) => " & b
```



```

b = 7          REM Set to True because 7 is not 0
s = s & CHR$(10) & "(7) => " & b
MsgBox s
End Sub

```



Figure 18. The dialog displayed by Listing 8.

The internal binary representation of True as -1 has all of the bits set to 1. The internal binary representation of False as 0 has all of the bits set to 0.

3.3.5. Numeric variables

Numeric variables contain numbers. Basic supports integers, floating-point, and currency numbers. Integers may be expressed as hexadecimal (base 16), octal (base 8), or the default decimal numbers (base 10).

A discussion of other number bases is important because internally, computers represent their data in binary format. It is easy to convert between the binary, hexadecimal, and octal number bases; and for humans it may be easier to visualize binary numbers when represented in other number bases.

Decimal numbers, base 10, are composed of the 10 digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Add 1 to 9 in decimal and the result is 10. Binary (base 2), octal (base 8), and hexadecimal (base 16) numbers are also commonly used in computing. Octal numbers are composed of the numbers 0, 1, 2, 3, 4, 5, 6, and 7. In octal, add 1 to 7 and the result is 10 (base 8). Hexadecimal numbers are composed of the 16 digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Binary numbers are composed of the two digits 0 and 1. Table 8 contains the numbers from 0 through 18 in decimal, and their corresponding values in binary, octal, and hexadecimal bases.

Table 8. Numbers in different bases.

Decimal	Binary	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5

Decimal	Binary	Octal	Hexadecimal
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12

Integers are assumed to be expressed as decimal numbers. Commas are not allowed. Hexadecimal numbers are preceded by “&H” and octal numbers are preceded by “&O” (letter O, not a zero). Unfortunately, there is no easy method to enter binary numbers. Table 9 presents a few simple guidelines for entering numbers.

Table 9. *A few guidelines for entering numbers in Basic.*

Example	Description
Use 1000 not 1,000	Write numbers without a thousands separator; do not use commas.
+ 1000	A space is permitted between a leading plus or minus sign and the number.
&HFE is the same as 254	Precede a hexadecimal number with &H.
&O11 is the same as 9	Precede an octal number with &O.
Use 3.1415 not 3,1415	Do not use commas for the decimal.
6.022E23	In scientific notation, the “e” can be uppercase or lowercase.
Use 6.6e-34 not 6.6e -34	Spaces are not allowed in a number. With the space, this evaluates as 6.6 - 34 = -27.4.
6.022e+23	The exponent may contain a leading plus or minus sign.
1.1e2.2 evaluates as 1.1e2	The exponent must be an integer. The fractional portion causes an error.

In general, assigning a String to a numeric variable sets the variable to zero and does not generate an error. If the first characters in the string represent a number, however, then the string is converted to a number and the non-numeric portion of the string is ignored — numeric overflow is possible.

Integer variables

An integer is a whole number that may be positive, negative, or equal to zero. Integer variables are a good choice for numbers representing non-fractional quantities, such as age or number of children. In Basic, Integer variables are 16-bit numbers supporting a range from -32768 through 32767. Floating-point numbers assigned to an integer are rounded to the nearest Integer value. Appending a variable name with “%” when it is declared is a shortcut to declaring it as type Integer.

Listing 9. Demonstrate integer variables.

```
Sub ExampleIntegerType
    Dim i1 As Integer, i2% REM i1 and i2 are both Integer variables
    Dim f2 As Double
    Dim s$
    f2= 3.5
    i1= f2 REM i1 is rounded to 4

    s = "3.50 => " & i1
    f2= 3.49
    i2= f2 REM i2 is rounded to 3
    s = s & CHR$(10) & "3.49 => " & i2
    MsgBox s
End Sub
```



Figure 19. Demonstrating integer variables in Listing 9.

Long Integer variables

“Long” is an integer type that has a greater range than type Integer. Long variables are 32-bit numbers supporting a range from -2,147,483,648 through 2,147,483,647. Long variables use twice as much memory as Integer variables but they can hold numbers that are *much* larger in magnitude. Floating-point numbers assigned to type Long are rounded to the nearest Long value. Appending a variable name with “&” when it is declared is a shortcut to declaring it as type Long. The output from Listing 10 is the same as Listing 9 (see Figure 19).

Listing 10. Demonstrate long variables.

```
Sub ExampleLongType
    Dim NumberOfDogs&, NumberOfCats As Long ' Both variables are Long
    Dim f2 As Double
    Dim s$
    f2= 3.5
    NumberOfDogs = f2 REM round to 4
    s = "3.50 => " & NumberOfDogs
```

```

f2= 3.49
NumberOfCats = f2 REM round to 3
s = s & CHR$(10) & "3.49 => " & NumberOfCats
MsgBox s
End Sub

```

Currency variables

Currency variables, as the name implies, are designed to hold financial information. The Currency type was originally introduced to avoid the rounding behavior of the floating-point types Single and Double. Visual Basic .NET removed the Currency type in favor of the Decimal type.

Currency variables are 64-bit, fixed-precision numbers. Calculations are performed to four decimal places and 15 non-decimal digits accuracy. This yields a range from -922,337,203,658,477.5808 through +922,337,203,658,477.5807. Appending a variable name with “@” when it is declared is a shortcut to declaring it as type Currency.

Listing 11. Demonstrate currency variables.

```

Sub ExampleCurrencyType
    Dim Income@, CostPerDog As Currency
    Income@ = 22134.37
    CostPerDog = 100.0 / 3.0
    REM Prints as 22134.3700
    Print "Income = " & Income@
    REM Prints as 33.3333
    Print "Cost Per dog = " & CostPerDog
End Sub

```

Single variables

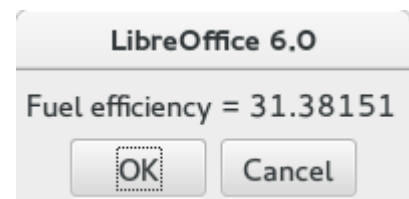
Single variables, unlike Integer variables, can have a fractional portion. They are called “floating-point numbers” because, unlike Currency variables, the number of decimals allowed is not fixed. Single variables are 32-bit numbers that are accurate to about seven displayed digits, making them suitable for mathematical operations of average precision. They support positive or negative values from 3.402823×10^{38} to 1.401298×10^{-45} . Any number smaller in magnitude than 1.401298×10^{-45} becomes zero. Appending a variable name with “!” when it is declared is a shortcut to declaring it as type Single.

Listing 12. Demonstrate single variables.

```

Sub ExampleSingleType
    Dim GallonsUsed As Single, Miles As Single, mpg!
    GallonsUsed = 17.3
    Miles = 542.9
    mpg! = Miles / GallonsUsed
    Print "Fuel efficiency = " & mpg!
End Sub

```



Double variables

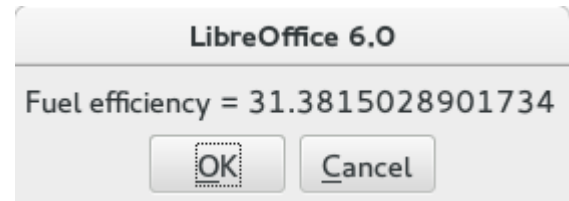
Double variables are similar to Single variables except that they use 64 bits and have about 15 significant digits. They are suitable for high-precision mathematical operations. Double variables

support positive or negative values from $1.79769313486232 \times 10^{308}$ to $4.94065645841247 \times 10^{-324}$. Any number smaller in magnitude than $4.94065645841247 \times 10^{-324}$ becomes zero. Appending a variable with “#” when it is declared is a shortcut to declaring it as type Double.

Listing 13. Demonstrate double variables.

```
Sub ExampleDoubleType
    Dim GallonsUsed As Double, Miles As Double, mpg#
    GallonsUsed = 17.3
    Miles = 542.9

    mpg# = Miles / GallonsUsed
    Print "Fuel efficiency = " & mpg#
End Sub
```



3.3.6. String variables contain text

String variables are used to hold text. In OOo, text is stored as Unicode version 2.0 values, which provides good support for multiple languages. Each String variable can hold up to 65,535 characters. Appending a variable with “\$” when it is declared is a shortcut to declaring it as type String.

Listing 14. Demonstrate string variables.

```
Sub ExampleStringType
    Dim FirstName As String, LastName$
    FirstName = "Andrew"
    LastName$ = "Pitonyak"
    Print "Hello " & FirstName & " " & LastName$
End Sub
```

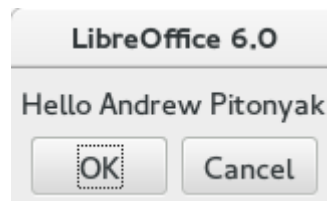


Figure 20. Demonstrating string variables in Listing 14.

In Visual Basic .NET, String variables may contain approximately 2 billion Unicode characters. LO strings used to be limited to 65,535 characters, but that limitation has been removed. I do not know the current limitation off hand, but, I did test to 10 million.

Place two double quote characters in a row to insert a double quotation character into a string.

```
S = "She said ""Hello"" " REM She said "Hello"
```

Visual Basic string constants are available when using “Option Compatible” (see Table 10). You must use Option Compatible for the module rather than using CompatibilityMode(True) because the string constant is recognized at compile time rather than run time.

Table 10. Visual Basic-compatible string constants.

Constant	Value	Description
vbCr	CHR\$(13)	Carriage return
vbCrLf	CHR\$(13) & CHR\$(10)	Carriage return/linefeed combination
vbFormFeed	CHR\$(12)	Form feed
vbLf	CHR\$(10)	Line feed
vbNewLine	CHR\$(13) & CHR\$(10) or CHR\$(10)	Platform-specific newline character — whatever is appropriate
vbNullChar	CHR\$(0)	Character with ASCII value 0
vbNullString	""	Empty string. This is a string with a terminating null.
vbTab	CHR\$(9)	Horizontal tab
vbVerticalTab	CHR\$(11)	Vertical tab

The string constants in Table 10 allow you to define constant strings with special characters. Previously, you had to define the string by using code that called the CHR\$() function.

```
Option Compatible
Const sGreeting As String = "Hello" & vbCrLf & "Johnny" ' This contains a CR.
```

3.3.7. Date variables

Date variables contain date and time values. Basic stores dates internally as a Double. Dates, like all numerical types, are initialized to zero, which corresponds to December 30, 1899 at 00:00:00. Adding or subtracting 1 to/from a date corresponds to adding or subtracting a day. One hour, one minute, and one second correspond to the numbers 1/24, 1/(24 * 60), and 1/(24 * 60 * 60), respectively. The date functions supported by Basic are introduced in Table 11 and fully discussed later.

Listing 15. Demonstrate date variables.

```
Sub ExampleDateType
    Dim tNow As Date, tToday As Date
    Dim tBirthDay As Date
    tNow = Now()
    tToday = Date()
    tBirthDay = DateSerial(1776, 7, 4)
    Print "Today = " & tToday
    Print "Now = " & tNow
    Print "A total of " & (tToday - tBirthDay) & _
        " days have passed since " & tBirthDay
End Sub
```

Negative numbers are valid and correspond to dates before December 30, 1899. January 1, 0001, corresponds to the floating-point number -693,595. Continuing backward produces dates that are B.C. (Before Christ, sometimes also referred to as B.C.E., meaning Before the Christian Era, or

Before the Common Era) rather than A.D. (Anno Domini). A thorough discussion of date handling is presented later.

Table 11. Functions and subroutines related to dates and times.

Function	Type	Description
CDate(expression)	Date	Convert a number or string to a date.
CDateFromIso(string)	Date	Convert to a date from an ISO 8601 date representation.
CDateToIso(date)	String	Convert a date to an ISO 8601 date representation.
Date()	String	Return the current date as a String.
DateSerial(yr, mnth, day)	Date	Create a date from component pieces: Year, Month, Day.
DateValue(date)	Date	Extract the date from a date/time value by truncating the decimal portion.
Day(date)	Integer	Return the day of the month as an Integer from a Date value.
GetSystemTicks()	Long	Return the number of system ticks as a Long.
Hour(date)	Integer	Return the hour as an Integer from a Date value.
IsDate(value)	Boolean	Is this a date?
Minute(date)	Integer	Return the minute as an Integer from a Date value.
Month(date)	Integer	Return the month as an Integer from a Date value.
Now()	Date	Return the current date and time as a Date object.
Second(date)	Integer	Return the seconds as an Integer from a Date value.
Time()	String	Return the time as a String.
Timer()	Date	Return the number of seconds since midnight as a Date. Convert to a Long.
TimeSerial(hour, min, sec)	Date	Create a date from component pieces: Hours, Minutes, Seconds.
WeekDay(date)	Integer	Return the integer 1 through 7, corresponding to Sunday through Saturday.
Year(date)	Integer	Return the year as an Integer from a Date value.

3.3.8. Create your own data types

In most implementations of the BASIC programming language, you can create your own data types. Lo Basic allows you to define and use your own data types.

Listing 16. Demonstrate user defined types.

```
Type PersonType
    FirstName As String
    LastName As String
End Type

Sub ExampleCreateNewType
    Dim Person As PersonType
    Person.FirstName = "Andrew"
    Person.LastName = "Pitonyak"
    PrintPerson(Person)
End Sub

Sub PrintPerson(x)
    Print "Person = " & x.FirstName & " " & x.LastName
```

TIP

Although user-defined types cannot directly contain an array, you can manage them in a variant type.

There are three ways that you can instantiate an instance of a user defined type. In the following example, the colon (:) is used to place two statements on the same line.

```
Dim x As New PersonType           ' The original way to do it.
Dim y As PersonType               ' New is no longer required.
Dim z : z = CreateObject("PersonType") ' Create the object when desired.
```

When you create your own type, you create a structure (frequently called a struct). LO has many predefined internal structures. A commonly used structure is “com.sun.star.beans.PropertyValue”. The internal LO structures can be created in the same way as user defined types, and also using CreateUnoStruct (see 10Universal Network Objects on page 252).

```
Dim a As New com.sun.star.beans.PropertyValue
Dim b As New com.sun.star.beans.PropertyValue
Dim c : c = CreateObject("com.sun.star.beans.PropertyValue")
Dim d : d = CreateUnoStruct("com.sun.star.beans.PropertyValue")
```

Although the structure's type is “com.sun.star.beans.PropertyValue”, it is common to abbreviate the type name as the last portion of the name — in this case, “PropertyValue”. Many of the objects have similar, long, cumbersome names, which are similarly abbreviated in this book.

Most variables copy by value. This means that when I assign one variable to another, the value from one is placed into the other. They do not reference the same data; they contain their own copy of the data. This is also true of user-defined types and internal LO structures. Variables that can be defined in this way are copied by value. Other types used internally by LO, called Universal Network Objects, are copied by reference. Although these are discussed later, it is important to start thinking about what happens when one variable is assigned to another. If I assign one variable to another and it is copied by reference, then both variables refer to the same data. If two variables refer to the same data, and if I change one variable, then I change both.

3.3.9. Declare variables with special types

You can use the keywords “As New” to define a variable as a known UNO struct. The word “struct” is an abbreviated form of the word “structure” that is frequently used by computer programmers. A struct has one or more data members, each of which may have different types. Structs are used to group associated data together.

Option Compatible provides a new syntax to define variables of known and unknown types. A simple example is declaring a variable of a specific type even if the type is not known to Basic.

```
Option Compatible           'Supported in OOo 2.0
Sub Main
  Dim oVar1 As Object
  Dim oVar2 As MyType
  Set oVar1 = New MyType    'Supported in OOo 2.0
```



```
Set oVar2 = New MyType 'Supported in OOo 2.0
Set oVar2 = New YourType 'Error, declared as MyType not YourType.
```

A new OLE object factory was introduced with OOo 2.0, which allows new types to be created. The new functionality allows Basic to manipulate Microsoft Word documents on Microsoft Windows operating systems if Microsoft Office is also installed on the computer.

```
Sub Main
    Dim W As Word.Application
    Set W = New Word.Application
    REM Dim W As New Word.Application 'This works in OOo 2.0
    REM W = CreateObject("Word.Application") 'This works in OOo 2.0
    W.Visible = True
End Sub
```

The use of CreateObject() does not rely on “Option Compatible” because this functionality is provided by the OLE object factory released with OOo 2.0.

3.3.10. Object variables

An Object is a complex data type that can contain more than a single piece of data. The code in Listing 16 shows an example of a complex data type. Object variables are intended to hold complex data types created and defined by Basic. When a variable of type Object is first declared, it contains the special value Null, which indicates that no valid value is present.

Historically, it has been safer to use the Variant , not Object, to refer to LO internals. This is discussed later.

3.3.11. Variant variables

Variant variables are able to hold any data type. They take on the type of whatever is assigned to them. When a variable of type Variant is first declared, it contains the special value Empty, which indicates that no value has been assigned to the variable. A Variant may be assigned an integer in one statement and then assigned text in the next. No typecasting occurs when assigning data to a Variant; it simply becomes the appropriate type.

The chameleon-like behavior of Variant variables allows them to be used as any other variable type. However, this flexibility comes with a cost: time. One final problem is that it is not always obvious what type a Variant will become after making some assignments.

Listing 17. Demonstrate variant types.

```
Sub ExampleTestVariants
    DIM s As String
    DIM v As Variant
    REM v starts Empty
    s = s & "1 : TypeName = " & TypeName(v) & " Value = " & v & CHR$(10)
    v = "ab217" : REM v becomes a String
    s = s & "2 : TypeName = " & TypeName(v) & " Value = " & v & CHR$(10)
    v = True : REM v becomes a Boolean
    s = s & "3 : TypeName = " & TypeName(v) & " Value = " & v & CHR$(10)
```

```

v = (5=5)      : REM v becomes an Integer rather than a Boolean
s = s & "4"    : TypeName = " & TypeName(v) & " Value = " & v & CHR$(10)
v = 123.456    : REM Double
s = s & "5"    : TypeName = " & TypeName(v) & " Value = " & v & CHR$(10)
v =123         : REM Integer
s = s & "6"    : TypeName = " & TypeName(v) & " Value = " & v & CHR$(10)
v = 1217568942 : REM This could be a Long but it turns into a Double
s = s & "7"    : TypeName = " & TypeName(v) & " Value = " & v & CHR$(10)
MsgBox s, 0, "The Variant Takes Many Types"
End Sub

```

Visual Basic .NET does not support type Variant. Undeclared variables are of type Object.

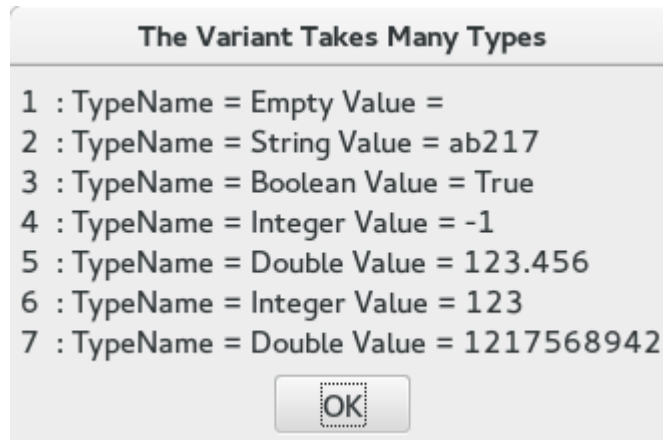


Figure 21. Variant takes the assigned type.

When data is assigned to a Variant, the data is not converted to an appropriate type, but rather the Variant becomes the type of the data. In line 6 of Figure 21, the Variant is an Integer. In line 7, the number is too large to be an Integer, but it is small enough to be a Long. Basic chooses to convert whole numbers larger than an integer and all floating-point numbers into a Double, even if they can be expressed as a Single or a Long.

3.3.12. Constants

A constant is a variable with no type, and that cannot change value. The variable is defined to be a placeholder that is substituted by the expression that defines the constant. Constants are defined with the keyword `Const`. The constant name can be any valid variable name.

```
Const ConstName=Expression
```

Constants improve macros in many ways. Consider a Gravity constant frequently used in physics. Physics scholars will recognize this as the acceleration due to gravity in meters per second squared.

```
Const Gravity = 9.81
```

Here are some specific benefits of using constants:

- Constants improve the readability of a macro. The word Gravity is easier to recognize than the value 9.81.

- Constants are easy to manage. If I require greater precision or if the gravitational pull changes, I have to change the value in only one location.
- Constants help prevent difficult-to-find errors by changing run-time errors into compile-time errors. Typing “Grevity” rather than “Gravity” is a compile-time error, while mistyping “9.81” as “9.18” is not.
- While a value like 9.81 may be obvious to you, it may not be as obvious to others reading your code later. It becomes what programmers call a “magic number,” and experienced programmers try to avoid magic numbers at all costs. Their unexplained meanings make for difficulties in maintaining code later, when the original programmer is not available to explain — or has forgotten the details entirely.

TIP OpenOffice.org defines the constant Pi. This is a mathematical constant with a value of approximately 3.1415926535897932385.

3.4. The With statement

The With statement is used to simplify accessing complex data types. Listing 16 defines a data type that contains two different variables: FirstName and LastName. You can access these variables by placing a period between the variable name and the data item.

```
Sub ExampleCreateNewType
    Dim Person As PersonType
    Person.FirstName = "Andrew"
    Person.LastName = "Pitonyak"
    ...
End Sub
```

Or... the With statement provides a shortcut for accessing multiple data elements from the same variable.

```
Sub ExampleCreateNewType
    Dim Person As PersonType
    With Person
        .FirstName = "Andrew"
        .LastName = "Pitonyak"
    End With
    ...
End Sub
```

Similarly:

```
Dim oProp As New com.sun.star.beans.PropertyValue
oProp.Name = "Person" 'Set Name Property
oProp.Value = "Boy Bill" 'Set Value Property
```

Using With:

```
Dim oProp As New com.sun.star.beans.PropertyValue
With oProp
    .Name = "Person" 'Set Name Property
```

```

    .Value = "Boy Bill"      'Set Value Property
End With

```

3.5. Arrays

An array is a data structure in which similar elements of data are arranged in an indexed structure — for example, a column of names or a table of numbers. See Table 12. An array allows you to store many different values in a single variable, and uses parentheses to define and access array elements. Basic does not support the use of square brackets as used by other languages such as C and Java.

Array variables are declared using the Dim statement. Think of a one-dimensional array as a column of values and a two-dimensional array as a table of values. Higher dimensions are supported but difficult to visualize. An array index is no longer limited to Integer values between -32,768 through 32,767.

Table 12. *Declaring an array is easy!*

Definition	Elements	Description
<code>Dim a(5) As Integer</code>	6	From 0 through 5 inclusive.
<code>Dim b(5 To 10) As String</code>	6	From 5 through 10 inclusive.
<code>Dim c(-5 To 5) As String</code>	11	From -5 through 5 inclusive.
<code>Dim d(5, 1 To 6) As Integer</code>	36	Six rows with six columns from 0 through 5 and 1 through 6.
<code>Dim e(5 To 10, 20 To 25) As Long</code>	36	Six rows with six columns from 5 through 10 and 20 through 25.

TIP You must declare array variables before using them, even if you don't use "Option Explicit."

If the lower dimension of an array is not specified, the default lower bound of an array is zero. (Programmers call these arrays "zero-based.") Thus, an array with five elements will have elements numbered a(0) through a(4). Use the keywords "Option Base 1" to change the default lower bound of an array to start at 1 rather than 0. This must be done before any other executable statement in the program.

```
Option Base { 0 | 1 }
```

TIP Specify the lower bound of an array rather than relying on the default behavior. This is more portable and it will not change when the Option Base statement is used.

Dim a(3) allows for four elements: a(0), a(1), a(2), and a(3). Option Base does not change the number of elements that an array can store; it changes only how they are indexed. Using Option Base 1, the same statement still allows for four elements: a(1), a(2), a(3), and a(4). I consider this behavior unintuitive and recommend against using Option Base. If you want specific array

bounds, the preferred option is to explicitly declare the array bounds. For example, `Dim a(1 To 4)`. Option Base has risks in terms of communicating clear documentation and ensuring portability.

Visual Basic handles Option Base 1 differently from LO Basic; VB changes the lower bound to 1 but does not change the upper bound. Visual Basic .NET no longer supports Option Base. With “Option Compatible,” “Option Base 1” does not increase the upper bound by 1. In other words, LO Basic acts like VB.

Accessing and changing the values in an array is easy. Initializing an array this way is tedious.

Listing 18. Demonstrate simple array.

```
Sub ExampleSimpleArray1
    Dim a(2) As Integer, b(-2 To 1) As Long
    Dim m(1 To 2, 3 To 4)

    REM Did you know that multiple statements can be placed
    REM on a single line if separated by a colon?
    a(0) = 0 : a(1) = 1 : a(2) = 2
    b(-2) = -2 : b(-1) = -1 : b(0) = 0 : b(1) = 1
    m(1, 3) = 3 : m(1, 4) = 4
    m(2, 3) = 6 : m(2, 4) = 8
    Print "m(2,3) = " & m(2,3)
    Print "b(-2) = " & b(-2)
End Sub
```

To quickly fill a Variant array, use the `Array` function (see Listing 19), which returns a Variant array with the included data. The functions `LBound` and `UBound` return the lower bound and upper bound of an array. Routines supported by Basic are summarized in Table 13, and discussed thoroughly later.

Listing 19. Use Array() to quickly fill an array.

```
Sub ExampleArrayFunction
    Dim a, i%, s$
    a = Array("Zero", 1, Pi, Now)
    Rem String, Integer, Double, Date
    For i = LBound(a) To UBound(a)
        s$ = s$ & i & " : " & TypeName(a(i)) & " : " & a(i) & CHR$(10)
    Next
    MsgBox s$, 0, "Example of the Array Function"
End Sub
```

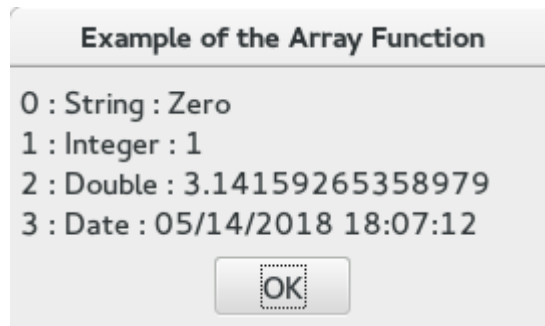


Figure 22. Different variable types in the same array.

A variable defined as an array but not dimensioned, such as `Dim a()`, is called an empty array. Test for an empty array by comparing the upper bound of the array to the lower bound. If the upper bound is less than the lower bound, the array is empty and no dimensions have been set. An array that has been dimensioned, such as `Dim a(5)`, is not empty.

The behavior for `LBound` and `UBound` has changed over time. Early releases of OOo produce an error for `UBound(b)` and some do not. All versions should work correctly with `UBound(b())`. At the time of this writing, the upper and lower array bounds for `c` (in Listing 20) fails because `c` is an empty object.

Listing 20. Parentheses are not always required but are always allowed.

```
Sub ArrayDimensionError
    On Error Goto ErrorHandler
    Dim a(), b(1 To 2), c
    Dim iLine As Integer
    Dim s$
    REM Valid constructs
    iLine = 1 : s = "a = (" & LBound(a()) & ", "
    iLine = 2 : s = s & UBound(a) & ") "d
    iLine = 3 : s = s & CHR$(10) & "b = (" & LBound(b()) & ", "
    iLine = 4 : s = s & UBound(b) & ") "
    iLine = 5 : s = s & CHR$(10) & "c = (" & LBound(c()) & ", "
    iLine = 6 : s = s & UBound(c) & ") "
    MsgBox s, 0, "LBound and UBound"
    Exit Sub
ErrorHandler:
    s = s & CHR$(10) & "Error " & Err & ": " & Error$ & " (line : " & iLine & ")"
    Resume Next
End Sub
```

Table 13. Summary of subroutines and functions related to arrays.

Function	Description
<code>Array(args)</code>	Return a Variant array that contains the arguments.
<code>DimArray(args)</code>	Return an empty Variant array. The arguments specify the dimension.
<code>IsArray(var)</code>	Return True if this variable is an array, False otherwise.
<code>Join(array)</code>	Concatenate the array elements separated by the optional string delimiter and return as a String. The default delimiter is a single space.
<code>Join(array, delimiter)</code>	

Function	Description
LBound(array) LBound(array, dimension)	Return the lower bound of the array argument. The optional dimension specifies which dimension to check. The first dimension is 1.
ReDim var(args) As Type	Change the dimension of an array using the same syntax as the DIM statement. The keyword Preserve keeps existing data intact — ReDim Preserve x(1 To 4) As Integer.
Split(str) Split(str, delimiter) Split(str, delimiter, n)	Split the string argument into an array of strings. The default delimiter is a space. The optional argument “n” limits the number of strings returned.
UBound(array) UBound(array, dimension)	Return the upper bound of the array argument. The optional dimension specifies which dimension to check. The first dimension is 1.

3.5.1. Changing the dimension of an array

The desired dimension of an array is not always known ahead of time. Sometimes, the dimension is known; but it changes periodically, and the code must be changed. An array variable can be declared with or without specifying the dimensions. Basic provides a few different methods to set or change the dimensions of an array.

The Array function generates a Variant array that contains data. This is a quick way to initialize an array. You are not required to set the dimension of the array, but, if you do, it will change to become the array returned by the Array function.

```
Dim a()  
a = Array(3.141592654, "PI", 9.81, "Gravity")
```

The arguments passed to the Array function become data in the returned Variant array. The DimArray function, on the other hand, interprets the arguments as the dimensions of an array to create (see Listing 21). The arguments can be expressions, so a variable can be used to set the dimension.

Listing 21. Redimension array.

```
Sub ExampleDimArray  
    Dim a(), i%  
    Dim s$  
    a = Array(10, 11, 12)  
    s = "" & LBound(a()) & " " & UBound(a())           Rem 0 2  
    a() = DimArray(3)                                   REM the same as Dim a(3)  
    a() = DimArray(2, 1)                                REM the same as Dim a(2,1)  
    i = 4  
    a = DimArray(3, i)                                  Rem the same as Dim a(3,4)  
    s = s & CHR$(10) & LBound(a(),1) & " " & UBound(a(),1) Rem 0, 3  
    s = s & CHR$(10) & LBound(a(),2) & " " & UBound(a(),2) Rem 0, 4  
    a() = DimArray()      REM an empty array  
    MsgBox s, 0, "Example Dim Array"  
End Sub
```

The Array and DimArray functions both return an array of Variants. The ReDim statement changes the dimension of an existing array. This can change both individual dimensions and the

number of dimensions. The arguments can be expressions because the ReDim statement is evaluated at run time.

```
Dim e() As Integer, i As Integer
i = 4
ReDim e(5) As Integer      REM Dimension is 1, a size of 0 To 5 is valid.
ReDim e(3 To 10) As Integer REM Dimension is 1, a size of 3 To 10 is valid.
ReDim e(3, i) As Integer    REM Dimension is 2, a size of (0 To 3, 0 To 4) is valid.
```

Some tips regarding arrays:

- LBound and UBound work with empty arrays.
- An empty array has one dimension. The lower bound is zero and the upper bound is -1.
- Use ReDim to cause an existing array to become empty.

The ReDim statement supports the keyword Preserve. This attempts to save the data when the dimensions of an array are changed. Increasing the dimension of an array preserves all of the data, but decreasing the dimension causes data to be lost by truncation. Data can be truncated at either end. If an element in the new array existed in the old array, the value is unchanged. Unlike some variants of BASIC, LO Basic allows all dimensions of an array to be changed while still preserving data.

```
Dim a() As Integer
ReDim a(3, 3, 3) As Integer
a(1,1,1) = 1 : a(1, 1, 2) = 2 : a(2, 1, 1) = 3
ReDim preserve a(-1 To 4, 4, 4) As Integer
Print "(" & a(1,1,1) & ", " & a(1, 1, 2) & ", " & a(2, 1, 1) & ")"
```

ReDim specifies both the dimensions and an optional type. If the type is included, it must match the type specified when the variable is declared or LO generates a compile-time error.

Listing 22 is a utility function that accepts a simple array and returns a string with all of the elements in the array. The ReDim example code, also in Listing 22, uses ArrayToString.

Listing 22. Utility function array to string.

REM ArrayToString accepts a simple array and places the value
REM of each element in the array into a string.

```
Function ArrayToString(a() As Variant) As String
    Dim i%, s$
    For i% = LBound(a()) To UBound(a())
        s$ = s$ & i% & " : " & a(i%) & CHR$(10)
    Next
    ArrayToString = s$
End Function
```

```
Sub ExampleReDimPreserve
    Dim a(5) As Integer, b(), c() As Integer
    a(0) = 0 : a(1) = 1 : a(2) = 2 : a(3) = 3 : a(4) = 4 : a(5) = 5
    Rem a is dimensioned from 0 to 5 where a(i) = i
    MsgBox ArrayToString(a()), 0, "a() at start"
```



```

Rem a is re-dimensioned from 1 to 3 where a(i) = i
ReDim Preserve a(1 To 3) As Integer
MsgBox ArrayToString(a()), 0, "a() after ReDim"

Rem Array() returns a Variant type
Rem b is dimensioned from 0 to 9 where b(i) = i+1
b = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
MsgBox ArrayToString(b()), 0, "b() at initial assignment"

Rem b is dimensioned from 1 to 3 where b(i) = i+1
ReDim Preserve b(1 To 3)
MsgBox ArrayToString(b()), 0, "b() after ReDim"

Rem The following is NOT valid because the array is already dimensioned
Rem to a different size
Rem a = Array(0, 1, 2, 3, 4, 5)

Rem c is dimensioned from 0 to 5 where c(i) = i
Rem If "ReDim" had been done on c, then this would NOT work
c = Array(0, 1, 2, "three", 4, 5)
MsgBox ArrayToString(c()), 0, "Integer array c() Assigned to a Variant"

Rem Ironically, this is allowed but c will contain no data!
ReDim Preserve c(1 To 3) As Integer
MsgBox ArrayToString(c()), 0, "ReDim Integer c() after assigned Variant"
End Sub

```

Visual Basic has different rules for changing the dimensions of an array, and these rules change between versions of Visual Basic. As a general rule, LO Basic is more flexible.

3.5.2. The unexpected behavior of arrays

Assigning one Integer variable to another copies the value, and the variables are not related in any other way. In other words, if you change the first variable's value, the second variable's value does not change. This is not true for array variables. Assigning one array variable to another makes a reference to the first array rather than copying the data. All changes made to either are automatically seen by the other. It doesn't matter which one is changed; they are both affected. This is the difference between "pass by *value*" (integers) and "pass by *reference*" (arrays).

Listing 23. Arrays copy as references.

```

Sub ExampleArrayCopyIsRef
    Dim a(5) As Integer, c(4) As Integer, s$
    c(0) = 4 : c(1) = 3 : c(2) = 2 : c(3) = 1 : c(4) = 0

    a() = c()
    a(1) = 7
    c(2) = 10
    s$ = "***** a() *****" & CHR$(10) & ArrayToString(a()) & CHR$(10) & _
        "***** c() *****" & CHR$(10) & ArrayToString(c())
    MsgBox s$, 0, "Change One, Change Both"
End Sub

```

To illustrate that arrays are assigned by reference, create three arrays — a(), b(), and c() — as shown in Figure 23. Internally, Basic creates three arrays that are referenced by a(), b(), and c().

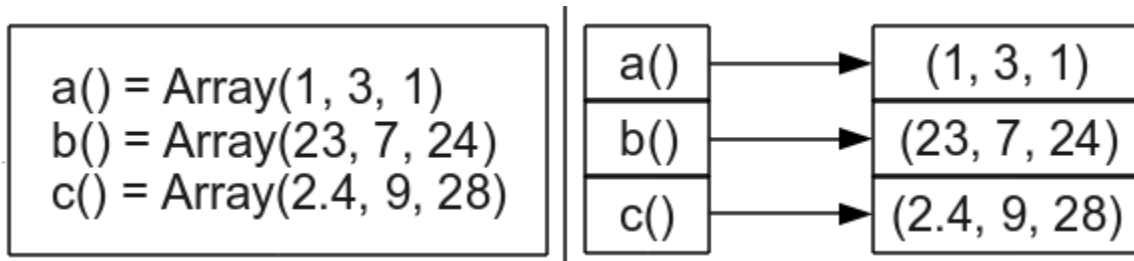


Figure 23. Assigning an array assigns a reference.

Assign array a() to array b(), and both a() and b() reference the same data. The variable a() does not reference the variable b(), it references the same data referenced by b() (see Figure 24). Therefore, changing a() also changes b(). The original array referenced by a() is no longer referenced.

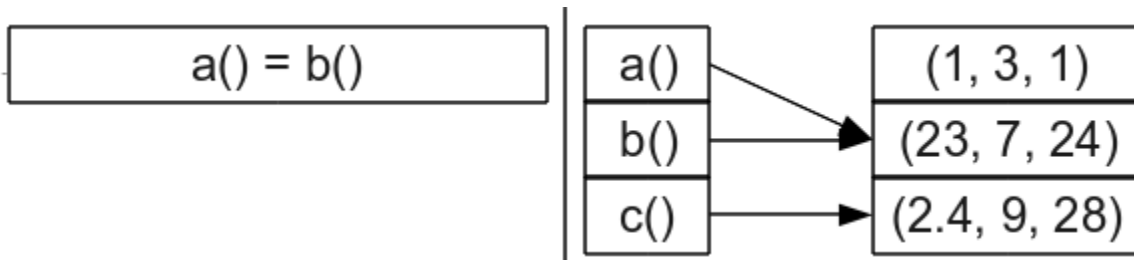


Figure 24. Assigning an array assigns a reference.

Assign array b() to array c(), and both b() and c() reference the same data. The variable a() remains unchanged, as shown in Figure 25.

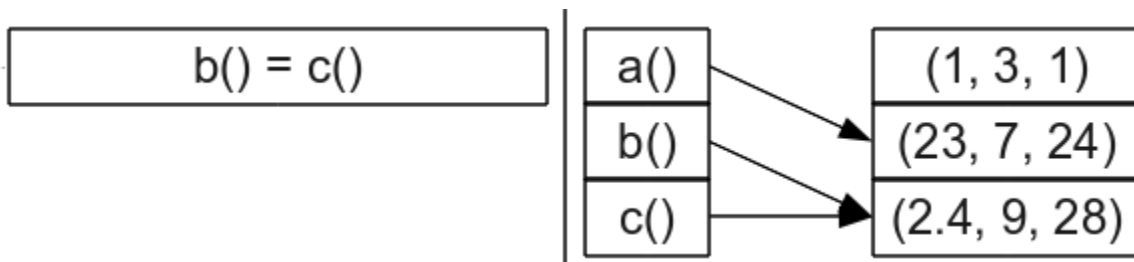


Figure 25. Assigning an array assigns a reference.

TIP Type checking is not performed when an array is assigned to another array. Do not assign arrays of different types to each other.

Because no type checking is performed when an array is assigned to another array, unexpected and obscure problems can occur. The Array function returns a Variant array, and is the quickest method to assign multiple values to an array variable. An obvious problem is that an Integer array may contain String values if it references a Variant array. A less obvious problem is that the ReDim

statement works based on the declared type. The statement “ReDim Preserve” on an Integer array assigned to a Variant array fails to preserve the data.

```
Dim a() As Integer          REM Declare a() as an Integer()
a() = Array(0, 1, 2, 3, 4, 5, 6) REM Assign a Variant() to an Integer()
ReDim Preserve a(1 To 3) As Integer REM This wipes the array
```

To safely assign arrays while maintaining the correct data type, another method is required. Copy each element in the array individually. This also prevents two array variables from referencing the same array.

Listing 24. More complex array example.

```
Sub ExampleSetIntArray
    Dim iA() As Integer
    SetIntArray(iA, Array(9, 8, "7", "six"))
    MsgBox ArrayToString(iA), 0, "Assign a Variant to an Integer"
End Sub

REM Dimension the first array to have the same dimensions as the second.
REM Perform an element-by-element copy of the array.
Sub SetIntArray(iArray() As Integer, v() As Variant)
    Dim i As Long
    ReDim iArray(LBound(v()) To UBound(v())) As Integer
    For i = LBound(v) To UBound(v)
        iArray(i) = v(i)
    Next
End Sub
```

3.6. Subroutines and functions

Subroutines are used to group lines of code into meaningful pieces of work. A function is a subroutine that returns a value. The use of subroutines and functions facilitates testing, code reuse, and readability. This in turn reduces errors.

The keyword Sub defines the beginning of a subroutine, and End Sub defines the end of a subroutine.

```
Sub FirstSub
    Print "Running FirstSub"
End Sub
```

To use a subroutine, place the name of the subroutine that you want to call on a line. The name can optionally be preceded by the keyword Call.

```
Sub Main
    Call FirstSub ' Call Sub FirstSub
    FirstSub      ' Call Sub FirstSub again
End Sub
```

Subroutine and function names must be unique in a module. They are subject to the same naming conventions as variables, including the use of spaces in their names.

```
Sub One
```

```

    [name with space]
End Sub
Sub [name with space]
    Print "I am here"
End Sub

```

Visual Basic allows a subroutine to be preceded by optional keywords such as Public or Private. Starting with OOO 2.0, you can define a routine as public or private, but the routine is always public unless CompatibilityMode(True) is used first.

Declare a subroutine as private by preceding Sub with the keyword Private.

```

Private Sub PrivSub
    Print "In Private Sub"
    bbxx = 4
End Sub

```

Using Option Compatible is not sufficient to enable Private scope, CompatibilityMode(True) must be used.

```

Sub TestPrivateSub
    CompatibilityMode(False) 'Required only if CompatibilityMode(True) already used.
    Call PrivSub()           'This call works.
    CompatibilityMode(True)  'This is required, even if Option Compatible is used
    Call PrivSub()           'Runtime error (if PrivSub is in a different module).
End Sub

```

The keyword Function is used to declare a function which, like a variable, can define the type it returns. If the type is not declared, the return type defaults to Variant. You can assign the return value at any point and as many times as you want before the function ends. The last value assigned is returned.

```

Sub test
    Print "The function returns " & TestFunc
End Sub
Function TestFunc As String
    TestFunc = "hello"
End Function

```

3.6.1. Arguments

A variable that is passed to a routine is called an argument. Arguments must be declared. The same rules for declaring variable types apply to declaring argument types.

A routine name can optionally be followed by parentheses, both when it is defined and when it is called. A routine that accepts arguments can optionally enclose the argument list in parentheses. The argument list follows the routine name on the same line. Blank space is allowed between the name of the routine and the argument list.

Listing 25. Simple argument testing.

```

Sub ExampleParamTest1()
    Call ParamTest1(2, "Two")
    Call ParamTest1 1, "One"

```

```

End Sub
Sub ParamTest1(i As Integer, s$)
    Print "Integer = " & i & " String = " & s$
End Sub

```

Pass by reference or by value

By default, arguments are passed by reference rather than by value. In other words, when the called subroutine modifies an argument, the caller sees the change. You can override this behavior by using the ByVal keyword. This causes a copy of the argument (rather than a reference to the argument) to be sent (see Listing 26 and Figure 26).

TIP

Constants passed as arguments by reference cause unexpected behavior if their value is modified in the called routine. The value may arbitrarily change back inside the called routine. For example, I had a subroutine that was supposed to decrement an Integer argument in a loop until it was zero; the argument never became zero.

Listing 26. Arguments by reference and by value.

```

Sub ExampleArgumentValAndRef()
    Dim i1%, i2%
    i1 = 1 : i2 = 1
    ArgumentValAndRef(i1, i2)
    MsgBox "Argument passed by reference was 1 and is now " & i1 & CHR$(10) & _
        "Argument passed by value was 1 and is still " & i2 & CHR$(10)
End Sub

Sub ArgumentValAndRef(iRef%, ByVal iVal)
    iRef = iRef + 1 ' This will affect the caller
    iVal = iVal - 1 ' This will not affect the caller
End Sub

```

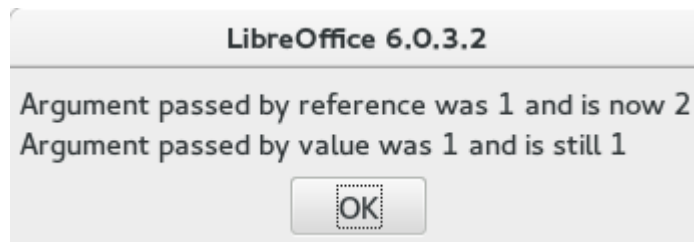


Figure 26. Pass by reference allows changes to be passed back to the caller.

Visual Basic supports the optional keyword ByRef. This keyword was introduced into Basic starting with OOo 2.0. Note that pass by reference is the default behavior.

A variable cannot be passed by value if the type does not match. The macros in Listing 27 and Listing 28 differ in the type used to declare the argument.

Listing 27. Simple swap with a string argument.

```

Sub sSwap( sDatum As String )
    Dim asDatum(1 to 3) As String
    Dim sDummy As String

    asDatum=Split(sDatum, ".")
    sDummy=asDatum(0)

```

```

asDatum(0)=asDatum(2)
asDatum(2)=sDummy
sDatum=Join(asDatum, "-")
End Sub

```

Listing 28. *Simple swap with a variant argument.*

```

Sub vSwap( vDatum As Variant )
    Dim asDatum(1 to 3) As String
    Dim sDummy As String

    asDatum=Split(vDatum, ".")
    sDummy=asDatum(0)
    asDatum(0)=asDatum(2)
    asDatum(2)=sDummy
    vDatum=Join(asDatum, "-")
End Sub

```

The following macro uses a variant and a string to call a macro that accepts a variant, and the macro that accepts the string. Passing a variant to a method that accepts a string argument passes the value by reference. The unexpected thing is that passing a string to the method that accepts a variant, passes the value by reference.

Listing 29. *Test reference by argument type.*

```

Sub passByReferenceTester
    Dim vVar As Variant
    Dim sVar As string
    Dim s As String

    vVar="01.02.2011"
    sVar="01.02.2011"

    s = vVar & " sSwap( variant var string param ) ==> "
    sSwap(vVar)
    s = s & vVar & CHR$(10)

    s = s & sVar & " sSwap( string var string param ) ==> "
    sSwap(sVar)
    s = s & sVar & CHR$(10)

    vVar="01.02.2011"
    sVar="01.02.2011"
    s = s & vVar & " vSwap( variant var variant param ) ==> "
    vSwap(vVar)
    s = s & vVar & CHR$(10)

    s = s & sVar & " vSwap( string var variant param ) ==> "
    vSwap(sVar)
    s = s & sVar & CHR$(10)

    MsgBox(s)
End Sub

```

It is important that you understand when a variable is passed to a method as a reference or a value. It is equally important to understand when the value contained in a variable is copied by value or copied by reference.

- Variables with simple types copy by value; for example, assigning one integer variable to another.
- Arrays always copy by reference. If you assign one array to another, both variables reference and modify the same array.
- UNO Services copy by reference. This means that you can do things such as `oDoc = ThisComponent`, and both variables reference the same object.
- Structs copy by value. This frustrates many people when they first encounter the behavior, but there is a very good reason for it. First, the problem;
`oBorder.TopLine.OuterLineWidth = 2` fails because `TopLine` is a struct and the value is returned as a copy rather than a reference. The code as shown changes the outer line width on a copy of the struct rather than the struct associated with the border object. The correct way to change the border is `(v = oBorder.TopLine : v.OuterLineWidth = 2 : oBorder.TopLine = v)`.

A lead developer on the original OpenOffice.org claimed that one or two services and structs do not assign / copy as expected, but he could not remember which. I have not encountered the objects that violate the guidelines, but, I try to remember the problem so that I am not taken unaware.

Optional arguments

You can declare arguments as optional by preceding them with the keyword `Optional`. All of the arguments following an optional argument must also be optional. Use the `IsMissing` function to determine if an optional argument is missing.

Listing 30. Optional arguments.

```
REM Make test calls with optional arguments.
REM Calls with Integer and Variant arguments should yield the same result.
REM Unfortunately, they do not.
Sub ExampleArgOptional()
    Dim s$
    s = "Variant Arguments () => " & TestOpt() & CHR$(10) & _
        "Integer Arguments () => " & TestOptI() & CHR$(10) & _
        "-----" & CHR$(10) & _
        "Variant Arguments (,,) => " & TestOpt(,,) & CHR$(10) & _
        "Integer Arguments (,,) => " & TestOptI(,,) & CHR$(10) & _
        "-----" & CHR$(10) & _
        "Variant Arguments (1) => " & TestOpt(1) & CHR$(10) & _
        "Integer Arguments (1) => " & TestOptI(1) & CHR$(10) & _
        "-----" & CHR$(10) & _
        "Variant Arguments (,2) => " & TestOpt(,2) & CHR$(10) & _
        "Integer Arguments (,2) => " & TestOptI(,2) & CHR$(10) & _
```

```

"-----" & CHR$(10) &_
"Variant Arguments (1,2) => " & TestOpt(1,2) & CHR$(10) &_
"Integer Arguments (1,2) => " & TestOptI(1,2) & CHR$(10) &_
"-----" & CHR$(10) &_
"Variant Arguments (1,,3) => " & TestOpt(1,,3) & CHR$(10) &_
"Integer Arguments (1,,3) => " & TestOptI(1,,3) & CHR$(10)
MsgBox s, 0, "Optional Arguments of Type Variant or Integer"
End Sub

REM Return a string that contains each argument. If the argument
REM is missing, then an M is used in its place.
Function TestOpt(Optional v1, Optional v2, Optional v3) As String
    TestOpt = "" & IIF(IsMissing(v1), "M", Str(v1)) &_
               IIF(IsMissing(v2), "M", Str(v2)) &_
               IIF(IsMissing(v3), "M", Str(v3))
End Function

REM Return a string that contains each argument. If the argument
REM is missing, then an M is used in its place.
Function TestOptI(Optional i1%, Optional i2%, Optional i3%) As String
    TestOptI = "" & IIF(IsMissing(i1), "M", Str(i1)) &_
                  IIF(IsMissing(i2), "M", Str(i2)) &_
                  IIF(IsMissing(i3), "M", Str(i3))
End Function

```

You can omit any optional arguments. Listing 30 demonstrates two functions that accept optional arguments. The functions are the same except for the argument types. Each function returns a string containing the argument values concatenated together. Missing arguments are represented by the letter “M” in the string. Although the return values from TestOpt and TestOpt1 should be the same for the same argument lists, they are not (see Figure 27). This is a bug.

TIP The IsMissing function returns incorrect results for variables that are not of type Variant when the missing argument is followed by a comma.

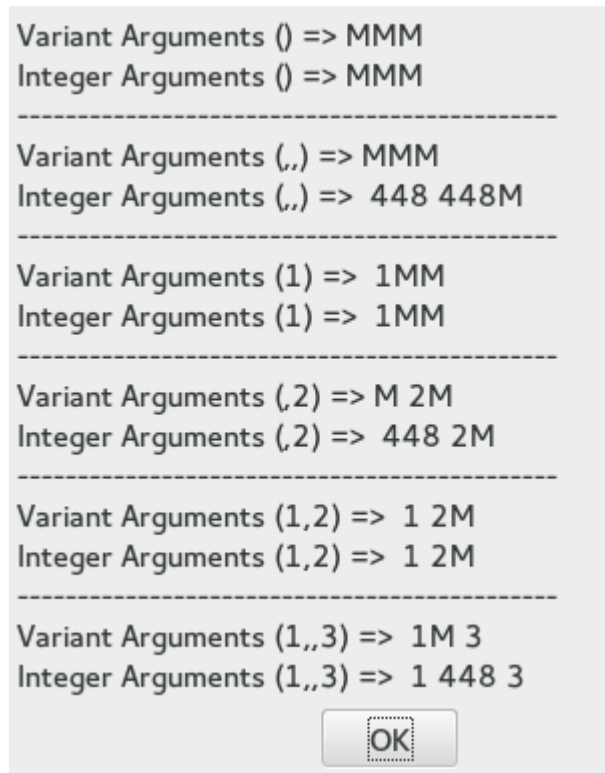


Figure 27. In rare cases, non-Variant optional arguments fail.

Default argument values

OOo version 2.0 introduced default values for missing arguments. This allows a default value to be specified if an optional argument is missing. You must use the keywords “Option Compatible” for default values to work.

```
Option Compatible
Sub DefaultExample(Optional n as Integer=100)
    REM If IsMissing(n) Then n = 100 'I will not have to do this anymore!
    Print n
End Sub
```

3.6.2. Recursive routines

A recursive routine calls itself. Consider calculating the mathematical function Factorial for positive integers. The usual definition is recursive.

Listing 31. Recursively generate factorial.

```
Sub DoFactorial
    Print "Recursive Factorial = " & RecursiveFactorial(4)
    Print "Iterative Factorial = " & IterativeFactorial(4)
End Sub

Function IterativeFactorial(ByVal n As Long) As Long
    Dim answer As Long
    answer =
```

```

Do While n > 1
    answer = answer * n
    n = n - 1
Loop
IterativeFactorial = answer
End Function

' This finally works in version 1.1
Function RecursiveFactorial(ByVal n As Long) As Long
    RecursiveFactorial = 1
    If n > 1 Then RecursiveFactorial = n * RecursiveFactorial(n-1)
End Function

```

Computers use a data structure called a stack. At home, I have a stack of books that I want to read. When I receive a new book, I place it on top of the stack. When I have time to read, I take the top book from the stack. This is similar to the data structure that a computer uses: a section of memory in a computer for temporary storage in which the last item stored is the first retrieved. Stacks are usually used when a computer calls a routine and passes arguments. A typical procedure follows:

1. Push the current run location onto the stack.
2. Push each argument onto the stack.
3. Call the desired function or subroutine.
4. The called routine uses the arguments from the stack.
5. The called routine frequently uses the stack to store its own variables.
6. The called routine removes the arguments from the stack.
7. The called routine removes and saves the caller's location from the stack.
8. If the called routine is a function, the return value is placed on the stack.
9. The called routine returns to the caller from the saved location on the stack.
10. If the called routine is a function, the return value is taken from the stack.

Although various optimizations are used, there is always some overhead associated with calling subroutines and functions. There is overhead in running time and in the memory required. The recursive version of Factorial continually calls itself. While calculating the factorial of four, there is one point at which the stack contains information for calls for 4, 3, 2, and 1. For some functions — the Fibonacci series, for example — this call behavior may be prohibitive, and a non-recursive algorithm should be used instead.

3.7. Scope of variables, subroutines, and functions

The idea of scope deals with the lifetime and visibility of a variable, subroutine, or function in Basic. The scope depends on the location of the declaration, and the keywords Public, Private,

Static, and Global. Dim is equivalent to Private, but variables are Private only if CompatibilityMode(True) is used.

3.7.1. Local variables defined in a subroutine or function

Variables declared inside a subroutine or function are called local variables. It is also commonly said that a variable is local to a routine if the variable is declared inside that routine.

You can declare a variable inside a subroutine or function by using the Dim keyword. Variables defined inside a routine are visible only inside that routine. It is not possible to directly access a variable defined inside a routine from outside the routine. However, it is possible to access a variable defined outside any routine — for example, in a module header — from inside a routine. When a variable or routine name is encountered inside a routine, Basic starts looking for the variable or routine in the following order: current routine, module, library, and other open libraries. In other words, it starts inside and works its way out.

Variables defined in a routine are created and initialized each time the routine is entered. The variables are destroyed every time the routine is exited because the routine is finished. Leaving the routine to call another routine does not cause the variables to be reinitialized.

Use the keyword Static to change a variable's creation and destruction times to the calling macro's start and finish times, respectively. Although the variable is visible only in the routine containing the variable, the variable is initialized once when the macro starts running, and the variable's values are retained through multiple calls to the same routine. In other words, you start with no macro running. The first time that a subroutine or function that contains a static variable is called, the static variables contain initial values based on their types. The static variables retain their values between calls as long as the macro as a whole did not stop running. The keyword Static uses the same syntax as the keyword Dim, and is valid only inside a subroutine or function. Listing 32 calls a routine that uses a static variable.

Listing 32. Static example.

```
Sub ExampleStatic
    ExampleStaticWorker()
    ExampleStaticWorker()
End Sub

Sub ExampleStaticWorker
    Static iStatic1 As Integer
    Dim iNonStatic As Integer

    iNonStatic = iNonStatic + 1
    iStatic1 = iStatic1 + 1
    MsgBox "iNonStatic = " & iNonStatic & CHR$(10) & _
        "iStatic1 = " & iStatic1
End Sub
```

3.7.2. Variables defined in a module

The Dim, Global, Public, or Private statements are used to declare variables in a module header. Global, Public, and Private use the same syntax as the Dim statement, but they can't declare variables inside a subroutine or function. Each variable type has a different life cycle, as summarized in Table 14.

The keywords Static, Public, Private, and Global are not used as modifiers to the keyword Dim; they are used instead of the keyword Dim.

Although it is sometimes necessary to define a variable in a module header, you should avoid it if possible. Variables defined in the header can be seen in other modules that don't expect them. It's difficult to determine why the compiler claims that a variable is already defined if it is defined in another library or module. Even worse, two working libraries may stop working because of naming conflicts.

Table 14. *Life cycle of a variable defined in a module header.*

Keyword	Initialized	Dies	Scope
Global	Compile time	Compile time	All modules and libraries.
Public	Macro start	Macro finish	Declaring library container.
Dim	Macro start	Macro finish	Declaring library container.
Private	Macro start	Macro finish	Declaring module.

Global

Use Global to declare a variable that is available to every module in every library. The library containing the Global variable must be loaded for the variable to be visible.

When a library is loaded, it is automatically compiled and made ready for use; this is when a Global variable is initialized. Changes made to a Global variable are seen by every module and are persisted even after the macro is finished. Global variables are reset when the containing library is compiled. Exiting and restarting OpenOffice.org causes all libraries to be compiled and all Global variables to be initialized. Modifying the module containing the Global definition also forces the module to be recompiled.

```
Global iNumberOfTimesRun
```

Variables declared Global are similar to variables declared Static, but Static works only for local variables, and Global works only for variables declared in the header.

Public

Use Public to declare a variable that is visible to all modules in the declaring library container. Outside the declaring library container, the public variables aren't visible. Public variables are initialized every time a macro runs.

An application library is a library that is declared in the “LibreOffice” library container. This is available when LO is running, is stored in its own directory, and every document can view it. Document-level libraries are stored in documents. The libraries are saved as part of the document and are not visible outside the document.

Public variables declared in an application library are visible in every document-level library. Public variables declared in a library contained in a document are not visible in application-level libraries. Declaring a Public variable in a document library effectively hides a Public variable declared in an application library. Simply stated (see Table 15), if you declare a Public variable in a document, it is visible only in the document and it will hide a Public variable with the same name declared outside the document. A Public variable declared in the application is visible everywhere — unless a variable declaration with more local scope takes priority over the declaration with more global scope.

```
Public oDialog As Object
```

Table 15. *The scope of a Public variable depends on where it is declared.*

Declaration Location	Scope
Application	Visible everywhere.
Document	Visible only in the declaring document.
Application and Document	Macros in the document are unable to see the application-level variable.

Private or Dim

Use Private or Dim to declare a variable in a module that should not be visible in another module. Private variables, like Public variables, are initialized every time a macro runs. A single variable name may be used by two different modules as their own variable if the variable is declared Private.

```
Private oDialog As Variant
```

- Declaring a variable using Dim is equivalent to declaring a variable as Private.
- Private variables are only private, however, only with CompatibilityMode(True).
- Option Compatible has no affect on private variables.

A Private variable is visible outside the declaring module unless CompatibilityMode(True) is used. To see for yourself, create two modules — Module1 and Module2 — in the same library. In Module1, add the declaration “Private priv_var As Integer”. Macros in Module2 can use the variable “priv_var”. Even if Module2 is located in a different library in the same document, the variable “priv_var” is visible and usable. If CompatibilityMode(True) is used, however, then the private variable is no longer visible outside of the declaring module.

In Module1, declare a variable “Private priv_var As Double”. A variable of the same name is declared in Module2, but it is an Integer variable. Each module sees its own Private variable.

Changing these two variables to have Public scope rather than Private introduces an ugly situation; only one of these is visible and usable, but you don't know which one it is without performing a test. Assign the value 4.7 to the variable, and see if it is an Integer or a Double.

3.8. Operators

An operator is a symbol that denotes or performs a mathematical or logical operation. An operator, like a function, returns a result. For example, the + operator adds two numbers. The arguments to the operator are called operands. Operators are assigned a precedence. An operator with a precedence of 1 is said to have a high precedence level; it is, after all, number 1!

TIP While typesetting mathematical equations, the minus sign (−) is represented using the Unicode character U+2212. With Basic, however, ASCII code 45 (-) must be used instead.

In Basic (see Table 16), operators are evaluated from left to right with the restriction that an operator with a higher precedence is used before an operator with a lower precedence. For example, $1 + 2 * 3$ evaluates to 7 because multiplication has higher precedence than addition. Parentheses may be used to modify the order of evaluation. For example, $(1+2) * 3$ evaluates to 9 because the expression inside the parentheses is evaluated first.

Table 16. Operators supported by OpenOffice.org Basic.

Precedence	Operator	Type	Description
1	NOT	Unary	Logical or bit-wise NOT
1	-	Unary	Leading minus sign, negation
1	+	Unary	Leading plus sign
2	^	Binary	Numerical exponentiation. Standard mathematical precedence would have exponentiation higher than negation.
3	*	Binary	Numerical multiplication
3	/	Binary	Numerical division
4	MOD	Binary	Numerical remainder after division
5	\	Binary	Integer division
6	-	Binary	Numerical subtraction
6	+	Binary	Numerical addition and string concatenation
7	&	Binary	String concatenation
8	IS	Binary	Do both operands reference the same object?
8	=	Binary	Equals
8	<	Binary	Less than
8	>	Binary	Greater than
8	<=	Binary	Less than or equal to
8	>=	Binary	Greater than or equal to
8	<>	Binary	Not equal
9	AND	Binary	Bit-wise for numerics and logical for Boolean

Precedence	Operator	Type	Description
9	OR	Binary	Bit-wise for numerics and logical for Boolean
9	XOR	Binary	Exclusive OR, bit-wise for numerics and logical for Boolean
9	EQV	Binary	Equivalence, bit-wise for numerics and logical for Boolean
9	IMP	Binary	Implication bit-wise for numerics and logical for Boolean.

TIP

Precedence does not follow standard mathematical rules; for example, negation should have a lower precedence than exponentiation so -1^2 should be -1 , not 1 .

Visual Basic uses a different precedence for operators — for example, numerical exponentiation and negation are switched, as are integer division and remainder after division.

The word “binary” means something made of or based on two things. “Unary” means something made of or based on one thing. A binary operator, not to be confused with a binary number, is placed between two operands. For example, the addition operator uses two operands with $1+2$. In Basic, binary operators are always evaluated from left to right based on operator precedence. A unary operator requires one operand that is placed directly to the right of the operator. For example, $-(1+3)$. By necessity, a series of unary operators are evaluated right to left. For example, $+-(1+3)$ must evaluate the rightmost negation operator first.

3.8.1. Mathematical and string operators

Mathematical operators can be used with all numerical data types. When operands of different types are mixed, a conversion is made to minimize the loss of precision. For example, $1 + 3.443$ causes a conversion to a floating-point number rather than a conversion to an Integer. If the first operand is a number and the second operand is a string, the string is converted to a number. If the string does not contain a valid numerical value, a zero is returned and no error is generated. Assigning a string directly to a numeric variable, however, always assigns the value zero and no errors are generated.

Listing 33. *Strings are automatically converted to numbers when required.*

```
Dim i As Integer
i = "abc"           'Assigning a string with no numbers yields zero not an error
Print i             '0
i = "3abc"          'Assigns 3, automatically converts as it can.
Print i             '3
Print 4 + "abc"     '4
```

Basic tries to automatically convert types. No errors are generated when a string is used where a number is required. This is discussed in depth later.

Unary plus (+) and minus (-)

Basic allows unary operators to have spaces between the operator and the operand (see Table 9). Unary operators also have the highest precedence and are evaluated from right to left. A leading

plus sign is arguably useless — it emphasizes that a constant is not negative but is otherwise effectively ignored. A leading minus sign indicates numeric negation.

Exponentiation (^)

Numerical exponentiation supports integer and floating-point exponents. The exponentiation operator can operate on a negative number only if the exponent is an integer.

```
result = number^exponent
```

A positive integer exponent has a conceptually simple representation. The number is multiplied by itself exponent times. For example, $2^4 = 2 * 2 * 2 * 2$.

1. OOo does not follow standard mathematical rules for exponentiation:
2. Exponentiation has a lower precedence than negation, so -1^2 incorrectly evaluates to 1. Basic evaluates multiple exponents (2^3^4) left to right ($(2^3)^4$), while standard mathematical precedence evaluates right to left. ($2^{(3^4)}$).

Listing 34. Demonstrate exponentiation.

```
Sub ExampleExponent
    Dim s$
    s = "2^3 = " & 2^3          REM 2*2*2 = 8
    s = s & CHR$(10) & "3^2 = " & 3^2      REM 3 *3 = 9
    s = s & CHR$(10) & "-3^2 = " & -3^2     REM (-3) * (-3) = 9
    s = s & CHR$(10) & "2^3^2 = " & 2^3^2   REM 2^3^2 = 8^2 = 64
    s = s & CHR$(10) & "4^0.5 = " & 4^.5     REM 2
    s = s & CHR$(10) & "4^-0.5 = " & 4^-.5   REM .5
    s = s & CHR$(10) & "-1^2 = " & -1^2     REM 1
    s = s & CHR$(10) & "-(1^2) = " & -(1^2) REM -1
    MsgBox s
End Sub
```

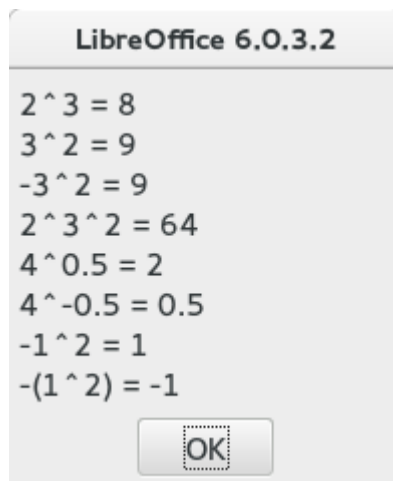


Figure 28. Using the exponentiation operator.

Multiplication (*) and Division (/)

Multiplication and division have the same precedence.

Listing 35. Demonstrate multiplication and division.

```
Sub ExampleMultDiv
    Print "2*3 = " & 2*3      REM 6
    Print "4/2.0 = " & 4/2.0  REM 2
    Print "-3/2 = " & -3/2    REM -1.5
    Print "4*3/2 = " & 4*3/2  REM 6
End Sub
```

Remainder after division (MOD)

The MOD operator is also called “remainder after division.” For example, 5 MOD 2 is 1 because 5 divided by 2 is 2 with a remainder of 1. All operands are rounded to Integer values before the operation is performed.

Listing 36. Definition of the MOD operator for integer operands x and y.

```
x MOD y = x - (y * (x\y))
```

Listing 37. Demonstrate mod operator.

```
REM x MOD y can also be written as
REM CInt(x) - (CInt(y) * (CInt(x)\CInt(y)))
REM CInt is used because the numbers must be rounded
REM before the operations are performed.
Sub ExampleMOD
    Dim x(), y(), s$, i%
    x() = Array (4, 15, 6, 6.4, 6.5, -15, 15, -15)
    y() = Array (15, 6, 3, 3, 3, 8, -8, -8)
    For i = LBound(x()) To UBound(x())
        s = s & x(i) & " MOD " & y(i) & " = " & (x(i) MOD y(i)) & CHR$(10)
    Next
    MsgBox s, 0, "MOD operator"
End Sub
```

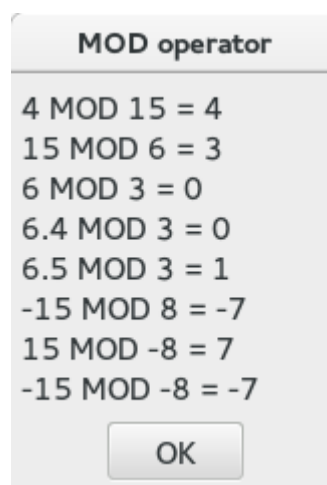


Figure 29. Using the MOD operator.

TIP

The operands for MOD are rounded to Integer values before the division is done.

Integer division (\)

Regular division expects to divide a Double by a Double, and it returns a Double as an answer. For example, $7.0 / 4.0$ is 1.75. Integer division, on the other hand, expects to divide two Integers, and it returns an Integer as an answer. For example, $7.2 \backslash 4.3$ converts the operands to $7 \backslash 4$ and then returns 1. The numeric constant operands used with the Integer division operator are truncated to Integer values, and then Integer division is performed. The result is a truncated result, not a rounded result. Listing 38 compares the difference between Integer division and regular division.

Listing 38. Demonstrate integer division.

```
Sub ExampleIntDiv
    Dim f As Double
    Dim s$
    f = 5.9
    s = "5/2 = " & 5/2          REM 2.5
    s = s & CHR$(10) & "5\2 = " & 5\2      REM 2
    s = s & CHR$(10) & "5/3 = " & 5/3      REM 1.666666667
    s = s & CHR$(10) & "5\3 = " & 5\3      REM 1
    s = s & CHR$(10) & "5/4 = " & 5/4      REM 1.25
    s = s & CHR$(10) & "5\4 = " & 5\4      REM 1
    s = s & CHR$(10) & "-5/2 = " & -5/2     REM -2.5
    s = s & CHR$(10) & "-5\2 = " & -5\2     REM -2
    s = s & CHR$(10) & "-5/3 = " & -5/3     REM -1.666666667
    s = s & CHR$(10) & "-5\3 = " & -5\3     REM -1
    s = s & CHR$(10) & "-5/4 = " & -5/4     REM -1.25
    s = s & CHR$(10) & "-5\4 = " & -5\4     REM -1
    s = s & CHR$(10) & "17/6 = " & 17/6     REM 2.83333333333333
    s = s & CHR$(10) & "17\6 = " & 17\6     REM 2
    s = s & CHR$(10) & "17/5.9 = " & 17/5.9 REM 2.88135593220339
    s = s & CHR$(10) & "17\5 = " & 17\5     REM 3
    s = s & CHR$(10) & "17\5.9 = " & 17\5.9 REM 3 because 5.9 was truncated to 5.
    s = s & CHR$(10) & "17\f = " & 17\f     REM 2 because f was rounded up to 6.
    s = s & CHR$(10) & "17\ (11.9/2) = " & 17\ (11.9/2) REM 3 because 11.9/2 truncated to 5.
    MsgBox s
End Sub
```

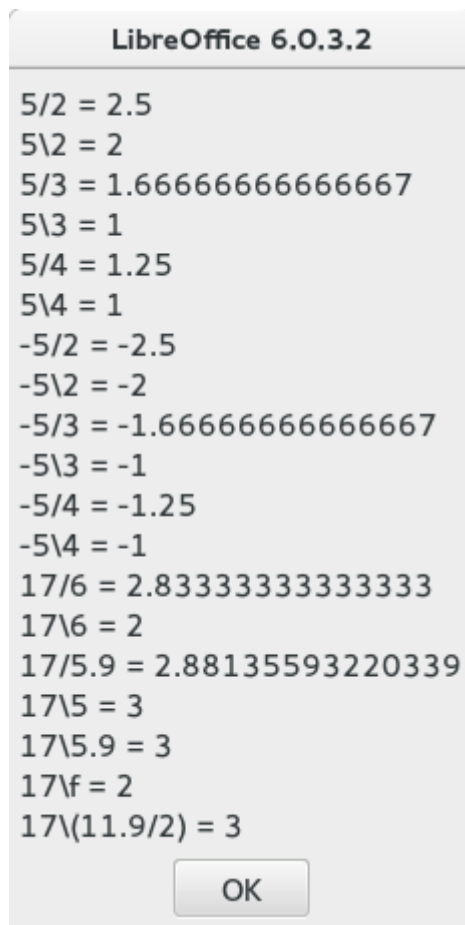


Figure 30. Integer division.

With integer division, constant numeric operands are truncated to Integer values before the division is done. When variables are part of an operand, the operand's result is rounded, otherwise truncated.

Addition (+), subtraction (-), and string concatenation (& and +)

Addition and subtraction have the same precedence, which is higher than the string concatenation operator. Care must be taken while adding numerical values because the plus operator can also signify string concatenation. When the first operand for the plus operator is a number, and the second is a string; the string is converted to a number. When the first operand for the plus operator is a string, and the second is a number; the number is converted to a string.

```
Print 123 + "3"      REM 126 (Numeric)
Print "123" + 3      REM 1233 (String)
```

The string operator tries to convert the operands to strings if at least one operand is a string.

```
Print 123 & "3"      REM 1233 (String)
Print "123" & 3      REM 1233 (String)
Print 123 & 3        REM Use at least one string or it will not work!
```

Mixing string manipulations and numerical manipulations may lead to confusing results, especially because string concatenation with the & operator has lower precedence than the + operator.

```
Print 123 + "3" & 4 '1264 Do addition then convert to String
Print 123 & "3" + 4 '12334 Do addition first but first operand is String
Print 123 & 3 + "4" '1237 Do addition first but first operand is Integer
```

3.8.2. Logical and bit-wise operators

Each logical operator asks a simple question and provides a True or False answer. For example, is it true that (you have money) AND (you want to purchase my book)? These types of operations are simple and are frequently used in Basic. Less frequently used, and provided for completeness to keep the computing professionals happy, are the bit-wise operators. Bit-wise operators are not difficult, but if you don't understand them, it isn't likely to affect your usage of Basic.

A logical operator is usually thought to operate on True and False values. In Basic, logical operators also perform bit-wise operations on Integer values. This means that each bit of the first operand is compared to the corresponding bit in the second operand to form the corresponding bit in the result. For example, the binary operands 01 and 10 use the 0 from 01 and the 1 from 10 to produce the first bit of the result.

The unusual thing about logical and bit-wise binary operators in Basic is that their precedence is the same. In other languages, AND typically has greater precedence than OR.

Table 17 illustrates the logical and bit-wise operators supported by OOo. True and False represent logical values, and 0 and 1 represent bit values.

Table 17. Truth table for logical and bit-wise operators.

x	y	x AND y	x OR y	x XOR y	x EQV y	x IMP y
True	True	True	True	False	True	True
True	False	False	True	True	False	False
False	True	False	True	True	False	True
False	False	False	False	False	True	True
1100	1010	1000	1110	0110	1001	1011

Internally, the logical operators cast their operands to type Long. An unexpected side effect is that a floating-point operand is converted to a Long, which might cause numerical overflow. The conversion from a floating-point number to a long integer number is done by rounding the value, not by truncating. The values chosen for True (-1) and False (0) allow this to work, but the return type with two Boolean operands is still sometimes of type Long.

Listing 39. Logical operands are of type long integer.

```
Sub LogicalOperandsAreLongs
  Dim v, b1 As Boolean, b2 As Boolean
  b1 = True : b2 = False
```

```

v = (b1 OR b2)
Print TypeName(v)    REM Long because operands are converted to Long.
Print v              REM -1   because the return type is Long.
Print (b2 OR "-1")  REM -1   because "-1" is converted to a Long.
End Sub

```

For some logical expressions, not all operands need to be evaluated. For example, the expression (False AND True) is known to be False by looking at the first operand and the operator AND. This is known as “short-circuit evaluation.” Sadly, this isn’t available in Basic; instead, all operands are evaluated.

TIP Basic does not support short-circuit evaluation, so $(x < 0 \text{ AND } y/x > 3)$ causes a division-by-zero error when x is zero.

The bit-wise operators are all illustrated the same way. Two arrays are filled with Boolean values and two integers are given an Integer value.

```

xi% = 12 : yi% = 10
x() = Array(True, True, False, False)
y() = Array(True, False, True, False)

```

The decimal number 12 is represented in base 2 as 1100, which corresponds to the values in $x()$. The decimal number 10 is represented in base 2 as 1010, which corresponds to the value in $y()$. The operator is then applied to “ $x(0)$ op $y(0)$ ”, “ $x(1)$ op $y(1)$ ”, “ $x(2)$ op $y(2)$ ”, “ $x(3)$ op $y(3)$ ”, and “ x_i op y_i ”. The result is displayed in a message box. The integers are displayed as base 2 to emphasize that a bit-wise operation is performed. Listing 40 demonstrates how an integer is converted to a stream of bits. This uses many techniques that are discussed later in this chapter.

Listing 40. *Convert an integer to binary.*

```

Sub TestIntoToBinary
    Dim s$
    Dim n%
    Dim x%
    x = InputBox("Enter an integer")
    If x <> 0 Then
        n = Log(Abs(x)) / Log(2) + 1
        If (x < 0) Then
            n = n + 4
        End If
    Else
        n = 1
    End If
    print "s = " & IntToBinaryString(x, n)
End Sub

REM Convert an Integer value to a string of bits
REM x is the integer to convert
REM n is the number of bits to convert
REM This would be easier if I could shift out the lowest while
REM retaining the sign bit of the number, but I cannot.
REM I emulate this by dividing by two, but this fails for negative

```

```

REM numbers. To avoid this problem, if the number is negative
REM I flip all of the bits, which makes it a positive number and
REM then I build an inverted answer
Function IntToBinaryString(ByVal x%, ByVal n%) As String
    Dim b0$ 'Bit 0 (the right most bit) for positive numbers.
    Dim b1$ 'Bit 1 for positive numbers.
    If (x >= 0) Then 'Not negative so the algorithm will work.
        b1 = "1" : b0 = "0" 'Use the standard bit values.
    Else 'A negative number so
        x = NOT x 'flip all of the bits.
        b1 = "0" : b0 = "1" 'Invert bit 0 and invert bit 1.
    End If
    Dim s$ 'Accumulate bits as a string in s$.
    Do While n > 0 'n is the number of bits to return.
        If (x AND 1) = 1 Then 'AND with 1 to find bit 0 (the right most bit).
            s = b1$ & s 'Bit 1 set so add a 1 (if x was negative add 0).
        Else
            s = b0$ & s 'Bit 1 clear so add a 0 (if x was negative add 1).
        End If
        x = x \ 2 'Integer division by 2.
        n = n - 1 'Decrement n by 1, just finished a bit.
    Loop 'Back to the top of the While.
    IntToBinaryString = s 'Assign the return value to the Function.
End Function

```

AND

Perform a logical AND operation on Boolean values, and a bit-wise AND on numerical values. Consider the phrase, “You can go to the movie if you have money AND if you have transportation.” Both conditions must be true before you are able to go to the movie. If both operands are True, then the result is True; otherwise the result is False.

Listing 41. Operator AND.

```

Sub ExampleOpAND
    Dim s$, x(), y(), i%, xi%, yi%
    xi% = 12 : yi% = 10
    x() = Array(True, True, False, False)
    y() = Array(True, False, True, False)
    For i = LBound(x()) To UBound(x())
        s = s & x(i) & " AND " & y(i) & " = " & CBool(x(i) AND y(i)) & CHR$(10)
    Next
    s = s & IntToBinaryString(xi, 4) & " AND " & IntToBinaryString(yi, 4) & _
        " = " & IntToBinaryString(xi AND yi, 4) & CHR$(10)
    MsgBox s, 0, "Operator AND example"
End Sub

```

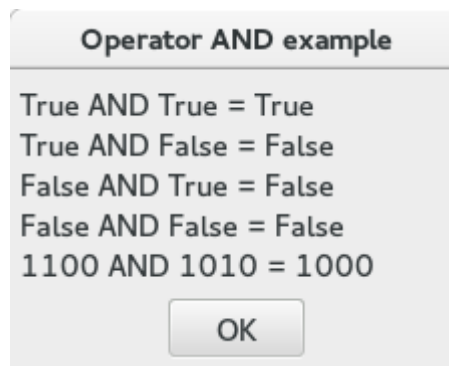


Figure 31. Using the AND operator.

OR

Perform a logical OR operation on Boolean values, and a bit-wise OR on numerical values. Consider the phrase, “You can purchase that if you have cash OR your friend has cash.” It does not matter who has cash. If either operand is True, then the result is True; otherwise the result is False.

Listing 42. Operator OR.

```
Sub ExampleOpOR
    Dim s$, x(), y(), i%, xi%, yi%
    xi% = 12 : yi% = 10
    x() = Array(True, True, False, False)
    y() = Array(True, False, True, False)
    For i = LBound(x()) To UBound(x())
        s = s & x(i) & " OR " & y(i) & " = " & CBool(x(i) OR y(i)) & CHR$(10)
    Next
    s = s & IntToBinaryString(xi, 4) & " OR " & IntToBinaryString(yi, 4) & _
        " = " & IntToBinaryString(xi OR yi, 4) & CHR$(10)
    MsgBox s, 0, "Operator OR example"
End Sub
```

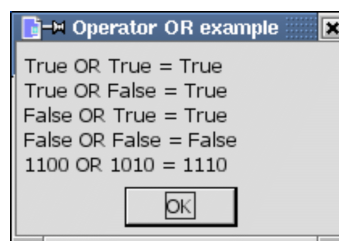


Figure 32. Using the OR operator.

XOR

The XOR operator is called “exclusive or”; this is a question of non-equivalence. The result is True if the operands have different values. The result is False if both operands have the same value. A logical XOR operation is performed on Boolean values, and a bit-wise XOR is performed on numerical values.

Listing 43. Operator XOR.

```
Sub ExampleOpXOR
    Dim s$, x(), y(), i%, xi%, yi%
    xi% = 12 : yi% = 10
    x() = Array(True, True, False, False)
    y() = Array(True, False, True, False)
    For i = LBound(x()) To UBound(x())
        s = s & x(i) & " XOR " & y(i) & " = " & CBool(x(i) XOR y(i)) & CHR$(10)
    Next
    s = s & IntToBinaryString(xi, 4) & " XOR " & IntToBinaryString(yi, 4) & _
        " = " & IntToBinaryString(xi XOR yi, 4) & CHR$(10)
    MsgBox s, 0, "Operator XOR example"
End Sub
```

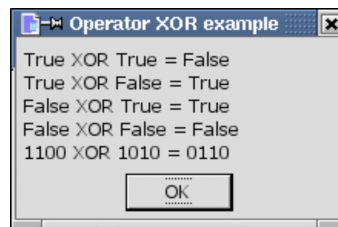


Figure 33. Using the XOR operator.

EQV

The EQV operator is a question of equivalence: Are the two operands the same? A logical EQV operation is performed for Boolean values, and a bit-wise EQV on numbers. If both operands have the same value, the result is True. If the operands don't have the same value, the result is False.

Listing 44. Operator EQV.

```
Sub ExampleOpEQV
    Dim s$, x(), y(), i%, xi%, yi%
    xi% = 12 : yi% = 10
    x() = Array(True, True, False, False)
    y() = Array(True, False, True, False)
    For i = LBound(x()) To UBound(x())
        s = s & x(i) & " EQV " & y(i) & " = " & CBool(x(i) EQV y(i)) & CHR$(10)
    Next
    s = s & IntToBinaryString(xi, 4) & " EQV " & IntToBinaryString(yi, 4) & _
        " = " & IntToBinaryString(xi EQV yi, 4) & CHR$(10)
    MsgBox s, 0, "Operator EQV example"
End Sub
```

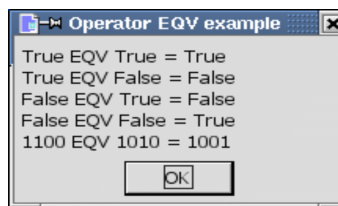


Figure 34. Using the EQV operator.

IMP

The IMP operator performs a logical implication. A logical IMP operation is performed on Boolean values, and a bit-wise IMP on numbers. As the name implies, “x IMP y” asks if the statement that “x implies y” is a true statement. To help understand logical implication, define x and y as follows:

```
x = The sky is cloudy
y = The sun is not visible
If x Then y
```

If both x and y are true — the sky is cloudy and the sun is not visible — the statement can be considered true. This statement makes no claim about y if x is not true. In other words, if the sky is not cloudy, this statement does not imply that the sun is, or is not, visible. For example, it might be a clear night, or (like a good computer geek) you might be inside a room without any windows. This explains why the entire statement is always considered valid when x is false. Finally, if x is true and y is not, the entire statement is considered false. If the sky is cloudy, and the sun is visible, the statement cannot possibly be correct; a cloudy day could not imply that the sun is visible.

Listing 45. Operator IMP.

```
Sub ExampleOpIMP
    Dim s$, x(), y(), i%, xi%, yi%
    xi% = 12 : yi% = 10
    x() = Array(True, True, False, False)
    y() = Array(True, False, True, False)
    For i = LBound(x()) To UBound(x())
        s = s & x(i) & " IMP " & y(i) & " = " & CBool(x(i) IMP y(i)) & CHR$(10)
    Next
    s = s & IntToBinaryString(xi, 4) & " IMP " & IntToBinaryString(yi, 4) & _
        " = " & IntToBinaryString(xi IMP yi, 4) & CHR$(10)
    MsgBox s, 0, "Operator IMP example"
End Sub
```

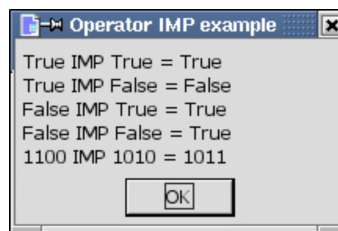


Figure 35. Using the IMP operator.

NOT

The NOT operator performs a logical NOT operation on Boolean values, and a bit-wise NOT on numerical values. This means that “Not True” is False and “Not False” is True. For bit-wise operations, a 1 becomes a 0 and a 0 becomes a 1.

```
Print NOT True    REM 0, which is False
Print NOT False   REM -1, which is True
```

```
Print NOT 2      REM -3, which took the bits 0010 to 1101
```

3.8.3. Comparison operators

The comparison operators work with numerical, Date, Boolean, and String data types.

```
Print 2 = 8/4 AND 0 < 1/3 AND 2 > 1    '-1=True
Print 4 <= 4.0 AND 1 >= 0 AND 1 <> 0    '-1=True
```

String comparisons are based on their internal representations as numbers and are case sensitive. The letter “A” is less than the letter “B”. The uppercase characters are less than the lowercase letters.

```
Dim a$, b$, c$
a$ = "A" : b$ = "B" : c$ = "B"
Print a$ < b$    'True
Print b$ = c$    'True
Print c$ <= a$    'False
```

Strange problems occur when all of the operands are string constants. If at least one operand is a variable, the expected results are achieved. This is likely related to how the operands are recognized and converted for use.

```
Print "A" < "B"      '0=False, this is not correct
Print "B" < "A"      '-1=True, this is not correct
Print 3 = "3"        'False, but this changes if a variable is used
```

When variables are used rather than string constants, the numerical values are converted to string types for the comparison.

```
Dim a$, i%, t$
a$ = "A" : t$ = "3" : i% = 3
Print a$ < "B"      'True, String compare
Print "B" < a$      'False, String compare
Print i% = "3"      'True, String compare
Print i% = "+3"     'False, String compare
Print 3 = t$        'True, String compare
Print i% < "2"      'False, String compare
Print i% > "22"     'True, String compare
```

TIP When comparing operands of different types, especially when mixing numeric and string types, it is safer to explicitly perform a type conversion. Either convert the string to a number, or the number to a string. The functions to do this are discussed later.

OOo recognizes the Visual Basic statement Option Compare { Binary | Text}, but as of OOo version 3.20, the statement still does nothing. The current behavior is a binary comparison of strings.

3.9. Flow control

Flow control is about deciding which line of code runs next. Calling a subroutine or function is a simple form of unconditional flow control. More complicated flow control involves branching and looping. Flow control allows macros to have complicated behavior that changes based on the current data.

Branching statements cause the program flow to change. Calling a subroutine or function is an unconditional branch. Basic supports conditional branching statements such as “if x, then do y”. Looping statements cause the program to repeat sections of code. Looping statements allow for a section to be repeated a specific number of times or until a specific “exit” condition has been achieved.

3.9.1. Define a label as a jump target

Some flow control statements, such as GoSub, GoTo, and On Error, require a label to mark a point in the code. Label names are subject to the same rules as variable names. Label names are immediately followed by a colon. Remember that a colon is also used as a statement separator that allows multiple statements to occupy the same line. Space between the label name and the colon causes the colon to be used as a statement separator, which means the label won’t be defined. The following lines all represent valid Basic code.

```
<statements>
i% = 5 : z = q + 4.77

MyCoolLabel:
<more statements>

JumpTarget: <more statements> REM no space between label and colon
```

TIP	Inserting a space between a label and the colon causes the colon to be used as a statement separator, and the label is not defined.

3.9.2. GoSub

The GoSub statement causes execution to jump to a defined label in the current routine. It isn’t possible to jump outside of the current routine. When the Return statement is reached, execution continues from the point of the original call. A Return statement with no previous GoSub produces a run-time error. In other words, Return is not a substitute for Exit Sub or Exit Function. It is generally assumed that functions and subroutines produce more understandable code than GoSub and GoTo.

TIP	GoSub is a persistent remnant from old BASIC dialects, retained for compatibility. GoSub is strongly discouraged because it tends to produce unreadable code. Use a subroutine or function instead. In fact, Visual Basic .NET no longer supports the GoSub keyword.

Listing 46. Example GoSub.

```
Sub ExampleGoSub
    Dim i As Integer
    GoSub Line2      REM Jump to line 2 then return, i is 1
    GoSub [Line 1]   REM Jump to line 1 then return, i is 2
    MsgBox "i = " + i, 0, "GoSub Example" REM i is now 2
    Exit Sub         REM Leave the current subroutine.
```

```

[Line 1]:      REM this label has a space in it
      i = i + 1  REM Add one to i
      Return    REM return to the calling location
Line2:        REM this label is more typical, no spaces
      i = 1     REM Set i to 1
      Return    REM return to the calling location
End Sub

```

3.9.3. GoTo

The GoTo statement causes execution to jump to a defined label in the current routine. It isn't possible to jump outside of the current routine. Unlike the GoSub statement, the GoTo statement doesn't know from where it came. GoTo is a persistent remnant from old BASIC dialects, retained for compatibility. GoTo is strongly discouraged because it tends to produce unreadable code. Use a subroutine or function instead.

Listing 47. Example GoTo.

```

Sub ExampleGoTo
    Dim i As Integer
    GoTo Line2      REM Okay, this looks easy enough
Line1:             REM but I am becoming confused
    i = i + 1       REM I wish that GoTo was not used
    GoTo TheEnd     REM This is crazy, makes me think of spaghetti,
Line2:             REM Tangled strands going in and out; spaghetti code.
    i = 1           REM If you have to do it, you probably
    GoTo Line1      REM did something poorly.
TheEnd:           REM Do not use GoTo.
    MsgBox "i = " + i, 0, "GoTo Example"
End Sub

```

3.9.4. On GoTo and On GoSub

These statements cause the execution to branch to a label based on a numeric expression N. If N is zero, no branching occurs. The numeric expression N must be in the range of 0 through 255. This is sometimes called a “computed goto,” because a computation is used to direct the program flow. It isn't possible to jump outside of the current subroutine.

```

Syntax: On N GoSub Label1[, Label2[, Label3[,...]]]
Syntax: On N GoTo  Label1[, Label2[, Label3[,...]]]

```

To reiterate how this works, if N = 1 then branch to Label 1, if N = 2 then branch to Label 2... If N is less than 1 or if N is greater than the number of labels, then the branch is not taken; it is simply ignored.

Listing 48. Example On GoTo.

```

Sub ExampleOnGoTo
    Dim i As Integer
    Dim s As String
    i = 1
    On i+1 GoSub Sub1, Sub2
    s = s & Chr(13)
    On i GoTo Line1, Line2

```

```

    REM The exit causes us to exit if we do not continue execution
    Exit Sub
Sub1:
    s = s & "In Sub 1" : Return
Sub2:
    s = s & "In Sub 2" : Return
Line1:
    s = s & "At Label 1" : GoTo TheEnd
Line2:
    s = s & "At Label 2"
TheEnd:
    MsgBox s, 0, "On GoTo Example"
End Sub

```

3.9.5. If Then Else

The If construct is used to execute a block of code based on an expression. Although you can use GoTo or GoSub to jump out of an If block, you cannot jump into an If block. The simplest If statement has the following form:

```
If Condition Then Statement
```

The condition can be any expression that either evaluates to — or is convertible to — True or False. Use a slightly more complicated version to control more than a single statement.

```

If Condition Then
    Statementblock
[ElseIf Condition Then]
    Statementblock
[Else]
    Statementblock
End If

```

If the first condition evaluates to True, the first block runs. The ElseIf statement allows multiple If statements to be tested in sequence. The statement block for the first true condition runs. The Else statement block runs if no other condition evaluates to True.

Listing 49. Example If.

```

Sub ExampleIf
    Dim i%
    i% = 4
    If i = 4 Then Print "i is four"
    If i <> 3 Then
        Print "i is not three"
    End If
    If i < 1 Then
        Print "i is less than 1"
    elseif i = 1 Then
        Print "i is 1"
    elseif i = 2 Then
        Print "i is 2"
    else
        Print "i is greater than 2"
    End If
End Sub

```

```
End If
End Sub
```

If statements can be nested.

```
If i <> 3 THEN
    If k = 4 Then Print "k is four"
    If j = 7 Then
        Print "j is seven"
    End If
End If
End If
```

3.9.6. IIf

The IIf (“Immediate If”) function returns one of two values based on a conditional expression.

Syntax: `object = IIf (Condition, TrueExpression, FalseExpression)`

This is very similar to the following code:

```
If Condition Then
    object = TrueExpression
Else
    object = FalseExpression
End If
```

This works as a great single-line If-Then-Else statement.

```
max_age = IIf(johns_age > bills_age, johns_age, bills_age)
```

3.9.7. Choose

The Choose statement selects from a list of values based on an index.

Syntax: `obj = Choose (expression, Select_1[, Select_2, ... [,Select_n]])`

The Choose statement returns a null if the expression is less than 1 or greater than the number of selection arguments. Choose returns “select_1” if the expression evaluates to 1, and “select_2” if the expression evaluates to 2. The result is similar to storing the selections in an array with a lower bound of 1 and then indexing into the array.

Listing 50. *A division-by-zero error occurs even though 1/(i-2) should be returned.*

```
i% = 3
Print Choose (i%, 1/(i+1), 1/(i-1), 1/(i-2), 1/(i-3))
```

Selections can be expressions and they can contain function calls. Every function is called and every expression is evaluated in the argument list for the call to the Choose statement. The code in Listing 50 causes a division-by-zero error because every argument is evaluated, not just the argument that will be returned. Listing 51 calls the functions Choose1, Choose2, and Choose3.

Listing 51. *Example Choose statement.*

```
Sub ExampleChoose
    Print Choose(2, "One", "Two", "Three")           'Two
    Print Choose(2, Choose1(), Choose2(), Choose3()) 'Two
End Sub
```

```

Function Choose1$()
    Print "I am in Choose1"
    Choose1 = "One"
End Function
Function Choose2$()
    Print "I am in Choose2"
    Choose2 = "Two"
End Function
Function Choose3$()
    Print "I am in Choose3"
    Choose3 = "Three"
End Function

```

TIP All arguments in a Choose statement are evaluated. If functions are used in the arguments for Choose, they are all called.

3.9.8. Select Case

The Select Case statement is similar to an If statement with multiple Else If blocks. A single-condition expression is specified and this is compared against multiple values for a match as follows:

```

Select Case condition_expression
    Case case_expression1
        StatementBlock1
    Case case_expression2
        StatementBlock2
    Case Else
        StatementBlock3
End Select

```

The condition_expression is compared in each Case statement. The first statement block to match is executed. The optional Case Else block runs if no condition matches. It is not an error if nothing matches and no Case Else block is present.

Case expressions

The conditional expression is evaluated once and then it is compared to each case expression until a match is found. A case expression is usually a constant, such as “Case 4” or “Case “hello””.

```

Select Case 2
    Case 1
        Print "One"
    Case 3
        Print "Three"
End Select

```

You can specify multiple values by separating them with commas: “Case 3, 5, 7”. The keyword **To** checks a range of values — for example, “Case 5 To 10”. Open-ended ranges are checked as “Case < 10” or as “Case IS < 10”.

TIP

The Case IS statement is different than the IS operator that decides if two objects are the same.

Every Case statement written “Case op expression” is shorthand for writing “Case IS op expression”. The form “Case expression” is shorthand for “Case IS = expression”. For example, “Case >= 5” is equivalent to “Case IS >= 5”, and “Case 1+3” is equivalent to “Case IS = 1+3”.

```
Select Case i
  Case 1, 3, 5
    Print "i is one, three, or five"
  Case 6 To 10
    Print "i is a value from 6 through 10"
  Case < -10
    Print "i is less than -10"
  Case IS > 10
    Print "i is greater than 10"
  Case Else
    Print "No idea what i is"
End Select
```

A Case statement can contain a list of expressions separated by commas. Each expression can include an open-ended range. Each expression can use the statement Case IS (see Listing 52).

Listing 52. The keyword IS is optional.

```
Select Case i%
  Case 6, Is = 7, Is = 8, Is > 15, Is < 0
    Print "" & i & " matched"
  Case Else
    Print "" & i & " is out of range"
End Select
```

If Case statements are easy, why are they frequently incorrect?

I frequently see incorrect examples of Case statements. It’s instructive to see what is repeatedly done incorrectly. Consider the simple examples in Table 18. The last examples are written correctly in Table 20.

Table 18. Case statements are frequently written incorrectly.

Example	Valid	Description
Select Case i Case 2	Correct	The Case expression 2 is evaluated as two. Two is compared to i.
Select Case i Case Is = 2	Correct	The Case expression 2 is evaluated as two. Two is compared to i.
Select Case i Case Is > 7	Correct	The expression 7 is evaluated as seven. Seven is compared to i.
Select Case i Case 4, 7, 9	Correct	The conditional expression i is compared individually to 4, 7, and 9.
Select Case x Case 1.3 TO 5.7	Correct	You can specify a range and use floating-point numbers.
Select Case i Case i = 2	Incorrect	The Case expression (i=2) is evaluated as True or False. True or False is compared to i. This is reduced to “IS = (i=2)”.

Example	Valid	Description
Select Case i Case i<2 OR i>9	Incorrect	The Case expression (i<2 OR 9<i) is evaluated as True or False. True or False is compared to i. This is reduced to “IS = (i<2 OR i>9)”.
Select Case i% Case i%>2 AND i%<10	Incorrect	The Case expression (i>2 AND i < 10) is evaluated as True or False. True or False is compared to i. This is reduced to “IS = (i>2 AND i<10)”.
Select Case i% Case IS>8 And i<11	Incorrect	Again, True and False are compared to i. This is reduced to “IS > (8 AND i<11)”. The precedence rules cause this to be reduced to “IS > (8 AND (i<11))”. This is usually not what’s intended.
Select Case i% Case IS>8 And IS<11	Incorrect	Compile error. The keyword IS must immediately follow Case.

I have seen examples with incorrect examples such as “Case i > 2 AND i < 10”. This fails. Don’t believe it even though you see it in print. Understand why this fails and you have mastered Case statements.

The next to the last incorrect example in Table 18 demonstrates the most common error that I see with Case expressions. Listing 53 considers the case when i is less than 11, and the case when i is greater than or equal to 11. To put it simply, IS>8 AND i<11 has the Case statement comparing the value of i to the result of a Boolean expression, which can be only 0 or -1. The big difficulty with Case statements is that they look like If statements, which are looking for a True or False, but Case statements are looking for a particular value against which to match the conditional, and 0 or -1 is not helpful.

Consider the second case in Listing 53, i >= 11. The operator < has higher precedence than the operator AND, so it is evaluated first. The expression i<11 evaluates to False (because I assumed that i>=11). False is internally represented as 0. Zero has no bits set, so 8 AND 0 evaluates to zero. For values of i greater than or equal to 11, the entire expression is equivalent to “IS > 0”. In other words, for i = 45, this Case statement is incorrectly accepted.

A similar argument for values of i less than 11, left as an exercise for the reader, demonstrates that the Case statement is equivalent to “Case IS > 8”. Therefore, values of i less than 11 are correctly evaluated, but values of i greater than or equal to 11 are not.

Listing 53. “Case IS > 8 AND i < 11” reduces incorrectly.

```
IS > (8 AND i < 11) => IS > (8 AND -1) => IS > 8 'Assume i < 11 is correct.
IS > (8 AND i < 11) => IS > (8 AND 0) => IS > 0 'Assume i >= 11 is unwanted.
```

Writing correct Case expressions

After you learn a few simple examples, it’s easy to write correct Case expressions. Table 19 abstractly enumerates the varieties, and Listing 57 concretely demonstrates the varieties.

Table 19. Simple Case varieties.

Example	Description
Case IS operator expression	This is both the simplest case and the most difficult case. If the expression is a constant, it is easy to understand. If the expression is more complicated,

Example	Description
	the only difficult part is building the expression.
Case expression	This is a reduction of “Case IS operator expression” when the operator checks for equality.
Case expression TO expression	Check an inclusive range. This is usually done correctly.
Case expression, expression,	Each expression is compared. This is usually done correctly.

For the difficult cases, it suffices to produce an expression that evaluates to the Case condition expression if it is correct, and anything else if it is not. In other words, for Select Case 4, the expression must evaluate to 4 for the statement block to run.

Listing 54. *If x is a String value, this will work for any Boolean expression.*

```
Select Case x
    Case IIF(Boolean expression, x, x&"1") ' Assumes that x is a string
```

In Listing 54, x is returned if the Boolean expression is True. The expression x=x is True, so the Case statement passes. If the Boolean expression is False, x&"1" is returned. This is not the same string as x, so the Case statement will not pass. A similar method is used for numerical values.

Listing 55. *If x is a numerical value, this will work for any Boolean expression.*

```
Select Case x
    Case IIF(Boolean expression, x, x+1) ' Assumes that x is numeric
```

In Listing 55, x is returned if the Boolean expression is True. The expression x=x is True, so the Case statement passes. If the Boolean expression is False, x+1 is returned. For numerical values, x=x+1 is not True, so the Case statement will not pass. There is a possibility of numerical overflow, but in general this works. A brilliant and more elegant solution for numerical values is provided by Bernard Marcelly, a member of the OOo French translation project.

```
Case x XOR NOT (Boolean Expression)
```

This assumes that the Boolean expression returns True (-1) if it should pass and False (0) if it should not.

Listing 56. *This code uses XOR and NOT in a Case statement.*

```
x XOR NOT(True)   = x XOR NOT(-1) = x XOR 0 = x
x XOR NOT(False)  = x XOR NOT( 0) = x XOR -1 <> x
```

After my initial confusion, I realized how brilliant this really is. There are no problems with overflow and it works for all Integer values of x. Do not simplify this to the incorrect reduction “x AND (Boolean expression)” because it fails if x is 0.

Listing 57. *Select Case example.*

```
Sub ExampleSelectCase
    Dim i%
    i = Int((20 * Rnd) - 2) 'Rnd generates a random number between zero and one
    Select Case i%
        Case 0
            Print "" & i & " is Zero"
```

```

Case 1 To 5
    Print "" & i & " is a number from 1 through 5"
Case 6, 7, 8
    Print "" & i & " is the number 6, 7, or 8"
Case IIf(i > 8 And i < 11, i, i+1)
    Print "" & i & " is greater than 8 and less than 11"
Case i% XOR NOT(i% > 10 AND i% < 16)
    Print "" & i & " is greater than 10 and less than 16"
Case Else
    Print "" & i & " is out of range 0 to 15"
End Select
End Sub

```

ExampleSelectCase in Listing 57 generates a random integer from -2 through 18 each time it runs. Run this repeatedly to see each Case statement used. Each of the cases could have used the IIF construct.

Now that I've explained the different methods to deal with ranges, it's time to reevaluate the incorrect cases in Table 18. The solutions in Table 20 are not the only possible solutions, but they use some of the solutions presented.

Table 20. Incorrect examples from Table 18 — now corrected.

Incorrect	Correct	Description
Select Case i Case i = 2	Select Case i Case 2	The variable i is compared to 2.
Select Case i Case i = 2	Select Case i Case IS = 2	The variable i is compared to 2.
Select Case i Case i<2 OR i>9	Select Case i Case IIf(i<2 OR i>9, i, i+1)	This works even if i is not an integer.
Select Case i% Case i%>2 AND i%<10	Select Case i% Case 3 TO 9	i% is an integer so the range is from 3 through 9.
Select Case i% Case IS>8 And i<11	Select Case i% Case i XOR NOT(i>8 AND i< 11)	This works because i% is an integer.

3.9.9. While ... Wend

Use the While ...Wend statement to repeat a block of statements while a condition is true. This construct has limitations that do not exist with the Do While ... Loop construct and offers no particular benefits. The While ... Wend statement does not support an Exit statement. You cannot use GoTo to exit a While ... Wend statement.

```

While Condition
    StatementBlock
Wend

```

Visual Basic .NET does not support the keyword Wend. This is another reason to use the Do While ... Loop instead.

3.9.10. Do ... Loop

The Loop construct has different forms and is used to continue executing a block of code while, or until, a condition is true. The most common form checks the condition before the loop starts, and repeatedly executes a block of code as long as the condition is true. If the initial condition is false, the loop is never executed.

```
Do While condition
    Block
[Exit Do]
Block
Loop
```

A similar, but much less common form, repeatedly executes the code as long as the condition is false. In other words, the code is executed until the condition becomes true. If the condition evaluates to true immediately, the loop never runs.

```
Do Until condition
    Block
[Exit Do]
Block
Loop
```

You can place the check at the end of the loop, in which case the block of code is executed at least once. With the following construct, the loop runs at least once and then repeatedly runs as long as the condition is true:

```
Do
    Block
[Exit Do]
Block
Loop While condition
```

To execute the loop at least once and then continue as long as the condition is false, use the following construct:

```
Do
    Block
[Exit Do]
Block
Loop Until condition
```

Exit the Do Loop

The Exit Do statement causes an immediate exit from the loop. The Exit Do statement is valid only within a Do ... Loop. Program execution continues with the statement that follows the innermost Loop statement. The subroutine ExampleDo in Listing 58 demonstrates a Do While Loop by searching an array for a number.

Listing 58. Do Loop example.

```
Sub ExampleDo
    Dim a(), i%, x%
    a() = Array(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30)
```

```

x = Int(32 * Rnd)          REM random integer between 0 and 32
i = LBound(a())           REM i is the lower bound of the array.
Do While a(i) <> x         REM while a(i) is not equal to x
    i = i + 1              REM Increment i
    If i > UBound(a()) Then Exit Do REM If i is too large, then exit
Loop                      REM Loop back to the Do While
If i <= UBound(a()) Then  REM If i is not too large then found x
    MsgBox "Found " & x & " at location " & i, 0, "Example Do"
Else
    MsgBox "Could not find " & x & " in the array", 0, "Example Do"
End If
End Sub

```

Which Do Loop should I use?

Basic supports four variants of the Do Loop construct. Each variant has a particular purpose and time for use. The most common problem is that the loop runs one time too few or one time too many because the conditional expression was incorrectly placed at the wrong location.

When deciding where to place the conditional expression of a Do Loop, ask yourself this question: “Must the loop always run at least once?” If the answer is no, the conditional expression must be at the top. This will prevent the code in the loop from running if the conditional expression fails. Consider, for example, printing all of the elements in an array of unknown size. The array might contain no elements at all, in which case the code in the loop shouldn’t run (see Table 21).

Table 21. While and Until loops are very similar.

Do While	Do Until
<pre> i% = LBound(a()) Do While i% <= UBound(a()) Print a(i%) i% = i% + 1 Loop </pre>	<pre> i% = LBound(a()) Do Until i% > UBound(a()) Print a(i%) i% = i% + 1 Loop </pre>

In each case in Table 21, if the array is empty, the loop never runs. Before the condition is evaluated, i% is set to the lower bound of the array. In each case, the loop continues to run while i % is not larger than the upper bound.

Consider the difference between a While loop and an Until loop with a simple example. While the car has gas, you may drive it. Until the car does not have gas, you may drive it. The primary difference between the While and the Until is the word NOT. Tending more toward Basic, I can write “Until NOT (the car has gas).” The choice between While and Until is usually based on which one you can write without the NOT.

If the loop should run at least once, move the conditional expression to the end of the Do Loop. Consider requesting user input until a valid value has been entered. The most natural choice is to place the conditional expression at the end.

```
Dim s$, x As Double
```

```

Do
    s$ = InputBox("Enter a number from 1 through 5")
    x = Cdbl(s$)      'Convert the string to a Double
Loop Until x >= 1 AND x <= 5

```

The loop must run at least once so that at least one number is entered. The loop repeats until a valid value is entered. As an exercise, consider how to write this as a While loop.

3.9.11. For ... Next

The For ... Next statement repeats a block of statements a specified number of times.

```

For counter=start To end [Step stepValue]
    statement block1
[Exit For]
    statement block2
Next [counter]

```

The numeric “counter” is initially assigned the “start” value. When the program reaches the Next statement, the counter is incremented by the “step” value, or incremented by one if a “step” value is not specified. If the “counter” is still less than or equal to the “end” value, the statement blocks run. An equivalent Do While Loop follows:

```

counter = start
Do While counter <= end
    statement block1
[Exit Do]
    statement block2
    counter = counter + step
Loop

```

The “counter” is optional on the “Next” statement, and it automatically refers to the most recent “For” statement.

```

For i = 1 To 4 Step 2
    Print i ' Prints 1 then 3
Next i     ' The i in this statement is optional.

```

The Exit For statement leaves the For statement immediately. The most recent For statement is exited. Listing 59 demonstrates this with a sorting routine. An array is filled with random integers, and then sorted using two nested loops. This technique is called a “modified bubble sort.”

First, iOuter is set to the last number in the array. The inner loop, using iInner, compares every number to the one after it. If the first number is larger than the second number, the two numbers are swapped. The second number is then compared to the third number. After the inner loop is finished, the largest number is guaranteed to be at the last position in the array.

Next, iOuter is decremented by 1. The inner loop this time ignores the last number in the array, and compares every number to the one after it. At the end of the inner loop, the second-highest number is second from the end.

With each iteration, one more number is moved into position. If no numbers are exchanged, the list is sorted.

Listing 59. Modified bubble sort.

```
Sub ExampleForNextSort
    Dim iEntry(10) As Integer
    Dim iOuter As Integer, iInner As Integer, iTemp As Integer
    Dim bSomethingChanged As Boolean

    ' Fill the array with integers between -10 and 10
    For iOuter = LBound(iEntry()) To UBound(iEntry())
        iEntry(iOuter) = Int((20 * Rnd) - 10)
    Next iOuter

    ' iOuter runs from the highest item to the lowest
    For iOuter = UBound(iEntry()) To LBound(iEntry()) Step -1

        'Assume that the array is already sorted and see if this is incorrect
        bSomethingChanged = False
        For iInner = LBound(iEntry()) To iOuter-1
            If iEntry(iInner) > iEntry(iInner+1) Then
                iTemp = iEntry(iInner)
                iEntry(iInner) = iEntry(iInner+1)
                iEntry(iInner+1) = iTemp
                bSomethingChanged = True
            End If
        Next iInner
        'If the array is already sorted then stop looping!
        If Not bSomethingChanged Then Exit For
    Next iOuter
    Dim s$
    For iOuter = LBound(iEntry()) To UBound(iEntry())
        s = s & iOuter & " : " & iEntry(iOuter) & CHR$(10)
    Next iOuter
    MsgBox s, 0, "Sorted Array"
End Sub
```

3.9.12. Exit Sub and Exit Function

The Exit Sub statement exits a subroutine, and the Exit Function statement exits a function. The routine is exited immediately. The macro continues running at the statement following the statement that called the current routine. These statements work only to exit the currently running routine, and they are valid only in their respective types. For example, you can't use Exit Sub in a function.

3.10. Error handling using On Error

Errors are usually placed into three “categories” — compile time, run time, and logic. Compile-time errors are typically syntax errors such as missing double quotation marks that prevent your macro from compiling. Compile-time errors are the easiest to deal with because they are found

immediately and the IDE shows you which line caused the problem. Run-time errors compile properly but cause an error when the macro runs. For example, dividing by a variable that at some point evaluates to zero will cause a run-time error. The third type, logic errors, are mistakes in the business logic of the program: They compile and run okay, but generate the wrong answers. They're the worst kind because you have to find them yourself — the computer won't help you at all. This section is about run-time errors: how to deal with them and how to correct them.

An error handler is a piece of code that runs when an error occurs. The default error handler displays an error message and stops the macro. Basic provides a mechanism to modify this behavior (see Table 22). The first form, `On Error Resume Next`, tells LO to ignore all errors: No matter what happens, keep running and pretend everything is fine. The second form, `On Error GoTo 0`, tells OOO to stop using the current error handler. Ignoring error handler scope issues, as explained later, think of `On Error GoTo 0` as restoring the default method of handling errors: Stop running and display an error message. The final form, `On Error GoTo LabelName`, allows you to define code to handle errors the way that you want. This is called “setting up an error handler.”

Table 22. *Supported On Error forms.*

Form	Usage
<code>On Error Resume Next</code>	Ignore errors and continue running at the next line in the macro.
<code>On Error GoTo 0</code>	Cancel the current error handler.
<code>On Error GoTo LabelName</code>	Transfer control to the specified label.

When an error occurs, the code that was running stops running, and control is transferred to the current error handler. The error handlers use the functions in Table 23 to determine the cause and location of the error. Visual Basic uses an Error object and does not support the functions in Table 23.

Table 23. *Error-related variables and functions.*

Function	Use
<code>CVErr</code>	Convert an expression to an error object.
<code>Erl</code>	Integer line number of the last error.
<code>Err</code>	Integer error number of the last error.
<code>Error</code>	Error message of the last error.

All error handlers must be declared in a routine and are local to the containing subroutine or function. When an error occurs, Basic starts working backward through the call stack until it finds an error handler. If it doesn't find an error handler, it uses the default handler. The default handler prints an error message and halts the program. The error information, such as `Erl`, indicates the line number in the current routine that caused the error. For example, if the current routine calls the function `b()` at line 34 and an error occurs in `b()`, the error is reported as occurring at line 34.

Listing 60 contains an example of this, and Figure 36 shows the call stack. Another example is shown in Listing 64.

Listing 60. Use Erl to get the line number.

```

x = x + 1           'Assume that this is line 33
Call b()           'Error in b() or something b() calls (line 34)
Exit Sub           'Leave the subroutine
ErrorHandler:       'No other handlers between here and the error
Print "error at " & Erl 'Prints line 34

```

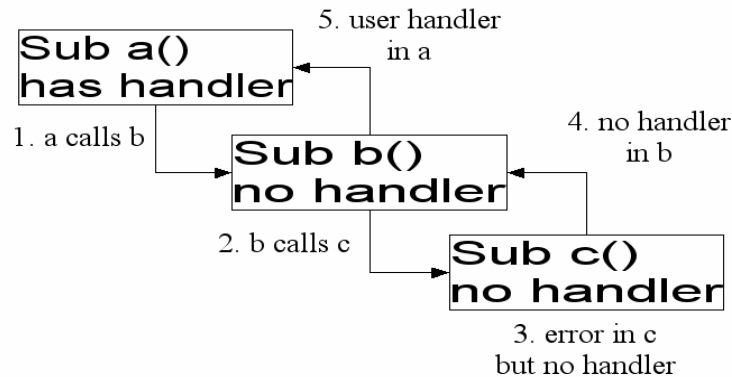


Figure 36. Walk the call stack to find a handler.

You cannot catch errors that occur in a DLL. Check the return code from the called DLL instead.

3.10.1. CVerErr

Use CVerErr (as mentioned in Table 23) to create an OOO specific internal data type that represents an error. I have never seen this done, but it can be useful for a very robust function.

CVerErr returns an internal OOO type that should be assigned to a variant variable. Why?

1. Use VarType to check the type of the returned value. A VarType of 10 means that an error object was returned (see Table 85).
2. CVerErr accepts an integer, which represents the internal error code that occurred. The internal error object casts to an integer and returns that internal error code.

The following example demonstrates CVerErr.

Listing 61. Using CVerErr

```

Sub CallNotZero
Dim xVar As Variant
Dim xInt As Integer
Dim i As Integer
Dim s As String
For i = -1 To 1
    xInt = NotZero(i)
    xVar = NotZero(i)
    s = s & "NotZero(" & i & ") = [" & xInt & "] when assigned to an Integer"

```

```

s = s & CHR$(10)
s = s & "NotZero(" & i & ") = [" & xVar & "] VarType=" & VarType(xVar)
s = s & CHR$(10)
Next
MsgBox s
End Sub

Function NotZero(x As Integer)
    If x <> 0 Then
        NotZero = x
    Else
        ' 7 is an arbitrary number meaning nothing.
        NotZero = CVErr(7)
    End If
End Function

```

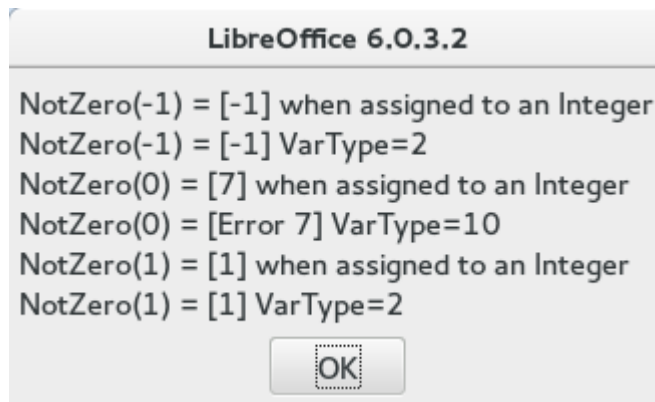


Figure 37. Return values using CVErr.

3.10.2. Ignore errors with On Error Resume Next

Error handling, in some cases, means ignoring errors. The On Error Resume statement tells Basic that if a standard error occurs, it should ignore the error and resume running the code at the next line in the macro (see Listing 62). The error information is cleared, so it isn't possible to check if an error occurred following the statement.

Listing 62. *The error is cleared by the Resume Next statement.*

```

Private zero%
sub ExampleErrorResumeNext
    On Error Resume Next
    Print 1/Zero%
    If Err <> 0 Then Print Error$ & " at line " & Erl 'Err was cleared
End Sub

```

3.10.3. Clear an error handler with On Error GoTo 0

Use the statement On Error GoTo 0 to clear an installed error handler. This is usually done inside an error handler or after the code that used one. If an error occurs inside an error handler, it isn't handled and the macro stops.

Listing 63. *The error handler is cleared with the statement On Error GoTo 0.*

```
Private zero%
sub ExampleErrorResumeNext
    On Error Resume Next
    Print 1/Zero%
    On Error GoTo 0
    ...
End Sub
```

Some versions of Visual Basic also support On Error GoTo -1, which is equivalent to On Error GoTo 0.

3.10.4. Specify your own error handler with On Error GoTo Label

To specify your own error handler, use On Error GoTo Label. To define a label in Basic, type some text on a line by itself and follow it with a colon. Line labels are no longer required to be unique; they must be unique only within each routine. This allows for consistency in naming error-handling routines rather than creating a unique name for every error handler (see Listing 64 and Figure 38). When an error occurs, execution is transferred to the label.

Listing 64. *Error Handling.*

```
Private zero%
Private error_s$
Sub ExampleJumpErrorHandler
    On Error GoTo ExErrorHandler
    JumpError1
    JumpError2
    Print 1/Zero%
    MsgBox error_s, 0, "Jump Error Handler"
    Exit Sub
ExErrorHandler:
    error_s = error_s & "Error in MainJumpErrorHandler at line " & Erl() & _
        " : " & Error() & CHR$(10)
    Resume Next
End Sub
Sub JumpError1
    REM Causes a jump to the handler in ExampleJumpErrorHandler.
    REM The main error handler indicates that the error location is
    REM at the call to JumpError1 and not in JumpError1.
    Print 1/zero%
    error_s = error_s & "Hey, I am in JumpError1" & CHR$(10)
End Sub

Sub JumpError2
    On Error GoTo ExErrorHandler
    Print 1/zero%
    Exit Sub
ExErrorHandler:
    error_s = error_s & "Error in JumpError2 at line " & Erl() & _
        " : " & Error() & CHR$(10)
    Resume Next
End Sub
```



Figure 38. The last error handler declared is used.

A routine can contain several On Error statements. Each On Error statement can treat errors differently. The error handlers in Listing 64 all used Resume Next to ignore the error and continue execution at the line following the error. Using multiple error handlers, it is possible to skip sections of code when an error occurs (see Listing 65).

TIP The OOo version 3.20 help still incorrectly states that error handling must occur at the start of a routine.

Listing 65. Skip sections of code when an error occurs.

```
On Error GoTo PropertiesDone 'Ignore any errors in this section.
a() = getProperties()         'If unable to get properties then
DisplayStuff(a(), "Properties") 'an error will prevent getting here.
PropertiesDone:
On Error GoTo MethodsDone    'Ignore any errors in this section.
a() = getMethods()
DisplayStuff(a(), "Methods")
MethodsDone:
On Error Goto 0               'Turn off current error handlers.
```

When you write an error handler, you must decide how to handle errors. The functions in Table 23 are used to diagnose errors and to display or log error messages. There is also the question of flow control. Here are some error-handling guidelines:

- Exit the subroutine or function using Exit Sub or Exit Function.
- Let the macro continue to run and ignore the error (see Listing 65).
- Use Resume Next to continue running the macro at the line following the error (see Listing 66 and Figure 39).
- Use Resume to run the same statement again. If the problem is not fixed, the error will occur again. This will cause an infinite loop.
- Use Resume LabelName to continue running at some specified location.

Listing 66. Resume next error handler.

```
Sub ExampleResumeHandler
    Dim s$, z%
    On Error GoTo Handler1 'Add a message, then resume to Spot1
    s = "(0) 1/z = " & 1/z & CHR$(10) 'Divide by zero, so jump to Handler1
Spot1: 'Got here from Handler1
    On Error GoTo Handler2 'Handler2 uses resume
    s = s & "(1) 1/z = "&1/z & CHR$(10) 'Fail the first time, work the second
```

```

On Error GoTo Handler3           'Handler3 resumes the next line
z = 0                           'Allow for division by zero again
s = s & "(2) 1/z = "&1/z & CHR$(10) 'Fail and call Handler3
MsgBox s, 0, "Resume Handler"
Exit Sub
Handler1:
s = s & "Handler1 called from line " & Erl() & CHR$(10)
Resume Spot1
Handler2:
s = s & "Handler2 called from line " & Erl() & CHR$(10)
z = 1 'Fix the error then do the line again
Resume
Handler3:
s = s & "Handler3 called from line " & Erl() & CHR$(10)
Resume Next
End Sub

```

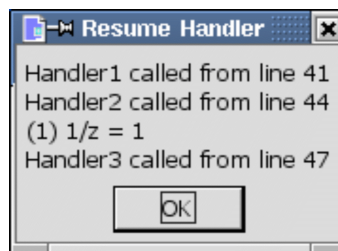


Figure 39. The last error handler declared is used.

TIP Errors that occur in an error handler are not handled; the macro just stops running.

3.10.5. Error handlers — why use them?

When I run a macro and it crashes, I usually understand the sometimes cryptic error messages and recognize how to deal with them. When others run my macros and an error occurs, I usually hear about it because they don't know how to handle it. This is a good indicator that I have not used proper error handling.

You do not have to write an error handler for each routine. If the current routine doesn't have an error handler, but the routine that called it does, the caller's error handler is called. For example, imagine that sub1 has an error handler and it calls sub2 that does not. If an error occurs in sub2, the error handler in sub1 is called.

When you use an error handler, you control how and when a user is notified of an error. Error handlers have uses other than controlling when a user is informed of an error. A properly used error handler can reduce the size of your macro. Consider, for example, the mathematical operations. It is cumbersome to check every mathematical operation before it is used.

```

If x <> 0 Then y = z / x
If x > 0 Then y = Log(x)
If i% < 32767 Then i% = i% + 1

```

Even with my paranoid code checking the arguments, I might still have a numeric overflow. Even worse, nothing is done if an error occurs, and processing continues as normal. Sometimes you can't check anything to avoid the error. For example, prior to OOo version 2.0, the DimArray function returned an invalid empty array. The LBound and UBound functions generate exceptions with these invalid empty arrays. Error handling was used to safely generate the LBound and UBound even if an error occurred. Consider the following cases:

- The argument is not an array.
- In an empty array, UBound < LBound; -1 and 0, for example.
- There are no problems if the array is not an invalid empty array.
- Should the optional dimension be considered?

The code in Listing 67 demonstrates a simple error handler that is able to simply ignore errors. The function returns True if the lower bound is less than or equal to the upper bound — in other words, if the array contains data. If an error occurs, either because the argument is not an array or because this is an invalid empty array, the line does not finish running so the assignment never takes place. If the assignment never takes place, the original default Boolean value of False is used. This is the correct answer. Writing a safe upper-bound or lower-bound routine is left as an exercise for the reader — the safe versions are not required with the fixed UBound and LBound functions released with OOo 2.0.

Listing 67. Determine if an array has stuff.

```
Sub ExampleArrayHasStuff
    Dim a(), b(3), v
    Print ArrayHasStuff(a())           'False, because empty
    Print ArrayHasStuff(v)             'False, not an array, so use error handler
    Print ArrayHasStuff(DimArray())    'False, bad array, so error handler called
    Print ArrayHasStuff(DimArray(3))  'True, this is not empty
    Print ArrayHasStuff(b())           'True, this is not empty
End Sub

Function ArrayHasStuff(v) As Boolean
    REM default value for a Boolean is False, so the default answer is False
    REM If an error occurs, then the statement never completes!
    REM This is a good time to use On Error Resume Next
    On Error Resume Next
    ArrayHasStuff = CBool(LBound(v) <= UBound(v))
End Function
```

An error handler can even be interactive. The code in Listing 68 attempts to copy a file that doesn't exist to a location that doesn't exist. Needless to say, an error is generated. An error message is displayed and the user is asked if the copy should be tried again. The user is given an opportunity to correct any errors and continue.

Listing 68. Copy a file.

```
Sub ExampleCopyAFile()
    CopyAFile("/I/do/not/exist.txt", "/neither/do/I.txt")
```

```

End Sub

Sub CopyAFile(Src$, Dest$)
    On Error GoTo BadCopy: 'Set up the error handler
TryAgain:
    FileCopy(Src$, Dest$) 'Generate the error

AllDone: 'If no error, then continue here
    Exit Sub 'Resume to the AllDone label from the handler

BadCopy: 'Display an error dialog
    Dim rc% 'Ask if should try again
    rc% = MsgBox("Failed to copy from " & Src$ & " to " & Dest$ & " because: "&_
        CHR$(10) & Error() & " at line " & Erl & CHR$(10) &_
        "Try again?", (4 OR 32), "Error Copying")
    If rc% = 6 Then 'Yes, try the command again
        Resume
    End If
    If rc% = 7 Then 'No
        Resume AllDone 'Go to the AllDone label
    End If
End Sub

```

3.11. Conclusion

Building any significant macro requires understanding the syntax of Basic. This chapter covered the key elements:

- The syntax of a macro determines valid and invalid constructs.
- The logic of a macro determines what the macro does.
- Flow control directs the macro as it runs.
- Error handling directs the macro when it does something unexpected.

A complete, well-constructed macro that accomplishes any significant function will most likely use all of these characteristics of OOO programming. The specific elements within OOO that one uses to build a specific program depend on the application, the desired logical behavior of the program, and the best judgment of the programmer. One major part of successful programming is developing the experience to apply the ideas of this chapter in the most effective way.

4. Numerical Routines

This chapter introduces the subroutines and functions supported by Basic that are related to numbers — including mathematical functions, conversion routines, formatting numbers as a string, and random numbers. This chapter also discusses alternate number bases.

Numerical subroutines and functions are routines that perform mathematical operations. If you use spreadsheets, you may already be familiar with mathematical functions such as Sum, which adds groups of numbers together, or even IRR, which calculates the internal rate of return of an investment. The numerical routines supported by Basic (see Table 24) are simpler in nature, typically operating on only one or two arguments rather than an entire group of numbers.

Table 24. *Subroutines and functions related to numbers and numerical operations.*

Function	Description
ABS(number)	The absolute value of a specified number.
ATN(number)	The angle, in radians, whose tangent is the specified number in the range of -Pi/2 through Pi/2.
CByte(expression)	Round the String or numeric expression to a Byte.
CCur(expression)	Convert the expression to a Currency type.
CDBl(expression)	Convert a String or numeric expression to a Double.
CDec(expression)	Generate a Decimal type; implemented only on Windows.
CInt(expression)	Round the String or numeric expression to the nearest Integer.
CLng(expression)	Round the String or numeric expression to the nearest Long.
COS(number)	The cosine of the specified angle in radians.
CSng(expression)	Convert a String or numeric expression to a Single.
Exp(number)	The base of natural logarithms raised to a power.
Fix(number)	Chop off the decimal portion.
Format(obj, format)	Fancy formatting, discussed in Chapter 6, “String Routines.”
Hex(n)	Return the hexadecimal representation of a number as a String.
Int(number)	Round the number toward negative infinity.
Log(number)	The logarithm of a number. In Visual Basic .NET this method can be overloaded to return either the natural (base e) logarithm or the logarithm of a specified base.
Oct(number)	Return the octal representation of a number as a String.
Randomize(num)	Initialize the random number generator. If num is omitted, uses the system timer.
Rnd	Return a random number as a Double from 0 through 1.
Sgn(number)	Integer value indicating the sign of a number.
SIN(number)	The sine of an angle in radians.
Sqr(number)	The square root of a number.
Str(number)	Convert a number to a String with no localization.
TAN(number)	The tangent of an angle in radians.
Val(str)	Convert a String to a Double. This is very tolerant to non-numeric text.

The mathematical functions presented in this chapter are well-known and understood by mathematicians, engineers, and others who look for excuses to use calculus in everyday life. If that is not you — if, perhaps, you do *not* consider the slide rule to be the coolest thing since sliced bread — don't panic when the coverage starts to become mathematical in nature. I have tried to make the information accessible while still providing the in-depth information for those who desire it. The routines are topically grouped into sections so you can skip sections that you know that you will not use.

The numerical routines perform operations on numerical data. Basic tries to convert arguments to an appropriate type before performing an operation. It is safer to explicitly convert data types using conversion functions, as presented in this chapter, than to rely on the default behavior. When an Integer argument is required and a floating-point number is provided, the default behavior is to round the number. For example, “16.8 MOD 7” rounds 16.8 to 17 before performing the operation. The Integer division operator, however, truncates the operands. For example, “Print 4 \ 0.999” truncates 0.999 to 0, causing a division-by-zero error.

TIP

Table 24 contains subroutines and functions, not operators such as MOD, +, and \. Operators were covered in Chapter 3, Language Constructs.

4.1. Trigonometric functions

Trigonometry is the study of the properties of triangles and trigonometric functions and of their applications. Discussions of trigonometric functions usually refer to right triangles, which have one angle of 90 degrees (see Figure 40). There is a set of defined relationships among the trigonometric functions, the lengths of the sides of a right triangle, and the corresponding angles in the triangle. When you know these relationships, you can use the trigonometric functions to solve trigonometric problems.

A practical problem that uses trigonometry is to estimate one's distance from a pole or tower of known height. By measuring the observed angle from the ground to the top of the pole, and knowing the height of the pole, your distance from that pole is the height of the pole divided by the tangent of the measured angle. This principle can be applied to golf, sailing, or hiking, to estimate distance from a fixed point of interest (the golf flag, for example, or a radio transmission tower).

The principal trigonometric functions are sine, cosine, and tangent. Each is defined as the ratio between two sides of a right triangle. The values of these functions for any value of the angle, x , correspond to the ratios of the lengths of the sides of the right triangle containing that angle, x . For a given angle, the trigonometric functions fix the lengths of the sides of the right triangle. Likewise, knowing the lengths of any two sides of the right triangle allows one to compute the value of the angle using one of the inverse trigonometric functions.

Basic uses radians as the unit of measure for angles; however, most non-scientists think in degrees. An angle that is 90 degrees, such as the corner of a square, is $\pi/2$ radians.

TIP The built-in constant π is approximately 3.1415926535897932385. π is a fundamental constant widely used in scientific calculations, and is defined as the ratio of the circumference of a circle to its diameter. The sum of the angles in any triangle — including a right triangle — is 180 degrees, or π radians. A tremendous amount of elegant and practical mathematical methods result from this connection between a triangle and a circle. All descriptions of periodic motion build on this foundation, making trigonometry a fundamental and very useful set of mathematical tools.

Using the relationship between degrees and radians, it is easy to convert between radians and degree measurements of angles.

```
degrees = (radians * 180) / Pi
radians = (degrees * Pi) / 180
```

To calculate the sine of a 45-degree angle, you must first convert the angle from degrees to radians. Here's the conversion:

```
radians = (45° * Pi) / 180 = Pi / 4 = 3.141592654 / 4 = 0.785398163398
```

You can use this value directly in the trigonometric function `SIN`.

```
Print SIN(0.785398163398) ' .707106781188
```

To determine the angle whose tangent is 0.577350269189, use the arctangent function. The returned value is in radians, so this value must be converted back to degrees.

```
Print ATN(0.577350269189) * 180 / Pi '29.9999999999731
```

TIP Rounding errors, as discussed later, affect these examples. With infinite precision, the previous example would result in an answer of 30 degrees rather than 29.9999999999731.

The answer is roughly 30 degrees. The triangle in Figure 40 is used to help explain the trigonometric functions.

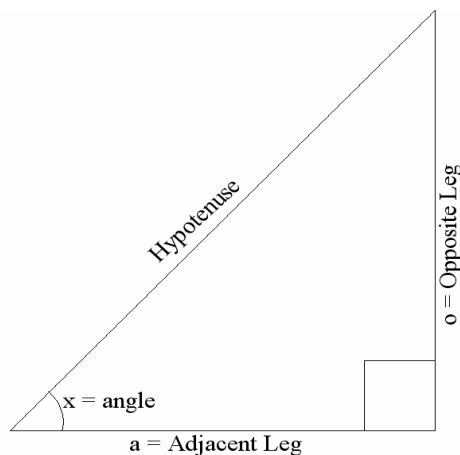


Figure 40. A right triangle has one angle at 90 degrees.

Table 25. Trigonometric functions supported by Basic.

Basic	VB	VB .NET	Return Value
ATN	ATN	Math.Atan	The angle, in radians, whose tangent is the specified number in the range of -Pi/2 through Pi/2.
COS	COS	Math.Cos	The cosine of the specified angle.
SIN	SIN	Math.Sin	The sine of an angle.
TAN	TAN	Math.Tan	The tangent of an angle.

The trigonometric functions supported by Basic are shown in Table 25 and illustrated using the right triangle in Figure 40. The single argument expression is converted to a double-precision number before the function is performed.

$\text{COS}(x) = \text{Adjacent Leg} / \text{Hypotenuse}$

$\text{SIN}(x) = \text{Opposite Leg} / \text{Hypotenuse}$

$\text{TAN}(x) = \text{Opposite Leg} / \text{Adjacent Leg} = \text{SIN}(x) / \text{COS}(x)$

$\text{ATN}(\text{Opposite Leg} / \text{Adjacent Leg}) = x$

The code in Listing 69 solves a series of geometry problems using the trigonometric functions. The code assumes a right triangle, as shown in Figure 40, with an opposite leg of length 3, and an adjacent leg of length 4. The tangent is easily calculated as 3/4 and the ATN function is used to determine the angle. A few other calculations are performed, such as determining the length of the hypotenuse using both the SIN and the COS functions. Also see Figure 41.

Listing 69. ExampleTrigonometric

```
Sub ExampleTrigonometric
    Dim OppositeLeg As Double
    Dim AdjacentLeg As Double
    Dim Hypotenuse As Double
    Dim AngleInRadians As Double
    Dim AngleInDegrees As Double
    Dim s As String
    OppositeLeg = 3
    AdjacentLeg = 4
    AngleInRadians = ATN(3/4)
    AngleInDegrees = AngleInRadians * 180 / Pi
    s = "Opposite Leg = " & OppositeLeg & CHR$(10) & _
        "Adjacent Leg = " & AdjacentLeg & CHR$(10) & _
        "Angle in degrees from ATN = " & AngleInDegrees & CHR$(10) & _
        "Hypotenuse from COS = " & AdjacentLeg/COS(AngleInRadians) & CHR$(10) & _
        "Hypotenuse from SIN = " & OppositeLeg/SIN(AngleInRadians) & CHR$(10) & _
        "Opposite Leg from TAN = " & AdjacentLeg * TAN(AngleInRadians)
    MsgBox s, 0, "Trigonometric Functions"
End Sub
```

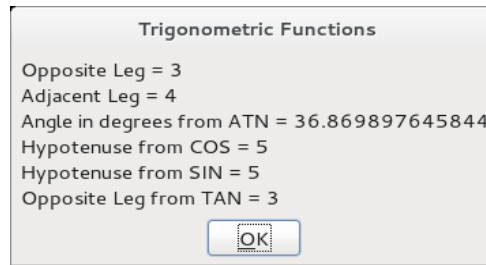


Figure 41. Use the trigonometric functions to solve problems with triangles.

4.2. Rounding errors and precision

Numerical calculations performed on either a computer or calculator are performed with only a finite number of digits; this introduces rounding errors. This isn't a problem with integer numbers. The number $1/3$ is represented in decimal form as 0.33333333, but there needs to be an infinite number of threes following the decimal point. With four digits of precision, this is written as 0.3333. This introduces inaccuracies in the representation and the resulting calculations.

$1/3 + 1/3 + 1/3 = 3/3 = 1$ 'The exact answer is 1
 $0.3333 + 0.3333 + 0.3333 = 0.9999$ 'The finite precision answer, off a bit

The simple macro in Listing 70 demonstrates this problem. The value 0.2 is repeatedly added to the variable *num* until the value is equal to 5. If infinite precision were used, or if the number 0.2 were exactly represented inside the computer, the loop would stop with the variable *num* containing the value 5. The variable never precisely equals the value 5, however, so the loop never stops. The value 5 is printed, but this is only because the Print statement rounds 4.9999999 to the value 5 when it prints the number.

Listing 70. *Rounding errors and finite precision prevent this from stopping.*

```
Dim num As Single
Do
    num = num + 0.2
    If num > 4.5 Then Print num 'prints 4.6, 4.8, 5, 5.199999...
Loop Until num = 5.0
Print num
```

Computers use complex rounding algorithms in an attempt to reduce the impact of finite precision — finite precision means that a finite number of digits and storage are used to represent a number. Although this helps, Listing 70 clearly demonstrates that the internal complex rounding algorithms do not solve the problem. When you compare two floating-point numbers to see if they are equal, it is safer to compare them to a range of values. The code in Listing 71 stops when the variable is greater than or equal to 5.

Listing 71. *Avoid rounding errors by using >= (greater than or equal to).*

```
Dim num As Single
Do
    num = num + 0.2
Loop Until num >= 5.0
```

```
Print num '5.199999
```

The code in Listing 71 works to some extent, but you probably want the loop to exit when the variable *num* is 4.9999999 rather than when it is 5.199999. You can do this by checking to see if two numbers are close rather than equal. The big question is, How close must two numbers be before they are considered equal? You can usually make a simple guess based on what you know about the problem. Single-precision variables can represent about eight digits of precision. Double-precision variables can represent about 16 digits of precision. Don't try to demand more precision from the variables than they support. The code in Listing 71 uses single-precision variables so you can expect roughly seven digits of precision. The code in Listing 72 prints the difference between 5 and *num* — notice that about six digits are correct.

Listing 72. Compare the variable to a range.

```
Dim num As Single
Do
    num = num + 0.2
Loop Until 4.99999 < num AND num < 5.00001
Print 5 - num '4.76837158203125E-07 = 0.000000476837158203125
```

The ABS function returns the absolute value of a number. You can use it to simplify the process of checking to see how close one number is to another.

```
If ABS(num - 5) < 0.00001 Then
```

Using ABS and subtraction indicates how close two numbers are to each other, but it may not be sufficient. For example, light travels at about 299,792,458 meters each second. This number contains nine digits. A single-precision number is accurate to about seven digits. See Listing 73.

Listing 73. Single-precision variables have only seven or eight digits of accuracy.

```
Dim c1 As Single 'Scientists usually use the letter c to represent
Dim c2 As Single 'the speed of light.
c1 = 299792458 'Speed of light in meters per second to nine digits
c2 = c1 + 16 'Add 16 to the speed of light
If c1 = c2 Then 'These are equal because only the first seven
    Print "Equal" 'or eight digits are significant
End If
```

The code in Listing 73 adds 16 to the speed of light, but this does not change the value. This is because only the first seven or eight digits are significant. The code in Listing 74 uses a number that is smaller in magnitude but uses the same number of digits. Adding 1 to the number would change a significant digit, but adding a smaller number still leaves the numbers equal.

Listing 74. Single-precision variables have only seven or eight digits of accuracy.

```
Dim c1 As Single 'Scientists usually use the letter c to represent
Dim c2 As Single 'the speed of light.
c1 = 299.792458 'This is nine digits but it is not the speed of light
c2 = c1 + .0000016 'Must add a smaller number for them to still be equal
If c1 = c2 Then 'These are equal because only the first seven
    Print "Equal" 'or eight digits are significant
End If
```

Floating-point numbers can have different magnitudes — magnitude refers to the size of the number — and it doesn't significantly affect the number of digits that are relevant. To check if two numbers are about the same value, large numbers can differ by a greater amount than small numbers. The greatest allowed difference is dependent upon the magnitude (size) of the numbers; a mathematician calls this the “relative error.” See Listing 75.

Listing 75. Compare two numbers.

```
REM This uses n1 as the primary number of interest
REM n2 is compared to n1 in a relative way
REM rel_diff is the desired relative difference
REM rel_diff is assumed non-negative
Function AreSameNumber(n1, n2, rel_diff) As Boolean
    AreSameNumber = False           'Assume that they are different
    If n1 <> 0 Then                  'Cannot divide by n1 if it is zero
        If ABS((n1-n2)/n1) <= rel_diff Then 'Divide difference by n1 for relative
            AreSameNumber = True      'comparison.
        End If                      'If n1, the number of interest, is
    ElseIf ABS(n2) <= rel_diff Then 'zero, then compare n2 for size.
        AreSameNumber = True
    End If
End Function
```

The code in Listing 75 divides the difference of two numbers by one of the numbers. The code in Listing 76 checks numbers of different sizes to see if they are the same number.

Listing 76. Test if same number.

```
Sub CheckSameNumber
    Dim s1 As Single
    Dim s2 As Single
    Print AreSameNumber(299792458, 299790000, 1e-5) 'True: five digits same
    Print AreSameNumber(299792458, 299700000, 1e-5) 'False: four digits same
    s1 = 299792458                                's1 assigned different value
    s2 = 299792448                                'than s2 but same number.
    Print AreSameNumber(s1, s2, 0.0)               'True: Same number in single precision.
    Print AreSameNumber(299.792458, 299.790000, 1e-5) 'True: five digits same
    Print AreSameNumber(2.99792458, 2.99700000, 1e-5) 'False: four digits same
End Sub
```

A large quantity of literature and research is available on the negative issues associated with floating-point numbers. A complete discussion is therefore well beyond the scope of this book. In general usage, the problems typically aren't that troublesome, but, when they arise, they can be most perplexing if you are not aware of the issues.

4.3. Mathematical functions

The mathematical functions in Basic take a numeric argument. All of the standard types are converted to a Double before they are used. Strings may include hexadecimal and octal numbers. The functions are the same as those available in Visual Basic (see Table 26).

Table 26. Mathematical functions supported by Basic.

Basic	VB	VB .NET	Return Value
ABS	ABS	Math.Abs	The absolute value of a specified number.
Exp	Exp	Math.Exp	The base of natural logarithms raised to a power.
Log	Log	Math.Log	The logarithm of a number. In VB .NET you can overload this method to return either the natural (base e) logarithm or that of a specified base.
Sgn	Sgn	Math.Sign	Integer value indicating the sign of a number.
Sqr	Sqr	Math.Sqrt	The square root of a number.

Use the ABS function to determine the absolute value of a number, which you can think of as simply throwing away the leading + or - sign from the front of the number. The geometrical definition of ABS(x) is the distance from x to 0 along a straight line.

```
ABS(23.33) = 23.33
ABS(-3)    = 3
ABS("-1")  = 1 'Notice that the string value "-1" is converted to a Double
```

Use the Sgn function to determine the sign of a number. An integer with the value -1, 0, or 1 is returned if the number is negative, zero, or positive.

```
Sgn(-37.4) = -1
Sgn(0)     = 0
Sgn("4")   = 1
```

The square root of 9 is 3, because 3 multiplied by 3 is 9. Use the Sqr function to get the square root of a number. The Sqr function can't calculate the square root of a negative number — attempting to do so causes a run-time error.

```
Sqr(100) = 10
Sqr(16)  = 4
Sqr(2)   = 1.414213562371
```

Logarithms were devised by John Napier, who lived from 1550 through 1617. Napier devised logarithms to simplify arithmetic calculations, by substituting addition and subtraction for multiplication and division. Logarithms have the following properties:

```
Log(x*y) = Log(x) + Log(y)
Log(x/y) = Log(x) - Log(y)
Log(x^y) = y * Log(x)
```

The Exp function is the inverse of the Log function. For example, Exp(Log(4)) = 4 and Log(Exp(2)) = 2. By design, logarithms turn multiplication problems into addition problems. This allows the use of logarithms as they were originally designed.

```
Print Exp(Log(12) + Log(3)) '36 = 12 * 3
Print Exp(Log(12) - Log(3)) ' 4 = 12 / 3
```

Logarithms are defined by the equation $y=b^x$. It is then said that the logarithm, base b, of y is x. For example, the logarithm base 10, $10^2 = 100$ so the logarithm, base 10, of 100 is 2. The natural

logarithm, with a base approximated by $e=2.71828182845904523536$, is frequently used because it has some nice mathematical properties. This is called the “natural logarithm” and is used in Basic. Visual Basic .NET allows you to calculate logarithms of other bases. This is easily done using the formula that the logarithm base b is given by $\text{Log}(x)/\text{Log}(b)$, regardless of the base of the logarithm that is used.

Logarithms are not as useful as a general shortcut for calculations today, when lots of computing power is available. However, the logarithmic relationship describes the behavior of many natural phenomena. For example, the growth of populations is often described using logarithms, because geometric growth expressed on a logarithmic graph displays as a straight line. Exponentials and logarithms are also used extensively in engineering computations that describe the dynamic behavior of electrical, mechanical, and chemical systems.

The macro in Listing 77 calculates the logarithm of the number x (first argument) to the specified base b (second argument). For example, use `LogBase(8, 2)` to calculate the log, base 2, of 8 (the answer is 3).

Listing 77. LogBase.

```
Function LogBase(x, b) As Double
    LogBase = Log(x) / Log(b)
End Function
```

4.4. Numeric conversions

Basic tries to convert arguments to an appropriate type before performing an operation. However, it is safer to explicitly convert data types using conversion functions, as presented in this chapter, than to rely on the default behavior, which may not be what you want. When an Integer argument is required and a floating-point number is provided, the default behavior is to round the number. For example, `16.8 MOD 7` rounds 16.8 to 17 before performing the operation. The Integer division operator, however, truncates the operands. For example, “Print `4 \ 0.999`” truncates 0.999 to 0, causing a division-by-zero error.

There are many different methods and functions to convert to numeric types. The primary conversion functions convert numbers represented as strings based on the computer’s locale. The conversion functions in Table 27 convert any string or numeric expression to a number. String expressions containing hexadecimal or octal numbers must represent them using the standard Basic notation. For example, the hexadecimal number 2A must be represented as “&H2A”.

Table 27. Convert to a numerical type.

Function	Type	Description
CByte(expression)	Byte	Round the String or numeric expression to a Byte.
CCur(expression)	Currency	Convert the String or numeric expression to a Currency. The locale settings are used for decimal separators and currency symbols.
CDec(expression)	Decimal	Generate a Decimal type; implemented only on Windows.
CInt(expression)	Integer	Round the String or numeric expression to the nearest Integer.

Function	Type	Description
CLng(expression)	Long	Round the String or numeric expression to the nearest Long.
CDBl(expression)	Double	Convert a String or numeric expression to a Double.
CSng(expression)	Single	Convert a String or numeric expression to a Single.

The functions that return a whole number all have similar behavior. Numeric expressions are rounded rather than truncated. A string expression that does not contain a number evaluates to zero. Only the portion of the string that contains a number is evaluated, as shown in Listing 78.

Listing 78. *CInt and CLng ignore non-numeric values.*

```
Print CInt(12.2)      ' 12
Print CLng("12.5")   ' 13
Print CInt("xxyy")   ' 0
Print CLng("12.1xx") ' 12
Print CInt(-12.2)     '-12
Print CInt("-12.5")   '-13
Print CLng("-12.5xx") '-13
```

CLng and CInt have similar, but not identical, behavior for different types of overflow conditions. Decimal numbers in strings that are too large cause a run-time error. For example, CInt("40000") and CLng("999999999999") cause a run-time error, but CLng("40000") does not. CLng never causes an overflow if a hexadecimal or octal number is too large; it silently returns zero without complaint. CInt, however, interprets hexadecimal and octal numbers as a Long and then converts them to an Integer. The result is that a valid Long generates a run-time error when it is converted to an Integer. A hexadecimal value that is too large to be valid returns zero with no complaints and then is cast to an Integer (see Listing 79).

Listing 79. *CInt interprets the number as a Long, then converts to an Integer.*

```
Print CLng("&HFFFFFFFFFE") '0 Overflow on a Long
Print CInt("&HFFFFFFFFFE") '0 Overflow on a Long then convert to Integer
Print CLng("&HFFFFE")      '1048574
Print CInt("&HFFFFE")       'Run-time error, convert to Long then overflow
```

The code in Listing 80 converts numerous hexadecimal numbers to a Long using CLng. See Table 28 for an explanation of the output in Listing 80.

Listing 80. *ExampleCLngWithHex.*

```
Sub ExampleCLngWithHex
    On Error Resume Next
    Dim s$, i%
    Dim v()
    v() = Array("&HE",      "&HFE",      "&HFFF",      "&HFFFF", _
                "&HFFFFF", "&HFFFFFF", "&HFFFFFFF", "&HFFFFFFF", _
                "&HFFFFFFF", _
                "&HE",      "&HFE",      "&HFE",      "&HFE", _
                "&HFE",      "&HFE",      "&HFE",      "&HFE", _
                "&HFE")
    For i = LBound(v()) To UBound(v())
```

```

s = s & i & " CLng(" & v(i) & ") = "
s = s & CLng(v(i))
s = s & CHR$(10)
Next
MsgBox s
End Sub

```

Table 28. Output from Listing 80 with explanatory text.

Input	CLng	Explanation
F	15	Correct hexadecimal value.
FF	255	Correct hexadecimal value.
FFF	4095	Correct hexadecimal value.
FFFF	65535	Correct hexadecimal value.
FFFFF	1048575	Correct hexadecimal value.
FFFFFF	16777215	Correct hexadecimal value.
FFFFFFF	268435455	Correct hexadecimal value.
FFFFFFFF	-1	Correct hexadecimal value; used to fail.
FFFFFFFFF	0	Overflow returns zero; this is nine hexadecimal digits.
E	14	Correct hexadecimal value.
FE	254	Correct hexadecimal value.
FFE	4094	Correct hexadecimal value.
FFFE	65534	Correct hexadecimal value.
FFFFE	1048574	Correct hexadecimal value.
FFFFFE	16777214	Correct hexadecimal value.
FFFFFEE	268435454	Correct hexadecimal value.
FFFFFFFE	-2	Correct hexadecimal value; used to fail.
FFFFFFFEE	0	Overflow returns zero; this is nine hexadecimal digits.

When writing numbers, you don't need to include leading zeros. For example, 3 and 003 are the same number. A Long Integer can contain eight hexadecimal digits. If only four digits are written, you can assume there are leading zeros. When the hexadecimal number is too large for a Long, a zero is returned. The negative numbers are just as easily explained. The computer's internal binary representation of a negative number has the first bit set. The hexadecimal digits 8, 9, A, B, C, D, E, and F all have the high bit set when represented as a binary number. If the first hexadecimal digit has the high bit set, the returned Long is negative. A hexadecimal number is positive if it contains fewer than eight hexadecimal digits, and it is negative if it contains eight hexadecimal digits and the first digit is 8, 9, A, B, C, D, E, or F. Well, this used to be true.

The CByte function has the same behavior as CInt and CLng, albeit with a few caveats. The return type, Byte, is interpreted as a character unless it is explicitly converted to a number. A Byte is a Short Integer that uses only eight bits rather than the 16 used by an Integer.

```

Print CByte("65")           'A has ASCII value 65

```

```
Print CInt(CByte("65xx")) '65 directly converted to a number.
```

TIP

An integer in VB .NET is equivalent to an LO Basic Long.

VB uses different rounding rules. Numbers are rounded to the nearest even number when the decimal point is exactly 0.5; this is called IEEE rounding.

The functions that return a floating-point number all have similar behavior. Numeric expressions are converted to the closest representable value. Strings that contain non-numeric components generate a run-time error. For example, CDbl("13.4e2xx") causes a run-time error. CDbl and CSng both generate a run-time error for hexadecimal and octal numbers that are too large.

Listing 81. CSng and CDbl handle string input.

```
Print CDbl(12.2)      ' 12.2
Print CSng("12.55e1") ' 125.5
Print CDbl("-12.2e-1") '-1.22
Print CSng("-12.5")   '-12.5
Print CDbl("xxyy")    ' run-time error
Print CSng("12.1xx")  ' run-time error
```

The functions CDbl and CSng both fail for string input that contains non-numeric data; the Val function does not. Use the Val function to convert a string to a Double that may contain other characters. The Val function looks at each character in the string, ignoring spaces, tabs, and new lines, stopping at the first character that isn't part of a number. Symbols and characters often considered to be parts of numeric values, such as dollar signs and commas, are not recognized. The function does, however, recognize octal and hexadecimal numbers prefixed by &O (for octal) and &H (for hexadecimal).

The Val function treats spaces differently than other functions treat spaces; for example, Val(" 12 34") returns the number 1234; CDbl and CSng generate a run-time error, and CInt returns 12 for the same input.

Listing 82. Treatment of spaces is different.

```
Sub NumsAreDifferent
    On Error GoTo ErrorHandler:
    Dim s$
    s = "Val(" 12 34") = "
    s = s & Val(" 12 34")
    s = s & CHR$(10) & "CInt(" 12 34") = "
    s = s & CInt(" 12 34")
    s = s & CHR$(10) & "CLng(" 12 34") = "
    s = s & CLng(" 12 34")
    s = s & CHR$(10) & "CSng(" 12 34") = "
    s = s & CSng(" 12 34")
    s = s & CHR$(10) & "CDbl(" 12 34") = "
    s = s & CDbl(" 12 34")
    MsgBox s
Exit Sub
ErrorHandler:
    s = s & " Error: " & Error
    Resume Next
```

End Sub

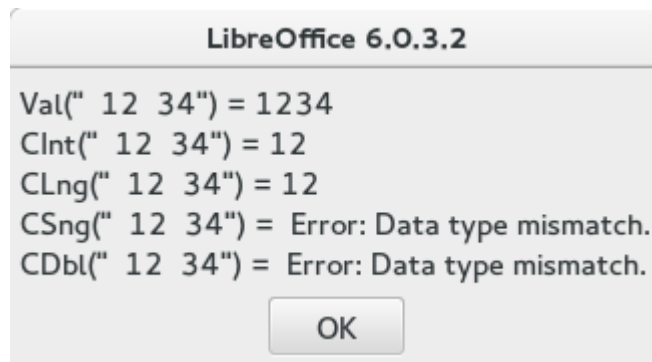


Figure 42. Spaces are not treated consistently during numeric conversion.

TIP

The Val function does not use localization while converting a number so the only recognized decimal separator is the period; the comma can be used as a group separator but is not valid to the right of the decimal. Use CDbl or CLng to convert numbers based on the current locale. In case you forgot, the locale is another way to refer to the settings that affect formatting based on a specific country. See Listing 83.

Listing 83. The Val function is the inverse of the Str function.

```
Sub ExampleVal
    Print Val(" 12 34") '1234
    Print Val("12 + 34") '12
    Print Val("-1.23e4") '-12300
    Print Val(" &FF") '0
    Print Val(" &HFF") '255
    Print Val("&HFFFF") '-1
    Print Val("&HFFFE") '-2
    Print Val("&H3FFFE") '-2, yes, it really converts this to -2
    Print Val("&HFFFFFFFFFFFFFF") '-1
End Sub
```

As of version 1.1.1, the behavior of the Val function while recognizing hexadecimal or octal numbers is strange enough that I call it a bug. Internally, hexadecimal and octal numbers are converted to a 32-bit Long Integer and then the least significant 16 bits are converted to an Integer. This explains why in Listing 83 the number H3FFFE is converted to -2, because only the least significant 16 bits are recognized — in case you forgot, this means the rightmost four hexadecimal digits. This strange behavior is demonstrated in Listing 84. The output is explained in Table 29.

Listing 84. ExampleValWithHex.

```
Sub ExampleValWithHex
    Dim s$, i%
    Dim l As Long
    Dim v()
    v() = Array("&HF", "&HFF", "&HFFF", "&HFFFF", _
        "&HFFFFFF", "&HFFFFFFF", "&HFFFFFFF", "&HFFFFFFF", _
        "&HFFFFFFF", _
```

```

        "&HE",        "&HFE",        "&HFFE",        "&HFFFE", _
        "&HFFFFE",   "&HFFFFFFE",   "&HFFFFFFFE",   "&HFFFFFFF", _
        "&HFFFFFFF", "&H11111111", "&H1111")
For i = LBound(v()) To UBound(v())
    s = s & "Val(" & v(i) & ") = " & Val(v(i)) & CHR$(10)
Next
'This worked in OOo 2.x, but, it
' fails in OOo 3.2.1
'l = "&H" & Hex(-2)
s = s & CHR$(10) & "Hex(-1) = " & Hex(-1) & CHR$(10)
s = s & "Hex(-2) = " & Hex(-2) & CHR$(10)
's = s & "l = &H" & Hex(-2) & " ==> " & l & CHR$(10)
MsgBox s
End Sub

```

Table 29. Output from Listing 84 with explanatory text.

Input	Output	Explanation
F	15	Hexadecimal F is 15.
FF	255	Hexadecimal FF is 255.
FFF	4095	Hexadecimal FFF is 4095.
FFFF	-1	Hexadecimal FFFF is -1 for a 16-bit (two-byte) integer.
FFFFF	-1	Only the rightmost two bytes (four characters) are recognized.
E	14	Hexadecimal E is 14.
FE	254	Hexadecimal FE is 254.
FFE	4094	Hexadecimal FFE is 4094.
FFFE	-2	Hexadecimal FFFE is -2 for a 16-bit (two-byte) integer.
FFFFE	-2	Only the rightmost two bytes are recognized.
FFFFFE	-2	Only the rightmost two bytes are recognized.
FFFFFFE	-2	Only the rightmost two bytes are recognized.
FFFFFFFE	-2	Only the rightmost two bytes are recognized.
HFFFFFFF	-2	Only the rightmost two bytes are recognized.
11111111	4639	Correct value, right most two bytes only.
1111	4639	Correct value

Val converts hexadecimal numbers, but, it only uses the rightmost two bytes.

Use the functions CByte, CInt, CLng, CSng, and CDBl to convert a number, string, or expression to a specific numeric type. Use the functions Int and Fix to remove the decimal portion and return a Double. A string expression that does not contain a number evaluates to zero. Only the portion of the string that contains a number is evaluated. See Table 30.

Table 30. Remove the decimal portion of a floating-point number.

Function	Type	Description
Int	Double	Round the number toward negative infinity.

Function	Type	Description
Fix	Double	Chop off the decimal portion.

The functions Int and Fix differ only in their treatment of negative numbers. Fix always discards the decimal, which is equivalent to rounding toward zero. Int, on the other hand, rounds toward negative infinity. In other words, “Int(12.3)” is 12 and “Int(-12.3)” is -13.

```
Print Int(12.2)      ' 12
Print Fix(12.2)      ' 12
Print Int("12.5")   ' 12
Print Fix("12.5")   ' 12
Print Int("xyy")     ' 0
Print Fix("xyy")     ' 0
Print Int(-12.4)     '-13
Print Fix(-12.4)     '-12
Print Fix("-12.1xx") '-12
Print Int("-12.1xx") '-13
```

The CCur function converts a numerical expression to a currency object. Visual Basic .NET removed support for the CCur function as well as the Currency data type. Basic still supports the Currency data type.

4.5. Number to string conversions

String conversion functions, shown in Table 31, change non-string data into strings. In OOo, text is stored as Unicode version 2.0 values, providing good support for multiple languages. Each String variable can hold up to 65,535 characters.

Table 31. String conversion functions.

Function	Description
Str	Convert from a number to a String with no localization.
CStr	Convert anything to a String. Numbers and dates are formatted based on locale.
Hex	Return the hexadecimal representation of a number as a String.
Oct	Return the octal representation of a number as a String.

4.6. Simple formatting

Use the CStr function to generally convert any type to a String. The returned value is dependent upon the input data type. Boolean values convert to the text “True” or “False.” Dates convert to the short date format used by the system. Numbers are converted to a string representation of the number. See Listing 85.

Listing 85. Output from CStr is locale specific; this is English (USA).

```
Dim n As Long, d As Double, b As Boolean
n = 999999999 : d = EXP(1.0) : b = False
Print "X" & CStr(b) 'XFalse
```

```
Print "X" & CStr(n) 'X999999999
Print "X" & CStr(d) 'X2.71828182845904
Print "X" & CStr(Now) 'X06/09/2018 20:24:24 (almost exactly 15 years after 1st edition)
```

The CStr function performs simple number formatting with knowledge of the current locale. Simple conversion of a number to a string is done with Str. Although the Str function is designed to deal specifically with numeric values, the output is very similar to CStr. When the Str function converts a number to a string, a leading space is always included for the sign of the number. A negative number includes the minus sign, and no leading empty space is present. A non-negative number, on the other hand, includes a leading empty space. The output of Str is not locale specific; a period is always used as the decimal separator. See Listing 86.

Listing 86. *Output from Str is not dependent upon locale.*

```
Dim n As Long, d As Double, b As Boolean
n = 999999999 : d = EXP(1.0) : b = False
Print "X" & Str(b) 'XFalse
Print "X" & Str(n) 'X 999999999
Print "X" & Str(d) 'X 2.71828182845904
Print "X" & Str(Now) 'X06/09/2018 20:28:48 (almost exactly 15 years after 1st edition)
```

The output from the code in Listing 85 and Listing 86 is the same except for a leading space in front of the non-negative numbers. If you run the code using a different locale, such as Germany, the output changes for Listing 85 but not Listing 86.

TIP There is little reason to use Str rather than CStr. Str may run a bit faster, but CStr knows about your current locale.

To demonstrate that CStr is locale specific, I changed my locale to German (Germany) and then ran the code in Listing 85 again. Listing 87 shows that the decimal is now expressed as a comma and that the date is now expressed as MM.DD.YYYY.

Listing 87. *Output from CStr is locale specific; this is German (Germany).*

```
Dim n As Long, d As Double, b As Boolean
n = 999999999 : d = EXP(1.0) : b = False
Print "X" & CStr(b) 'XFalse
Print "X" & CStr(n) 'X999999999
Print "X" & CStr(d) 'X2,71828182845904
Print "X" & CStr(Now) 'X14.08.2018 20:39:49
```

4.7. Other number bases, hexadecimal, octal, and binary

Basic provides the functions Hex and Oct to convert a number to hexadecimal and octal. No native support is provided for a conversion to and from binary. You can't directly use the output from Hex and Oct to convert the string back to a number because it is missing the leading "&H" and "&O".

```
Print Hex(447) '1BF
Print CInt("&H" & Hex(747)) '747
Print Oct(877) '1555
Print CInt("&O" & Oct(213)) '213
```

The source code for Chapter 2 contains the function `IntToBinaryString`, which converts an integer to a binary number in the Operators module in the source code file `SC02.sxw`. The function is very flexible but it isn't particularly fast. A faster routine using the `Hex` function is shown in Listing 88.

Listing 88. *IntToBinaryString*.

```
Function IntToBinaryString(ByVal x As Long) As String
    Dim sHex As String
    Dim sBin As String
    Dim i As Integer
    sHex = Hex(x)
    For i=1 To Len(sHex)
        Select Case Mid(sHex, i, 1)
            Case "0"
                sBin = sBin & "0000"
            Case "1"
                sBin = sBin & "0001"
            Case "2"
                sBin = sBin & "0010"
            Case "3"
                sBin = sBin & "0011"
            Case "4"
                sBin = sBin & "0100"
            Case "5"
                sBin = sBin & "0101"
            Case "6"
                sBin = sBin & "0110"
            Case "7"
                sBin = sBin & "0111"
            Case "8"
                sBin = sBin & "1000"
            Case "9"
                sBin = sBin & "1001"
            Case "A"
                sBin = sBin & "1010"
            Case "B"
                sBin = sBin & "1011"
            Case "C"
                sBin = sBin & "1100"
            Case "D"
                sBin = sBin & "1101"
            Case "E"
                sBin = sBin & "1110"
            Case "F"
                sBin = sBin & "1111"
        End Select
    Next
    IntToBinaryString = sBin
End Function
```

The code in Listing 88 may be long, but it's very simple. There is a correlation between hexadecimal digits and binary digits; each hexadecimal digit is composed of four binary digits. This relationship does not exist for base 10 numbers. The number is converted to a hexadecimal

number using the Hex function. Each hexadecimal digit is converted to the corresponding binary digits. To convert a binary number in String form back to an Integer, use the code in Listing 89.

Listing 89. BinaryStringToLong.

```
Function BinaryStringToLong(s$) As Long
    Dim sHex As String
    Dim sBin As String
    Dim i As Integer
    Dim nLeftOver As Integer
    Dim n As Integer

    n = Len(s$)
    nLeftOver = n MOD 4
    If nLeftOver > 0 Then
        sHex = SmallBinToHex(Left(s$, nLeftOver))
    End If
    For i=nLeftOver + 1 To n Step 4
        sHex = sHex & SmallBinToHex(Mid(s$, i, 4))
    Next
    BinaryStringToLong = CLng("&H" & sHex)
End Function

Function SmallBinToHex(s$) As String
    If Len(s$) < 4 Then s$ = String(4-Len(s$), "0") & s$
    Select Case s$
        Case "0000"
            SmallBinToHex = "0"
        Case "0001"
            SmallBinToHex = "1"
        Case "0010"
            SmallBinToHex = "2"
        Case "0011"
            SmallBinToHex = "3"
        Case "0100"
            SmallBinToHex = "4"
        Case "0101"
            SmallBinToHex = "5"
        Case "0110"
            SmallBinToHex = "6"
        Case "0111"
            SmallBinToHex = "7"
        Case "1000"
            SmallBinToHex = "8"
        Case "1001"
            SmallBinToHex = "9"
        Case "1010"
            SmallBinToHex = "A"
        Case "1011"
            SmallBinToHex = "B"
        Case "1100"
            SmallBinToHex = "C"
        Case "1101"
```

```

        SmallBinToHex = "D"
    Case "1110"
        SmallBinToHex = "E"
    Case "1111"
        SmallBinToHex = "F"
    End Select
End Function

```

To convert a binary string to an Integer, the number is first converted to a hexadecimal number. A set of four binary digits correspond to a single hexadecimal digit. The number is padded on the left with zeros so that the string can be broken up into blocks of four binary digits. Each block of four binary digits is converted to a single hexadecimal digit. The CLng function is then used to convert the hexadecimal number to decimal form. The routine in Listing 90 demonstrates the use of these functions; also see Figure 43.

Listing 90. ExampleWholeNumberConversions.

```

Sub ExampleWholeNumberConversions
    Dim s As String
    Dim n As Long
    Dim nAsHex$, nAsOct$, nAsBin$
    s = InputBox("Number to convert:", "Long To Other", "1389")
    If IsNull(s) Then Exit Sub
    If Len(Trim(s)) = 0 Then Exit Sub
    n = CLng(Trim(s)) 'Trim removes leading and trailing spaces
    nAsHex = Hex(n)
    nAsOct = Oct(n)
    nAsBin = IntToBinaryString(n)
    s = "Original number = " & CStr(n) & CHR$(10) & _
        "Hex(" & CStr(n) & ") = " & nAsHex & CHR$(10) & _
        "Oct(" & CStr(n) & ") = " & nAsOct & CHR$(10) & _
        "Binary(" & CStr(n) & ") = " & nAsBin & _
        " ==> " & BinaryStringToLong(nAsBin)
    MsgBox(s, 0, "Whole Number Conversions")
End Sub

```

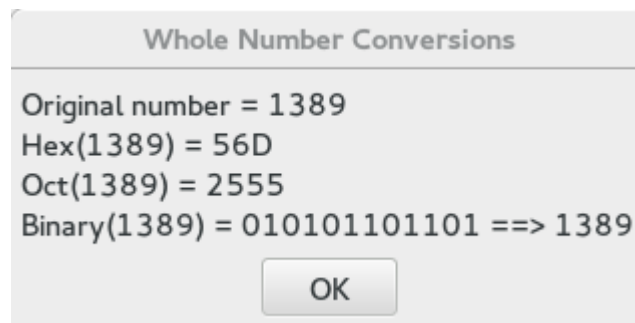


Figure 43. Convert a whole number to hexadecimal, octal, and binary.

4.8. Random numbers

Basic generates floating-point random numbers ranging from 0 through 1. Random numbers generated by computers aren't, in general, random. An algorithm is used that generates "random

numbers” based on a previous random number. The first number on which all other random numbers are based is called the “seed.” You use the Randomize function to specify the seed value. If a value is omitted, the Randomize function uses a value obtained from the system timer. Specifying the starting seed allows you to test programs and generate the same sequence of random numbers each time.

The Rnd function is used to generate the random numbers between 0 and 1. The OOo help claims that the Rnd function accepts an argument; I checked the source code for the Rnd function, which has not changed from version 1.x through 3.2.1, and the argument is ignored.

TIP The included help files correctly claim that the Rnd function accepts an argument that is ignored.

```
Print Rnd()    'Some number from 0 through 1
Print Rnd()    'Another number from 0 through 1
```

The random number generated is some number from 0 through 1. To obtain a different range, perform a few mathematical operations. For example, multiplying a number between 0 and 1 by 10 yields a number between 0 and 10. To use a range that does not start at 0, add an appropriate offset. See Listing 91.

Listing 91. *Return a random number in a range.*

```
Function RndRange(lowerBound As Double, upperBound As Double) As Double
    RndRange = lowerBound + Rnd() * (upperBound - lowerBound)
End Function
```

Use an appropriate function, such as CInt or CLng, if you want a whole number rather than a floating-point number.

```
CLng(lowerBound + Rnd() * (upperBound - lowerBound))
```

I had two functions that solved the same problem — determining the GCD (Greatest Common Divisor) of two integers — and I wanted to know which was faster. I generated random integers and called each routine a few thousand times. While performing timing tests, it’s important to use the same data for each trial. I was able to use random numbers because the Randomize statement allows me to generate the same random numbers every time.

```
Randomize(2)    'reset the random number generator to a known state
t1 = getSystemTicks()
For i = 0 To 30000
    n1 = CLng(10000 * Rnd())
    n2 = CLng(10000 * Rnd())
    call gcd1(n1, n2)
Next
total_time_1 = getSystemTicks() - t1

Randomize(2)    'reset the random number generator to a known state
t1 = getSystemTicks()
For i = 0 To 30000
    n1 = CLng(10000 * Rnd())
    n2 = CLng(10000 * Rnd())
```

```
call gcd2(n1, n2)
Next
total_time_2 = getSystemTicks() - t1
```

4.9. Conclusion

The standard mathematical functions in OpenOffice.org Basic contain few surprises. The conversion functions work well, with some idiosyncrasies while converting strings to numbers. Be certain to choose a function that can handle the format and the ranges used. Rounding is another issue that requires special attention. Although the rounding behavior is documented and consistent, different functions and operators cause rounding to occur differently.

5. Array Routines

This chapter introduces the subroutines and functions supported by Basic that are used to manipulate arrays. It covers methods for manipulating arrays, creating arrays with data, creating arrays with no data, and changing the dimension of arrays. This chapter also presents methods to inspect array variables.

An array is a data structure in which similar elements of data are arranged in an indexed structure — for example, a column of names or a table of numbers. Basic has subroutines and functions that change array dimensions, inspect existing arrays, and convert between arrays and scalar (non-array) data types.

The majority of the routines listed in Table 32 require an array variable as the first argument. Array variables used as arguments to routines can be written with trailing parentheses. Parentheses after the variable are optional, but they used to be required (see Listing 92).

TIP There is no way to determine if a() refers to an array or a function while reading code; you must find where the item in question is declared.

Listing 92. *Parentheses are not always required but are always allowed.*

```
Sub AreArrayParensRequired
    Dim a(1 To 2)      'a() is declared with specified dimensions
    Dim b()            'b() is declared as an array without specified dimensions
    Dim c              'c is a variant and may reference an array.
    c = Array(1, 2)    'c references a Variant array
    Print IsArray(a()) 'True
    Print IsArray(b()) 'True
    Print IsArray(c()) 'True
    Print IsArray(a)   'True
    Print IsArray(b)   'True
    Print IsArray(c)   'True
End Sub
```

Table 32. *Summary of subroutines and functions related to arrays.*

Function	Description
Array(args)	Return a Variant array that contains the arguments.
DimArray(args)	Return an empty Variant array. The arguments specify the dimension(s).
IsArray(var)	Return True if this variable is an array, False otherwise.
Join(array) Join(array, delimiter)	Concatenate the array elements separated by the optional string delimiter and return as a String. The default delimiter is a single space.
LBound(array) LBound(array, dimension)	Return the lower bound of the array argument. The optional dimension specifies which dimension to check. The first dimension is 1.
ReDim [Preserve] var(args) [As Type]	Change the dimension of an array using the same syntax as the DIM statement. The keyword Preserve keeps existing data intact. “As Type” is optional.
Split(str) Split(str, delimiter)	Split the string argument into an array of strings. The default delimiter is a space. The optional argument n limits the number of strings returned.

Function	Description
Split(str, delimiter, n)	
UBound(array)	Return the upper bound of the array argument. The optional dimension specifies which dimension to check. The first dimension is 1.
UBound(array, dimension)	

The word “dimension” is used to refer to arrays similarly to the way that “dimensions” is used to refer to spatial dimensions. For example, an array with one dimension is like a line; you can set boxes along the line that represent data. An array with two dimensions is like a grid with rows and columns of data.

```
Dim a(3) As Integer      'One-dimensional array
Dim b(3 To 5) As String  'One-dimensional array
Dim c(5, 4) As Integer   'Two-dimensional array
Dim d(1 To 5, 4) As Integer 'Two-dimensional array
```

5.1. How big can an array be?

The maximum size of an array is limited by available memory and is dependent on your operating system. In practice, assume that you can have about 1 billion entries.

For software developers, internally, an array is implemented using a C++ `std::vector`, now go look it up.

5.2. Array() quickly builds a one-dimensional array with data

Use the Array function to quickly build a Variant array with data. The Array function returns a Variant array that contains the arguments to the function. See Listing 93. This is an efficient method to create a Variant array containing a predefined set of values. One entry in the array is created for each argument.

Use the Array function to quickly generate an array that already has data. The following code creates an array with five elements, zero through four, and then individually initializes each element.

```
Dim v(4)
v(0) = 2 : v(1) = "help" : v(2) = Now : v(3) = True : v(4) = 3.5
```

This can be done in a much simpler manner. Constants can be used as arguments to functions. You aren’t always forced to assign a value to a variable so that you can call a function.

```
Dim v()
Dim FirstName$ : FirstName = "Bob"
v() = Array(0, "help", Now, True, 3.5)
Print Join(Array("help", 3, "Joe", FirstName))
```

The argument list is a comma-separated list of expressions, which can be of any type because each element in the returned array is a variant.

Listing 93. *The Array function returns a Variant array.*

```
Dim vFirstNames           'A Variant can reference an array
Dim vAges()               'A Variant array can reference an array
vFirstNames = Array("Tom", "Rob") 'Array contains strings
vAges = Array(18, "Ten")    'Array contains a number and a string
Print vAges(0)             'First element has the value 18
```

Variant variables can contain any type, including an array. I frequently use variant variables when I retrieve values from methods when I'm not certain of the return type. I then inspect the type and use it appropriately. This is a convenient use of the Variant variable type. Because a Variant variable can contain any type — including an array — each element in a Variant array can also contain an array. The code in Table 33 demonstrates placing an array inside another array. The code in each column is roughly equivalent.

TIP	Although user defined structures cannot contain arrays, they can contain a variant, which can contain an array.

One advantage of Variant arrays is that it's possible to easily build collections of information of different types. For example, the item description (String), stock-tracking ID (Integer), and invoice amount (Double or Currency) can readily be stored as rows in an array, with each type of data stored in a single row for each customer. This allows array rows to behave more like rows in a database. Older programming languages required use of separately declared arrays for each data type, with added programming overhead of managing the use of multiple, related arrays of data.

Table 33. *A Variant can contain an array, so these accomplish the same thing.*

<pre>Dim v(1) v(0) = Array(1, 2, 3) v(1) = Array("one", "two", "three")</pre>	<pre>Dim v() v = Array(Array(1, 2, 3), _ Array("one", "two", "three"))</pre>

In OOo version 1.x, to address an array containing an array, you had to first extract the contained array, and then subscript the contained array. Much of my existing code was written based on this.

Listing 94. *Cumbersome method to subscript an array in an array.*

```
v = Array(Array(1, 2, 3), Array("one", "two", "three"))
x = v(0)           'This is very cumbersome.
Print x(1)         'Prints 2
```

Somewhere between version 2.x and 3.x, the obvious solution was introduced; you can directly access the contained array. This is particularly useful while using data arrays returned by Calc containing cell data.

Listing 95. *In OOo 3.x, you no longer need to extract the contained array to use it.*

```
Sub ArrayInArray
  Dim v() : v = Array(Array(1, 2, 3), Array("one", "two", "three"))
  Print v(0)(1)
End Sub
```

Although it's easy to create an array inside of an array, it's typically easier to use an array with multiple dimensions, as shown in Listing 96. The “array of arrays” construction is sometimes useful, if there is an obvious relationship with the natural organization of the data. Generally, it is best to select a way to organize the data that has the most direct, natural, and memorable relationship to how it is produced, used, and manipulated.

Listing 96. *It is easier to use multi-dimensioned arrays than arrays inside of arrays.*

```
Dim v(0 To 1, 0 To 2)
v(0, 0) = 1      : v(0, 1) = 2      : v(0, 2) = 3
v(1, 0) = "one" : v(1, 1) = "two" : v(1, 2) = "three"
Print v(0, 1)    'prints 2
```

A Variant array can be assigned to any other array, regardless of its declared type. Assigning one array to another causes one array to reference the other; they become the same array. As mentioned earlier, this is a bad idea. This is considered a bug and it may not be allowed in later versions of Basic. Use the Array function and enjoy the flexibility, but assign the returned Variant array to either a Variant or a Variant array.

Listing 97. *Assign an string array to an integer array.*

```
Sub BadArrayTypes
    Dim a(0 To 1) As Integer
    Dim b(0 To 1) As String
    b(0) = "zero": b(1) = "one"
    a() = b()
    Print a(0)
End Sub
```

TIP

Assigning a Variant array to variables declared as a non-Variant array is ill-advised. For example, after an Integer array has been assigned to reference a Variant array, it's possible to assign non-integer values to elements in the array. References to the array won't return the expected values due to the mismatch between Integer and Variant data types.

```
Dim a(0 To 4) As Integer ' Create an array of Integers.
a(2) = "Tom"             ' Assign a string to an Integer variable.
Print a(2)               ' 0, because the string is converted to zero.
a() = Array(4, "Bob", 7) ' Array always returns a variant array.
a(2) = "Tom"             ' a() is now a variant.
Print a(2)               ' Tom
```

5.3. DimArray creates empty multi-dimensional arrays

The DimArray function creates and returns a dimensioned Variant array. This allows the dimensions of the array to be determined at run time. The arguments specify the dimensions of the array; each argument specifies one dimension. If no arguments are present, an empty array is created.

The primary use of the DimArray statement is to create an empty, dimensioned Variant array. If you know the size of the array that you will need, you can declare it when you declare the variable. If you don't know the size, and if a Variant array is acceptable, then you can create the empty, dimensioned array at run time.

Listing 98. *DimArray returns a dimensioned Variant array that contains no data.*

```
i% = 7
v = DimArray(3*i%) 'Same as Dim v(0 To 21)
v = DimArray(i%, 4) 'Same as Dim v(0 To 7, 0 To 4)
```

The code in Listing 98 does not show how the variable v is declared. This works equally well if v is declared as a Variant, or a Variant array. The argument list is a comma-separated list of expressions. Each expression is rounded to an integer and used to set the range of one dimension of the returned array.

```
Dim a As Variant
Dim v()
Dim i As Integer
i = 2
a = DimArray(3) 'Same as Dim a(0 To 3)
a = DimArray(1+i, 2*i) 'Same as Dim a(0 To 3, 0 To 4)

v() = DimArray(1) 'Same as Dim v(0 To 1)
v(0) = Array(1, 2, 3) 'Oh no, not this again!
v(1) = Array("one", "two", "three") 'You can do it, but yuck!

v = DimArray(1, 2) 'Now that makes more sense!
v(0, 0) = 1 : v(0, 1) = 2 : v(0, 2) = 3
v(1, 0) = "one" : v(1, 1) = "two" : v(1, 2) = "three"
Print v(0, 1) 'prints 2
```

TIP

Option Base 1 has no effect on the dimensions of the array returned by the DimArray function. For each dimension, the lower bound of the range is always zero and the upper bound is the rounded integer value of the relevant expression.

5.4. Change the dimension of an array

Use ReDim to change the dimensions of an existing array by using the same syntax as the DIM statement. Increasing the dimension of an array while using the keyword Preserve preserves all of the data, but decreasing the dimension causes data to be lost by truncation. Unlike some variants of BASIC, LO Basic allows all dimensions of an array to be changed while also preserving data.

The primary use of the ReDim statement is to change the dimension of an existing array. If you know the size of the array that you will need, you can declare it when you declare the variable. If you don't know the size ahead of time, you can declare the array with any size, including as an empty array, and then change the dimension when you know it.

```
Dim v() As Integer
Dim x(4) As Integer
i% = 7
ReDim v(3*i%) As Integer 'Same as Dim v(0 To 21) As Integer.
ReDim x(i%, 1 To 4) As Integer 'Same as Dim x(0 To 7, 1 To 4).
```

The ReDim statement changes the dimension of an existing array, even an empty one. ReDim specifies both the dimensions and the type. The type specified with the ReDim statement must

match the type specified when the variable is declared. If the types differ, you'll see the compile-time error "Variable already defined."

```
Dim a() As Integer           'Empty Integer Array.
Dim v(8)                     'Variant array with nine entries.
ReDim v()                     'v() is a valid empty array.
ReDim a(2 To 4, 5) As Integer 'a() is a two-dimensional array.
```

The `DimArray` function creates and returns a dimensioned Variant array that contains no data. This is not useful if you require an array of a specific type, or if you simply need to change the dimensions of an existing array while preserving data. The `ReDim` statement changes the dimensions of an existing array with the option of preserving existing data. You can use the `ReDim` statement to change a dimensioned array to an empty array.

The subroutine in Listing 99 contains many examples of the `ReDim` statement using the keyword `Preserve`. Figure 44 shows the results of these commands.

Listing 99. *Use `ReDim` with `Preserve` to change dimension and preserve data.*

```
Sub ExampleReDimPreserve
    Dim a(5) As Integer      'A dimensioned array, 0 To 5
    Dim b()                  'An empty array of type Variant
    Dim c() As Integer       'An empty array of type Integer
    Dim s$                   'The string that accumulates the output text

    REM a is dimensioned from 0 to 5 where a(i) = i
    a(0) = 0 : a(1) = 1 : a(2) = 2 : a(3) = 3 : a(4) = 4 : a(5) = 5
    s$ = "a() at start = " & Join(a()) & CHR$(10)

    REM a is dimensioned from 1 to 3 where a(i) = i
    ReDim Preserve a(1 To 3) As Integer
    s$ = s$ & "ReDim Preserve a(1 To 3) = " & Join(a()) & CHR$(10)

    ReDim a() As Integer
    s$ = s$ & "ReDim a() has LBound = " & _
        LBound(a()) & " UBound = " & UBound(a()) & CHR$(10)

    REM Array() returns a Variant type
    Rem b is dimensioned from 0 to 9 where b(i) = i+1
    b = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    s$ = s$ & CHR$(10) & "b() at start = " & Join(b()) & CHR$(10)

    REM b is dimensioned from 1 to 3 where b(i) = i+1
    Dim il%, iu%
    il = 1 : iu = 3
    ReDim Preserve b(il To iu)
    s$ = s$ & "ReDim Preserve b(1 To 3) = " & Join(b()) & CHR$(10)

    ReDim b(-5 To 5)
    s$ = s$ & "ReDim b(-5 To 5) = " & Join(b()) & CHR$(10)
    s$ = s$ & "ReDim b(-5 To 5) has LBound = " & _
        LBound(b()) & " UBound = " & UBound(b()) & CHR$(10) & CHR$(10)
```

```

ReDim b(-5 To 5, 2 To 4)
s$ = s$ & "ReDim b(-5 To 5, 2 To 4) has dimension 1 LBound = " & _
    LBound(b()) & " UBound = " & UBound(b()) & CHR$(10)
s$ = s$ & "ReDim b(-5 To 5, 2 To 4) has dimension 2 LBound = " & _
    LBound(b(), 2) & " UBound = " & UBound(b(), 2) & CHR$(10)

MsgBox s$, 0, "ReDim Examples"
End Sub

```

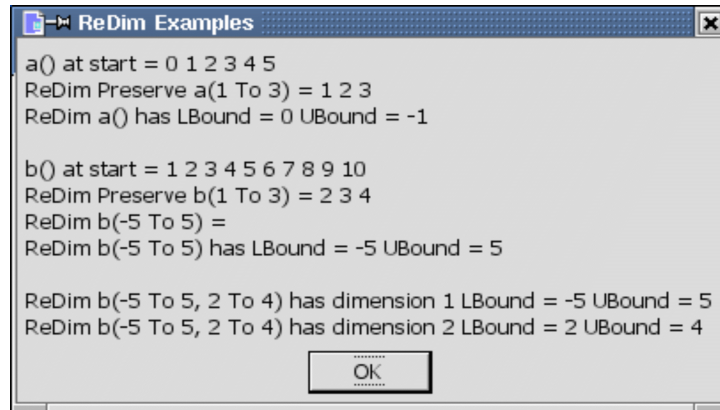


Figure 44. Use ReDim to change the dimensions of an array.

5.5. Array to String and back again

Just as it's common to convert an array of values into a single string for display, it is also common to split a string into multiple pieces. Basic provides these abilities with the functions Join and Split.

The first argument to the Join function is a one-dimensional array. Any other dimension causes a run-time error. The elements in the array are concatenated with an optional delimiter string between each element. The default delimiter is a single space.

```

Join(Array(1, 2, 3))      '1 2 3   using the default delimiter
Join(Array(1, 2, 3), "X") '1X2X3  specifying a delimiter
Join(Array(1, 2, 3), "")  '123    specifying an empty delimiter

```

The Split function returns a Variant array of strings, created by breaking a string into multiple strings based on a delimiter. In other words, it parses a string into pieces with one command. The delimiter separates portions of the string. For example, the delimiter "XY" splits "12XY11XY22" into the strings ("12", "11", "22"). The delimiter defaults to a space but can be any string expression with length greater than zero.

```

Split("1 2 3")           'return Array("1", "2", "3") split on " "
Split("1, 2, 3", ", ")   'return Array("1", "2", "3") split on ", "

```

The optional third argument is used to limit the size of the returned array. This is used only to limit the returned size and has no effect if the returned size is less than the limit. For example, the 4 in Split("1X2X3", "X", 4) has no effect because the returned array has only three elements. If the

size is limited, however, the last element in the array contains the remainder of the unparsed string.

```
Split("1, 2, 3", ",", 2) 'return Array("1", "2, 3") split on ",", "
```

TIP

The second argument to Split is a string, so, LO automatically converts it to a string. The statement Split("0 1 2 3", 2) converts the 2 to a string and uses it as the delimiter. The returned array contains two elements, "0 1 " and " 3". You must specify the delimiter if you want to specify the number of strings returned. The correct format is Split("0 1 2 3", " ", 2).

The Split function assumes that a string comes before and after each delimiter, even if the string has length zero.

```
Split("X1XX2X", "X") = ("", "1", "", "2", "")
```

The first returned string is empty because the first argument contains a leading delimiter. Two consecutive delimiters produce an empty string between the "1" and the "2". Finally, the trailing string is empty because there is a trailing delimiter.

The Split function is almost the inverse of the Join function. The Join function can use a zero-length string as the delimiter, but the Split function cannot. If the joined string contains the delimiter, splitting the string will produce a different set of strings. For example, joining "a b" and "c" with a space produces "a b c". Splitting this with a space produces ("a", "b", "c"), which is not the original set of strings.

I spent a lot of time writing and debugging a macro to parse through a string to remove all occurrences of the text "Sbx". Using Split and Join is significantly smaller and faster:

```
Join(Split(s, "Sbx"), "")
```

5.6. Array inspection functions

The most fundamental thing to ask about an array is whether or not it really is an array. The IsArray function returns True if the argument is an array, and False otherwise. Use the LBound and UBound functions to determine the lower and upper bounds of an array. An array is empty if the upper bound is less than the lower bound.

The first argument to LBound and UBound is the array to check. The second optional argument is an integer expression specifying which dimension is returned. The default value is 1, which returns the lower bound of the first dimension.

```
Dim a()  
Dim b(2 to 3, -5 To 5)  
Print LBound(a()) ' 0  
Print UBound(a()) '-1 because the array is empty  
Print LBound(b()) ' 2 no optional second argument so defaults to 1  
Print LBound(b(), 1) ' 2 optional second argument specifies first dimension  
Print UBound(b(), 2) ' 5
```

If the value of the second argument doesn't contain a valid value, if it's greater than the number of dimensions, or if it's less than 1, a run-time error occurs.

Listing 100. SafeUBound will not generate an error.

```
Function SafeUBound(v, Optional n) As Integer
    SafeUBound = -1          'If an error occurs, this is already set
    On Error GoTo BadArrayFound 'On error skip to the end
    If IsMissing(n) Then     'Was the optional argument used?
        SafeUBound = UBound(v)
    Else
        SafeUBound = UBound(v, n) 'Optional argument is present
    End If
BadArrayFound:              'Jump here on error
    On Error GoTo 0          'Turn off this error handler
End Function
```

The macro in Listing 100 properly returns -1 if an error occurs. The proper value is returned for invalid empty arrays, but it also returns -1 if the first argument isn't an array or if the second argument is simply too large. The ArrayInfo function in Listing 101 uses a similar technique to return array information about a variable. Also see Figure 45.

Listing 101. Print information about an array.

```
REM If the first argument is an array, the dimensions are determined.
REM Special care is given to an empty array that was created using DimArray
REM or Array.
REM a      : Variable to check
REM sName : Name of the variable for a better looking string
Function arrayInfo(a, sName$) As String

    REM First, verify that:
    REM the variable is not NULL, an empty Object
    REM the variable is not EMPTY, an uninitialized Variant
    REM the variable is an array.
    If IsNull(a) Then
        arrayInfo = "Variable " & sName & " is Null"
        Exit Function
    End If
    If IsEmpty(a) Then
        arrayInfo = "Variable " & sName & " is Empty"
        Exit Function
    End If
    If Not IsArray(a) Then
        arrayInfo = "Variable " & sName & " is not an array"
        Exit Function
    End If

    REM The variable is an array, so get ready to work
    Dim s As String          'Build the return value in s
    Dim iCurDim As Integer  'Current dimension
    Dim i%, j%              'Hold the LBound and UBound values
    On Error GoTo BadDimension 'Set up the error handler
    iCurDim = 1             'Ready to check the first dimension

    REM Initial pretty return string
    s = "Array dimensioned as " & sName$ & "("
```

```

Do While True                                'Loop forever

    i = LBound(a()), iCurDim                'Error if dimension is too large or
    j = UBound(a()), iCurDim                'if invalid empty array

    If i > j Then Exit Do                     'If empty array then get out

    If iCurDim > 1 Then s = s & ", " 'Separate dimensions with a comma
    s = s & i & " To " & j                 'Add in the current dimensions
    iCurDim = iCurDim + 1                 'Check the next dimension
Loop

REM Only arrive here if the array is a valid empty array.
REM Otherwise, an error occurs when the dimension is too
REM large and a jump is made to the error handler
REM Include the type as returned from the TypeName function.
REM The type name includes a trailing "()" so remove this
s = s & ") As " & Left(TypeName(a), LEN(TypeName(a))-2)
arrayInfo = s
Exit Function

BadDimension:
REM Turn off the error handler
On Error GoTo 0

REM Include the type as returned from the TypeName function.
REM The type name includes a trailing "()" so remove this
s = s & ") As " & Left(TypeName(a), LEN(TypeName(a))-2)

REM If errored out on the first dimension then this must
REM be an invalid empty array.
If iCurDim = 1 Then s = s & " *** INVALID Empty Array"
arrayInfo = s
End Function

Sub UseArrayInfo
    Dim i As Integer, v
    Dim ia(1 To 3) As Integer
    Dim sa() As Single
    Dim m(3, 4, -4 To -1)

    Dim s As String
    s = s & arrayInfo(i, "i") & CHR$(10)      'Not an array
    s = s & arrayInfo(v, "v") & CHR$(10)      'Empty variant
    s = s & arrayInfo(sa(), "sa") & CHR$(10)   'Empty array
    s = s & arrayInfo(Array(), "Array") & CHR$(10) 'BAD empty array
    s = s & arrayInfo(ia(), "ia") & CHR$(10)
    s = s & arrayInfo(m(), "m") & CHR$(10)
    MsgBox s, 0, "Array Info"
End Sub

```

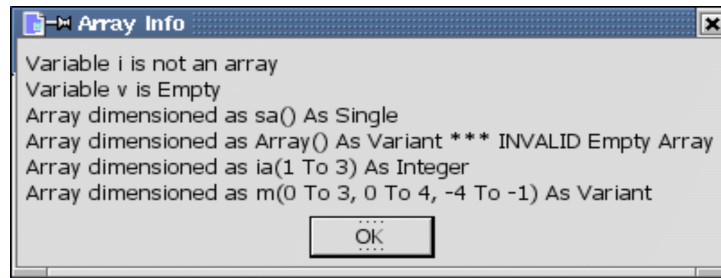


Figure 45. Use proper error handling to determine the dimension of the array.

An array with one dimension may have an upper bound that is less than the lower bound. This indicates that the array has no allocated spots for data. This is different than an array that has data locations allocated but no data has been saved in them. For most data types, such as Integer, if space is allocated for an integer, then it has a value.

```
Dim a(3) As Integer 'This array has four integer values, they are all zero
Dim b(3)           'This array has four Variants, they are all Empty
Dim c()           'This array has one dimension and no space Ubound < Lbound
v = Array()       'This array has zero dimensions.
```

5.7. Conclusion

Array handling in Basic is very flexible. You have the ability to inspect arrays and to change their dimensions. Using the Variant type in Basic provides a great deal of flexibility for creating collections of related data of different types. Strings and arrays are related; string arrays can be processed with Join and Split functions, permitting the creation of compact code that is very powerful for processing string information.

6. Date Routines

This chapter introduces the subroutines and functions supported by OpenOffice.org Basic that are related to dates — including functions to retrieve the current date and time, manipulate dates and times, and perform timing functions. It also discusses some potentially unexpected behavior around October 4, 1582, and again around December 30, 1899.

Date variables contain both a date and a time value. Basic stores dates internally as a floating-point Double. The portion of the number to the left of the decimal contains the date, and the fractional portion to the right of the decimal contains the time. For example, adding 1 to a date value adds one day to the date. Adding 1/24 to a date value adds one hour to a date value; remember that there are 24 hours in a day. The date and time functions supported by OpenOffice.org are listed in Table 34.

Table 34. *Functions and subroutines related to dates and times.*

Function	Type	Description
CDate(expression)	Date	Convert a number or string to a date.
CDateFromIso(string)	Date	Convert to a date from an ISO 8601 date representation.
CDateToIso(date)	String	Convert a date to an ISO 8601 date representation.
Date()	String	Return the current date as a string.
DateAdd	Date	Add an interval to a date.
DateDiff	Integer	Returns the number of intervals between two dates.
DatePart	Variant	Obtain a specific part of a date value.
DateSerial(yr, mnth, day)	Date	Create a date from component pieces: Year, Month, Day.
DateValue(date)	Date	Extract the date from a date/time value by truncating the decimal portion.
Day(date)	Integer	Return the day of the month as an Integer from a Date value.
FormatDateTime	String	Format the date and time as a string. Requires OptionCompatible.
GetSystemTicks()	Long	Return the number of system ticks as a Long.
Hour(date)	Integer	Return the hour as an Integer from a Date value.
IsDate(value)	Boolean	Is this (value, converted to a string) a date?
Minute(date)	Integer	Return the minute as an Integer from a Date value.
Month(date)	Integer	Return the month as an Integer from a Date value.
MonthName	String	Return the name of the month based on an integer argument (1-12). Compatibility mode must be true.
Now()	Date	Return the current date and time as a Date object.
Second(date)	Integer	Return the seconds as an Integer from a Date value.
Time()	Date	Return the time as a String in the format HH:MM:SS.
Timer()	Date	Return the number of seconds since midnight as a Date. Cast this to a Long.
TimeSerial(hour, min, sec)	Date	Create a date from component pieces: Hours, Minutes, Seconds.
TimeValue("HH:MM:SS")	Date	Extract the time value from a date; a pure time value between 0 and 1.

Function	Type	Description
WeekDay(date)	Integer	Return the integer 1 through 7 corresponding to Sunday through Saturday.
WeekdayName	String	Return the day of the week based on an integer argument (1-7).
Year(date)	Integer	Return the year as an Integer from a Date value.

6.1. Compatibility issues

When OpenOffice.org added a number and a date, the result was always a date. LibreOffice changed this behavior to always return a number. After breaking existing macros, LibreOffice changed the code so that some combinations involving a date and a number return a date. The safe thing to do is to always cast the result to the type that you want:

```
CDate(2 + Now)
```

6.2. Retrieve the current date and time

Basic has functions to determine the current date and time: Date, Time, and Now (described in Table 35). The Date and Time functions return a string with the current date and time, respectively. The strings are formatted based on the current locale (**Tools | Options | Language Settings | Languages**; and then set the locale). The Now command returns a Date object that contains both the current date and the current time.

TIP Now returns a Date object, which internally is stored as a Double. The functions Date and Time both return a String.

Table 35. Date and time functions in Basic.

Function	Description
Date	Return the current date as a String.
Now	Return the current date and time as a Date object.
Time	Return the current time as a String.

Printing the date and time is easy.

```
Print Date
Print Time
Print Now
```

6.3. Dates, numbers, and strings

Basic recognizes dates in two different string formats. The obvious format is set by the locale. A less obvious format is the ISO 8601 date format. String formats are always assumed to be in a locale-specific format except for routines specific to the ISO 8601 format. Arguments passed to the date and time functions are converted to an appropriate type if possible. As a result, most of the functions in Table 36 accept string, numeric, and date arguments.

Table 36. Date and string conversion functions

Function	Description
CDate	Convert a number or string to a date.
DateValue	Convert a formatted string from December 1, 1582 through December 31, 9999 to a Date value that contains no time.
CDateFromIso	Convert to a date from an ISO 8601 date representation.
CDateToIso	Convert a date to an ISO 8601 date representation.
IsDate	Is this string a properly formatted date?

Use the IsDate function to test if a string contains a valid date. The argument is always converted to a string before it is used, so a numeric argument will return False. The IsDate function tests more than just syntax — it checks to see if the string contains a valid date. For example, “02/29/2003” fails because February 2003 contains only 28 days. The same validity check is not performed on the time component of the string (see Listing 102 and Listing 103).

Listing 102. IsDate verifies that a string contains a valid date.

```
Print IsDate("December 1, 1582 2:13:42") 'True
Print IsDate("2:13:42")                  'True
Print IsDate("12/1/1582")                'True
Print IsDate(Now)                        'True
Print IsDate("26:61:112")                 'True: 112 seconds and 61 minutes!!!
Print IsDate(True)                       'False: Converts to string first
Print IsDate(32686.22332)                 'False: Converts to string first
Print IsDate("02/29/2003")               'False: Only 28 days in February 03
```

The apparent inconsistency with the IsDate function is that “02/29/2003” is an invalid date but “26:61:112” is valid. With time values, if a section of the time is too large, it is simply added to the next section. For example, 61 minutes is one hour and one minute. Again, 112 seconds adds one minute and 52 seconds to the final computed time value. This is demonstrated in Listing 103 and shown in Figure 46. Notice that, in line two, 30 hours becomes six hours and the day is incremented by one.

Listing 103. Converting time is strange.

```
Sub ExampleTimeConversions
    On Error GoTo Oops:
    Dim Dates()
    Dim i As Integer
    Dim s As String
    Dates() = Array("1/1/1 00:00:00 ", "1/1/1 22:40:00 ", "1/1/1 30:40:00 ", _
        "1/1/1 30:100:00 ", "1/1/1 30:100:100")
    For i = LBound(Dates()) To UBound(Dates())
        s = s & CStr(i) & " " & Dates(i) & " => "
        s = s & CDate(Dates(i))
        s = s & CHR$(10)
    Next
    MsgBox s, 0, "Strange Time Values"
Exit Sub
```

```

Oops:
s = s & " Error"
Resume Next
End Sub

```

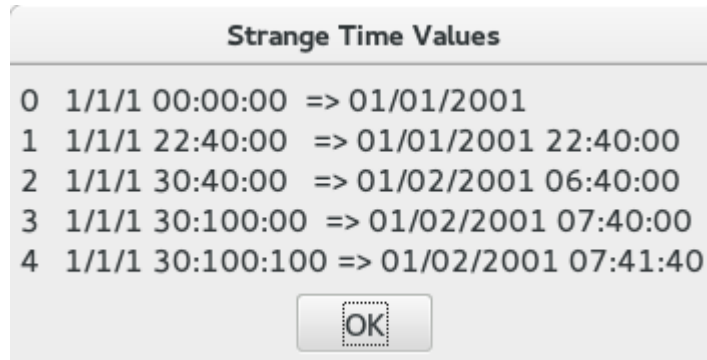


Figure 46. What appear to be invalid times are valid.

Apart from the strange behavior with respect to time, dates and times are converted with no problems except for invalid dates during the skip from the Julian to the Gregorian calendar.

6.4. Locale formatted dates

Use the CDate function to convert a string or number to a date. The CDate function performs a locale-specific conversion including the time component. The DateValue function removes the time portion by removing the fractional portion of the underlying Double. This causes unexpected results with some date values. See Listing 104.

Listing 104. CDate returns a date and a time; DateValue removes the time.

```

Print Now                'for example, 08/16/2003 16:05:53
Print DateValue(Now)     'for example, 08/16/2003

```

The default language on the computers that I use is “English USA.” To use a different locale, select **Tools | Options | Language Settings | Languages** to open the Languages tab on the Options dialog. Then choose a locale.

To test different locales, use the code in Listing 105.

Listing 105. Print date-related information dependent on current locale.

```

Dim d As Date
d = CDate("1/2/3")    'You can use 1.2.3 or 1/2/3 regardless of locale
Print d               'Prints locale specific
Print Year(d)
Print Month(d)
Print Day(d)

```

I ran the code in Listing 105 using four different locales: English USA, English UK, French France, and German Germany. The results are shown in Table 37. The format used to print a date is locale-specific, as you can see in the Germany column. The CDate function accepts dates formatted using the period as a separator, even for the USA locale. Initializing d using CDate("1.2.3") rather than CDate("1/2/3") does not change the output in Table 37.

Table 37. *Locale affects the date.*

Code	USA	UK	France	Germany
Print d	"01/02/2003"	"01/02/2003"	"01/02/2003"	"01.02.2003"
Print Year(d)	2003	2003	2003	2003
Print Month(d)	1	2	2	2
Print Day(d)	2	1	1	1

6.5. ISO 8601 dates

The International Standard ISO 8601 specifies the numeric representations of date and time. This standard notation helps to avoid confusion in international communication caused by numerous different notations, and increases the portability of computer user interfaces. In addition, these formats have several important advantages for computer usage compared to other traditional date and time notations.

The international standard date notation is YYYY-MM-DD, which is a four-digit year followed by a two-digit month and a two-digit day. The year is based on the usual Gregorian calendar. For example, March 8, 2003 is written as 2003-03-08. The separators are optional, so you could also express the date as 20030308; this is the format returned by CDateToISO. See Listing 121. Other components of the date format are beyond the scope of this book. The ISO format has several advantages:

- ISO 8601 is easily comparable and sortable with a string comparison. This is why I prefer this format when appending dates to file names.
- ISO 8601 is easily readable and writable by software because no translation to and from month names is required.
- ISO 8601 is language and locale independent.

There is no ambiguity compared to other date formats. For example, in other formats, there's often a question as to whether the month or the day is listed first. The convention in Europe, for example, would express the fifth day of June in 2003 as 5/6/03, while in the United States — and North and South America, generally — the same date would commonly be expressed as 6/5/03. The opportunity for confusion is great, especially around the time bills are due near the beginning of each month! This is illustrated by the Month and Day row entries shown in Table 37. Listing 106 Converts the same data to the appropriate ISO 8601 string .

Listing 106. *Convert to ISO 8601.*

```
Sub ExampleCDateToISO
    Print CDateToISO("12/30/1899")      '18991230
    Print CDateToISO(Now)                '20100816
    Print CDateFromISO("20030313")      '03/13/2003
End Sub
```

6.6. Problems with dates

Date and time information usually requires little thought. Just use the date and time functions and the correct expected results are produced. Unfortunately, this falls apart as your dates start moving back in time. Three specific dates and times are of interest for each function:

- December 30, 1899 at 00:00:00, when the internal Date representation is a numerical zero. Any pure time value (without a date component) appears to occur on December 30, 1899.
- October 4, 1582, when the Julian calendar was dropped.
- October 15, 1582, when the Gregorian calendar begins.

Various calendar systems have been used at different times and places around the world. The Gregorian calendar, on which Basic is based, is used almost universally. The predecessor to the Gregorian calendar is the Julian calendar. The two calendars are almost identical, differing only in how they handle leap years. The Julian calendar has a leap year every fourth year, while the Gregorian calendar has a leap year every fourth year except century years that aren't exactly divisible by 400.

The change from the Julian calendar to the Gregorian calendar occurred in October of 1582, based on a scheme instituted by Pope Gregory XIII. The Julian calendar was used through October 4, 1582, at which point 10 days were skipped and it became October 15, 1582. Typically, for dates on or before 4 October 1582, the Julian calendar is used; for dates on or after 15 October 1582, the Gregorian calendar is used. Thus, there is a 10-day gap in calendar dates, but no discontinuity in Julian dates or days of the week. Astronomers, however, typically use Julian dates because they don't have a 10-day gap; discontinuous dates do not typically work well in numerical calculations. As seen in Listing 116, Dates are printed based on the Gregorian calendar, but when the component parts are extracted, they are based on the Julian date.

The ISO 8601 standard, introduced to standardize the exchange of date and time-related data, introduces a complication. The standard states that every date must be consecutive, so changing to the Julian calendar violates the standard (because at the switchover date, the dates would not be consecutive).

The following examples demonstrate converting a date / time with CDate, DateValue, and CDateToISO. The DateValue operates on the value returned from CDate. The standard values work as expected (see Table 38). January 1, 2001 is 36892 days after December 30, 1899, and January 1, 1900 is 2 days after December 30, 1899.

Table 38. *Dates after January 1, 1900 work fine.*

Date / Time	DateValue	CDate	CDateToISO
01/01/1900 12:00 AM	2	2	19000101
01/01/1900 06:00 AM	2	2.25	19000101
01/02/1900 12:00 AM	3	3	19000102
01/02/1900 06:00 AM	3	3.25	19000102
01/01/2001 12:00 AM	36892	36892	20010101
01/01/2001 06:00 AM	36892	36892.25	20010101
01/01/2001 12:00 PM	36892	36892.5	20010101

Values near December 30, 1899 reveals some bugs.

- 1) DateValue generates an error on December 30, 1899. This may be intentional so that DateValue generates an error on a pure time value, which cannot be distinguished from a date/time on December 30, 1899.
- 2) DateValue returns an incorrect answer for all dates before December 30, 1899 for all time values other than midnight.
- 3) CDateToISO returns an incorrect answer for all dates before January 1, 1900 for all time values other than midnight.
- 4) CDateToISO has an exceptional failure on on December 31, 1899 for times after midnight.

Table 39. *Dates near December 30, 1899 are a problem.*

Date / Time	DateValue	CDate	CDateToISO
12/28/1899 12:00 AM	-2	-2	18991228
12/28/1899 06:00 AM	-1	-1.75	18991229
12/29/1899 12:00 AM	-1	-1	18991229
12/29/1899 06:00 AM	Error	-0.75	18991230
12/30/1899 12:00 AM	Error	0	18991230
12/30/1899 06:00 AM	Error	0.25	18991231
12/31/1899 12:00 AM	1	1	18991231
12/31/1899 06:00 AM	1	1.25	18991201

Purely invalid dates generate an error as they should; for example, dates between the end of the Julian calendar and the start of the Gregorian calendar. Issues with DateValue and CdateToISO continue when the time is not midnight.

Table 40. Nothing special about dates near the Julian / Gregorian calendar change.

Date / Time	DateValue	CDate	CDateToISO
10/04/1582 12:00 AM	-115859	-115859	15821014
10/04/1582 06:00 AM	-115858	-115858.75	15821015
10/05/1582 00:00:00	Error	Error	Error
10/05/1582 06:00:00	Error	Error	Error
10/15/1582 12:00 AM	-115858	-115858	15821015
10/15/1582 06:00 AM	-115857	-115857.75	15821016

Listing 107 Demonstrates the issues converting to date / time values.

Listing 107. Demonstrate odd date behavior.

```

Sub OddDateTimeBehavior
    On Error GoTo Oops:
    Dim Dates()
    Dim i As Integer
    Dim s As String
    Dates() = Array("10/04/1582 00:00:00", "10/04/1582 06:00:00", _
        "10/05/1582 00:00:00", "10/05/1582 06:00:00", _
        "10/15/1582 00:00:00", "10/15/1582 06:00:00", _
        "12/28/1899 00:00:00", "12/28/1899 06:00:00", _
        "12/29/1899 00:00:00", "12/29/1899 06:00:00", _
        "12/30/1899 00:00:00", "12/30/1899 06:00:00", _
        "12/31/1899 00:00:00", "12/31/1899 06:00:00", _
        "01/01/1900 00:00:00", "01/01/1900 06:00:00", _
        "01/02/1900 00:00:00", "01/02/1900 06:00:00", _
        "1/1/1 00:00:00", "1/1/1 06:00:00", "1/1/1 12:00:00" _
    )
    For i = LBound(Dates()) To UBound(Dates())
        s = s & CStr(i) & "    " & Dates(i) & " => "
        s = s & CDb1(DateValue(CDate(Dates(i))))
        s = s & " => "
        s = s & CDb1(CDate(Dates(i))) & " => " & CDateToISO(Dates(i))
        s = s & CHR$(10)
    Next
    MsgBox s, 0, "Strange Time Values"
Exit Sub
Oops:
    s = s & " Error"
    Resume Next
End Sub

```

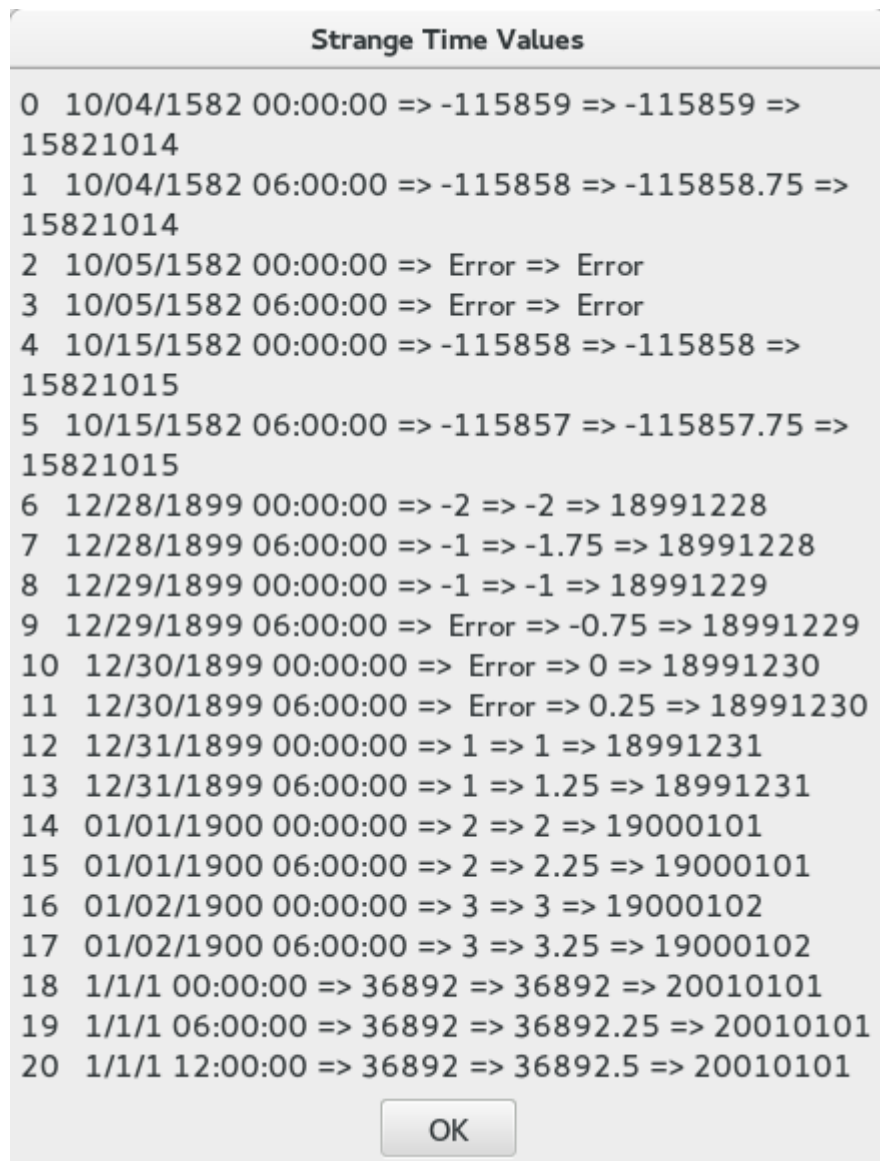


Figure 47. Some Date values convert poorly.

TIP

DateValue fails with a run-time error for dates with a zero day component, such as DateValue(CDate("12/30/1899 06:00:00")). Dates before this return an incorrect value. I might also argue that dates and times that use the same data type is a bug, because it isn't possible to distinguish between a time and a date value for the day 12/30/1899.

DateValue truncates the decimal portion of the number to determine the date. The Int function, however, always rounds toward negative infinity, which produces the correct result. See Listing 108. Remember that the Int function rounds towards negative infinity and returns a double.

Listing 108. Round toward negative infinity and convert it to a Date.

```
Function SafeDateValue(v) Date
    SafeDateValue = CDate(Int(CDate(v)))
End Function
```


SafeDateValue in Listing 108 corrects the wrong behavior. Listing 109 repeats Listing 107 using SafeDateValue, so now the correct values are obtained.

Listing 109. *Round toward negative infinity and convert it to a Date.*

```
Sub SafeDateTimeBehavior
    On Error GoTo Oops:
    Dim Dates()
    Dim i As Integer
    Dim s As String
    Dates() = Array("10/04/1582 00:00:00", "10/04/1582 06:00:00", _
        "10/05/1582 00:00:00", "10/05/1582 06:00:00", _
        "10/15/1582 00:00:00", "10/15/1582 06:00:00", _
        "12/28/1899 00:00:00", "12/28/1899 06:00:00", _
        "12/29/1899 00:00:00", "12/29/1899 06:00:00", _
        "12/30/1899 00:00:00", "12/30/1899 06:00:00", _
        "12/31/1899 00:00:00", "12/31/1899 06:00:00", _
        "01/01/1900 00:00:00", "01/01/1900 06:00:00", _
        "01/02/1900 00:00:00", "01/02/1900 06:00:00", _
        "1/1/1 00:00:00", "1/1/1 06:00:00", "1/1/1 12:00:00" _
    )
    For i = LBound(Dates()) To UBound(Dates())
        s = s & CStr(i) & "    " & Dates(i) & " => "
        s = s & CDb1(SafeDateValue(CDate(Dates(i))))
        s = s & " => "
        s = s & CDb1(CDate(Dates(i))) & " => " & CDateToISO(SafeDateValue(Dates(i)))
        s = s & CHR$(10)
    Next
    MsgBox s, 0, "Strange Time Values"
Exit Sub
Oops:
    s = s & " Error"
    Resume Next
End Sub
```

6.7. Extract each part of a date

Date objects are based on floating-point Double numbers so mathematical operations and comparisons can be used with Date objects. The Date and Time functions, however, return strings, so they can't be used in this capacity. Basic provides functions to retrieve the individual pieces of a date (see Table 41).

Table 41. Date component extraction functions in Basic.

Function	Description
Year(date)	Return the year portion of a Date value as an Integer.
Month(date)	Return the month portion of a Date value as an Integer.
Day(date)	Return the day portion of a Date value as an Integer.
Hour(date)	Return the hour portion of a Date value as an Integer.
Minute(date)	Return the minutes portion of a Date value as an Integer.
Second(date)	Return the seconds portion of a Date value as an Integer.
WeekDay(date)	Return an integer value from 1 through 7, corresponding to the day of the week, Sunday through Saturday.

The functions in Table 41 all expect a Date object, which is internally based on a Double. Basic automatically converts the argument to the appropriate type if possible. The Date function returns a string with no time information (everything to the right of the decimal is zero) so there is no time information for the Hour, Minute, and Second functions to return. Similarly, the Time function returns a string with no date information (everything to the left of the decimal is zero), which corresponds to December 30, 1899.

```
Print "Year = " & Year(0.223)      '1899, 0 for date means December 30, 1899
Print "Year = " & Year(Time)       '1899, No date information from Time()
Print "Month = " & Month(Date)     'Current month
Print "Day = " & Day(Now)          'Now contains date and time information
Print "Hour = " & Hour(Date)       '0, No time information from Date()
Print "Minutes = " & Minute(Now)   'Current minutes
Print "Seconds = " & Second(Time)  'Current seconds
```

Use the WeekDay function to determine the day of the week. Some calendars start on Monday and some start on Sunday; Basic assumes that Sunday is the first day of the week. See Listing 110.

Listing 110. Determine the day of the week.

```
Sub ExampleWeekDayText
    Print "Today is " & WeekDayText(Date)
End Sub

Function WeekDayText(d) As String
    Select Case WeekDay(d)
        case 1
            WeekDayText="Sunday"
        case 2
            WeekDayText="Monday"
        case 3
            WeekDayText="Tuesday"
        case 4
            WeekDayText="Wednesday"
        case 5
            WeekDayText="Thursday"
        case 6
            WeekDayText="Friday"
        case 7
```

```

        WeekDayText="Saturday"
    End Select
End Function

```

The DatePart function allows you to pass a string expression that determines the part of the date of interest; as expressed as a string as the first argument. Running DatePart with September 15, 2010 at 19:13:20 yields the results shown in Table 42.

Table 42. DatePart string specifier.

Format	Description	Result
yyyy	Four-digit year	2010
q	Quarter	3
m	Month	9
y	Day of year	258
w	Weekday	4
ww	Week of year	38
d	Day of Month	15
h	Hour	19
n	Minute	13
s	Second	20

Listing 111. Use DatePart to extract components of a date.

```

Sub ExampleDatePart
    Dim TheDate As Date
    Dim f
    Dim i As Integer
    Dim s$
    TheDate = Now
    f = Array("yyyy", "q", "m", "y", "w", "ww", "d", "h", "n", "s")
    s = "Now = " & TheDate & CHR$(10)
    For i = LBound(f) To UBound(f)
        s = s & "DatePart(" & f(i) & ", " & TheDate & ") = " & _
            DatePart(f(i), TheDate) & CHR$(10)
    Next
    MsgBox s
End Sub

```

TIP

A German user reported that DatePart failed if it were not run with CompatibilityMode(True). In the latest US/English build, MonthName only works with CompatibilityMode(True)

DatePart supports an optional third argument that specifies when a week is assumed to start (see Table 43). The optional fourth argument, specifies when the year is assumed to start. The third and fourth arguments affect things such as the week of the year, but will not affect other values such as the day of the year.

Table 43. *DatePart week start and year start values.*

Value	Week Start Description	Year Start Description
0	Use system default value.	Use system default value.
1	Sunday (default)	Week 1 is the week with January first (default).
2	Monday	Week 1 is the first week containing four or more days of that year.
3	Tuesday	Week 1 is the first week containing only days of the new year.
4	Wednesday	
5	Thursday	
6	Friday	
7	Saturday	

After extracting portions of the date, it is useful to print the values in an easy to read format. Use `MonthName` to convert the month number to the month name. Setting the optional second argument to `True` causes the name to be abbreviated.

Listing 112. *Print the Month as a string.*

```
Sub ExampleMonthName
    Dim i%
    Dim s$
    CompatibilityMode(True)
    For i = 1 To 12
        s = s & i & " = " & MonthName(i, True) & " = " & MonthName(i) & CHR$(10)
    Next
    CompatibilityMode(False)
    MsgBox s, 0, "MonthName"
End Sub
```

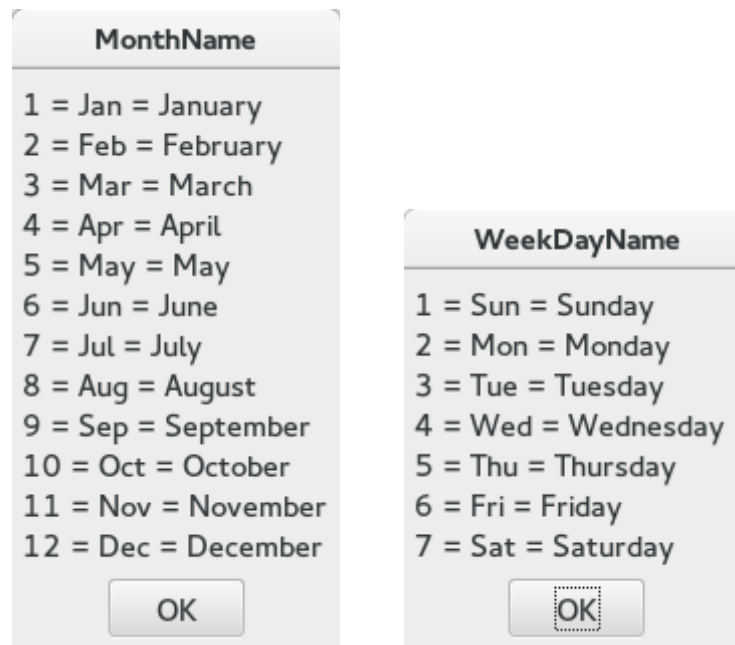


Figure 48. Use *MonthName* and *WeekDayName* to convert an integer into the name. *WeekDayName*, similar to *Monthname*, returns the day of the week as a string. Oddly, *WeekDayName* only works if compatibility mode is on. *WeekDayName* also supports a third argument, which specifies the day that the week begins (see Table 44).

Listing 113. Print the day of the week as a string.

```
Sub ExampleWeekDayName
    Dim i%
    Dim s$
    CompatibilityMode(True)
    For i = 1 To 7
        s = s & i & " = " & WeekDayName(i, True) & " = " & WeekDayName(i) & CHR$(10)
    Next
    MsgBox s, 0, "WeekDayName"
End Sub
```

Table 44. Third argument for *WeekDayName* to specify the first day of the week.

Value	Description
0	Use National Language Support API setting.
1	Sunday (default)
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday
7	Saturday

Use the WeekDay function to extract the day of the week based on a date. The optional second argument specifies the day on which the week starts; the values are the same as for WeekDayName (see Table 44) except that 0 corresponds to the system default setting.

```
Print WeekDay(Now)
```

Use FormatDateTime to display the date and time in a common format. The first argument is the date. The second argument is optional and specifies how the date should be formatted (see Table 45).

Listing 114. Format a date/time string.

```
Sub ExampleFormatDateTime
    Dim s$, i%
    Dim d As Date
    d = Now
    CompatibilityMode(True)
    s = "FormatDateTime(d) = " & FormatDateTime(d) & CHR$(10)
    For i=0 To 4
        s = s & "FormatDateTime(d, " & i & ") = " & FormatDateTime(d, i) & CHR$(10)
    Next
    MsgBox s, 0, "FormatDateTime"
End Sub
```

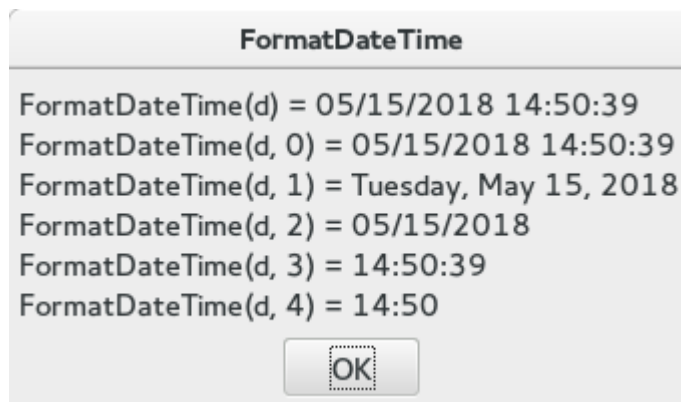


Figure 49. Format a date and time with FormatDateTime.

Table 45. Second argument to FormatDateTime specify format.

Value	Description
0	Default format with a short date and and a long time.
1	Long date format with no time.
2	Short date format.
3	Time in the computer's regional settings.
4	24 hour hours and minutes as hh:mm.

6.8. Date arithmetic

Internally, a date is represented as a floating point number. The decimal portion represents the time, and the integer portion represents the date. The representation allows mathematical manipulation, but, it requires a bit of thought; for example, add:

- 1 Add one day `Now + 1`
- 1/24 Add one hour `Now + 1/24`
- 365 Add one year, but leap years are a problem.

Use `DateAdd` to simplify the process. The third argument is the date. The first argument (see Table 42) determines how the value in the second argument is interpreted. I can repeat the list above as:

```
Print DateAdd("d", 1, Now)        'Add one day.
Print DateAdd("h", 1, Now)        'Add one hour.
Print DateAdd("yy", 1, Now)       'Add one year.
```

Use `DateDiff` to determine the number of “intervals” between two dates; for example, the number weeks between two dates. The first argument specifies the interval to use (see Table 42). The second argument is the first date, and the third argument is the second date. The optional fourth and fifth arguments specify the first day of the week and the first week of the year (see Table 43).

```
Print DateDiff("yyyy", "03/13/1965", Date(Now))    'Years from March 13, 1965 to now
Print DateDiff("d", "03/13/1965", Date(Now))       'Days from March 13, 1965 to now
```

6.9. Assembling dates from components

The functions `Hour`, `Minute`, `Second`, `Year`, `Month`, and `Day` are used to break a date into parts. The functions `DateSerial` and `TimeSerial` are used to put the dates back together again. The function `DateSerial` creates a `Date` object from the year, month, and day. The function `TimeSerial` creates a `Date` object from the hours, minutes, and seconds.

```
Print DateSerial(2003, 10, 1) '10/01/2003
Print TimeSerial(13, 4, 45)   '13:04:45
```

The first argument to the `DateSerial` function is the year, the second argument is the month, and the final argument is the day. If the month or day aren’t valid values, a run-time error occurs. Year values greater than 100 are used directly. Year values less than 100, however, have 1900 added to them. Two-digit years are mapped to the years 1900 and later (see Listing 115).

Listing 115. *DateSerial adds 1900 to years earlier than 100.*

```
Sub ExampleDateSerial
    On Error Goto OOPS
    Dim x
    Dim i%
    Dim s$

    x = Array(2003, 10, 1, _
              1899, 12, 31, _
              1899, 12, 30, _
```

```

1899, 12, 29, _
1899, 12, 28, _
99, 10, 1, _
3, 10, 1, _
0, 1, 1, _
-3, 10, 1, _
-99, 10, 1, _
-100, 10, 1, _
-1800, 10, 1, _
-1801, 10, 1)
i = LBound(x)
Do While i < UBound(x)
    s = s & DateSerial(x(i), x(i+1), x(i+2))
    s = s & " <= (" & ToStringWithLen(x(i), 4) & ", "
    s = s & ToStringWithLen(x(i+1), 3) & ", "
    s = s & ToStringWithLen(x(i+2), 3) & ") "
    s = s & CHR$(10)
    i = i + 3
Loop
MsgBox s
Exit Sub
OOPS:
    s = s & ERROR
Resume Next
End Sub

```

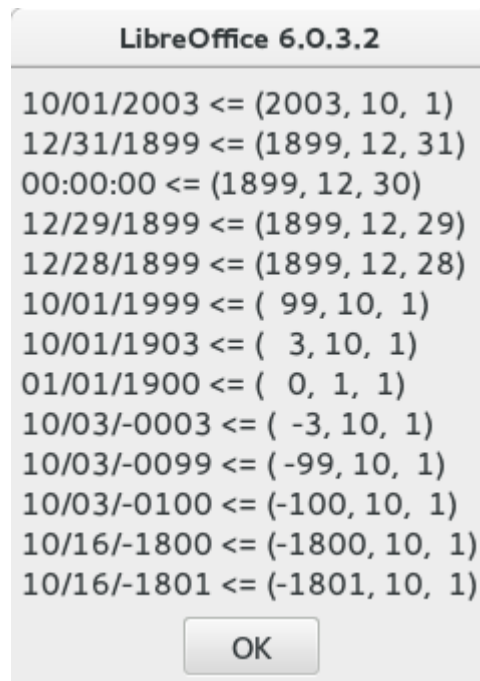


Figure 50. DateSerial creates a date from a year, month, and day value.

TIP

DateSerial adds 1900 to years earlier than 100, including negative years. An error is generated if the resulting year is still lower than 100. This allows years from -1800 through -1, which should probably be an error. The Date type is able to handle dates prior to 1/1/00, but obviously, DateSerial cannot.

Dates are printed based on the Gregorian calendar, but when the component parts are extracted, they are based on the Julian date.

Listing 116. *DateSerial accepts Gregorian dates before 10/15/1582.*

```
Print DateSerial(1582, 10, 15)      '10/15/1582
Print DateSerial(1582, 10, 14)      '10/04/1582
Print Day(DateSerial(1582, 10, 14)) '14
```

6.10. Measuring elapsed time over short intervals

You can obtain simple elapsed time by subtracting two date values. For example, `CLng(Now - CDate("1/1/2000"))` determines the number of days elapsed since January 1, 2000. Basic supports returning elapsed time as the number of seconds (Timer) and as the number of system ticks (GetSystemTicks). See Table 46. Internally, computers have a system timer that advances at a certain rate; the rate is hardware dependent. On Intel-based computers this value is 1/17th of a second. Each time that the timer advances by 1, it's called a "system tick." The number returned by GetSystemTicks is always based on milliseconds, even if it relies on a less accurate clock.

Table 46. *Elapsed time functions in Basic.*

Function	Description
GetSystemTicks	Return the number of system ticks as a Long. The reported time is always in milliseconds, even if the underlying timer is less accurate.
Timer	Return the number of seconds since midnight as a Date. Cast this to a Long.

Use GetSystemTicks to get the system-dependent number of system timer ticks. This value is typically used to time internal operations because it has higher precision than the internal time and date functions (see Listing 117 and Figure 51). The return value is a Long.

Listing 117. *Measuring elapsed time.*

```
Sub ExampleElapsedTime
    Dim StartTicks As Long
    Dim EndTicks As Long
    Dim StartTime As Date
    Dim EndTime As Date
    StartTicks = GetSystemTicks()
    StartTime = Timer
    Wait(200) 'Pause execution for 0.2 seconds
    EndTicks = GetSystemTicks()
    EndTime = Timer
    MsgBox "After waiting 200 ms (0.2 seconds), " & Chr$(10) & _
        "System ticks = " & CStr(EndTicks - StartTicks) & Chr$(10) & _
        "Time elapsed = " & CStr((EndTime - StartTime)) & _
        " seconds" & Chr$(10), 0, "Elapsed Time"
End Sub
```

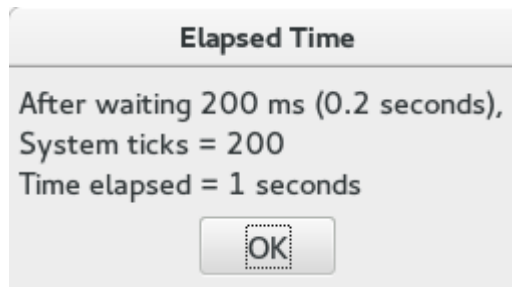


Figure 51. *GetSystemTicks has better resolution than Now.*

Run the macro in Listing 117 a few times. Sometimes it shows 0 seconds, and sometimes it shows 1 second. The resolution is too large for short duration items.

The Timer function returns the number of seconds since midnight as a Date object. The problem with this is that at 10 seconds past midnight, the return value is 10. The Date object, however, interprets 10 to mean “10 days”. Cast the returned type directly to a numerical type using CLng or CDBl to obtain the number of elapsed seconds.

```
Dim nSeconds As Long
nSeconds = Timer
Print "Number of seconds = " & nSeconds
Print "Number of seconds = " & CLng(Timer)
```

TIP The Timer function returns the number of seconds since midnight. Using Timer to determine elapsed time for spans that start before midnight and end after midnight produces useless results.

6.11. How fast does this run? A real-world example!

The greatest common divisor (GCD) of two integers is the largest integer that divides both integers with no remainder. For example, the GCD of 6 and 9 is 3. The numbers 1 and 3 both divide the numbers 6 and 9 (see Table 47). The greatest of these is 3.

Table 47. *Dividing 6 and 9 by integers.*

Number	6 Divided By #	9 Divided By #	Both Divide
1	6	9	Yes
2	3	4 remainder 1	No
3	2	3	Yes
4	1 remainder 2	2 remainder 1	No
5	1 remainder 1	1 remainder 4	No
6	1	1 remainder 3	No
7	0 remainder 6	1 remainder 2	No
8	0 remainder 6	1 remainder 1	No
9	0 remainder 6	1	No

This example begins around the year 300 B.C. with a guy living in ancient Greece named Euclid. Euclid was a pretty smart guy who wrote numerous books, including *Data*, concerning the solution of problems through geometric analysis, *On Divisions (of Figures)*, the *Optics*, the *Phenomena*, a paper on spherical geometry for astronomers, the *Elements*, a 13-volume textbook on geometry, and several lost works on higher geometry. His impact on society was huge. One of his most well-known contributions is an extremely efficient algorithm to solve the GCD problem. Now jump ahead a few thousand years to Olivier Bietzer, who noticed that I had an impractically slow algorithm for solving the GCD problem. Olivier, who certainly knows a lot about these things, wrote the macro in Listing 118 that solves the GCD problem using Euclid's algorithm, and sent it to me.

Listing 118. Calculate the GCD.

```
'Author: Olivier Bietzer
'e-mail: olivier.bietzer@laposte.net
'This uses Euclid's algorithm and it is very fast!
Function GCD_1(ByVal x As Long, ByVal y As Long) As Long
    Dim pgcd As Long, test As Long

    ' We must have x >= y and positive values
    x=abs(x) : y=abs(y)
    If (x < y) Then
        test = x : x = y : y = test
    End If
    If y = 0 Then Exit Function

    ' Euclid says ....
    pgcd = y          ' by definition, PGCD is the smallest
    test = x MOD y    ' remainder after division
    Do While (test)   ' While not 0
        pgcd = test   ' pgcd is the remainder
        x = y         ' x,y and current pgcd permutation
        y = pgcd
        test = x MOD y ' test again
    Loop
    GCD_1 = pgcd      ' pgcd is the last non 0 rest ! Magic ...
End Function
```

In general, the best way to speed up a solution to a computational problem is by using a better algorithm. The algorithm in Listing 118 runs roughly 1000 times faster than the routine that I had. If a faster algorithm is not available, you can look for other ways to improve performance. (Sometimes it's possible to invent a wholly new and improved algorithm; but that is quite a trick! If you succeed in developing a new, faster algorithm for a widely known problem, you may have great career potential as a professional mathematics or computer science professor.) The code in Listing 118 is already pretty lean. There isn't a whole lot to remove, but I thought that I could reduce the number of assignments (see Listing 119).

Listing 119. Calculate the GCD (another way).

```
Function GCD_2(ByVal x As Long, ByVal y As Long) As Long
```

```

Dim pgcd As Long, test As Long

' We must have x >= y and positive values
x=abs(x) : y=abs(y)
If (x < y) Then
    test = x : x = y : y = test
End If
If y = 0 Then Exit Function

Do While (y)      ' While not 0
    pgcd = y      ' pgcd is the remainder
    y = x MOD pgcd ' test again
    x = pgcd      ' x,y and current pgcd permutation
Loop
GCD_2 = pgcd      ' pgcd is the last non 0 remainder ! Magic ...
End Function

```

Now the question is, which function is faster? If you use a stopwatch to see how quickly I can blink, the results aren't likely to be very accurate because of measurement errors. It's much easier to tell me to blink as many times as I can in four seconds or to time how quickly I can blink 50 times. The code in Listing 120 does something similar. It sits in a tight loop and calls each GCD implementation 5000 times. I want to know how long it takes to call the GCD function 5000 times, but I'm actually timing how long it takes to loop 5000 times, generate 10,000 random numbers, and call the GCD function 5000 times. To compensate for this, the amount of time required to loop 5000 times and generate 10,000 random numbers is measured.

TIP The test program takes a few seconds to run, so, be patient.

Listing 120. Time the two different GCD methods.

```

Sub testGCD
    Dim nStartTicks As Long      'When I started timing
    Dim nEndTicks As Long       'When I stopped timing
    Dim nLoopTicks As Long      'Ticks to do only the loop
    Dim nGCD_1_Ticks As Long    'Ticks for GCD_1
    Dim nGCD_2_Ticks As Long    'Ticks for GCD_2
    Dim nMinIts As Long         'Number of iterations
    Dim x&, y&, i&, n&         'Temporary long numbers
    Dim s$                      'Hold the output string

    nMinIts = 5000              'Set the number of iterations
    Randomize(2)                'Set to a known state
    nStartTicks = GetSystemTicks() 'Start ticks
    For i& = 1 To nMinIts       'Control the number of iterations
        x = 10000 * Rnd()       'Generate the random data
        y = 10000 * Rnd()       'Generate the random data
    Next
    nEndTicks = GetSystemTicks()
    nLoopTicks = nEndTicks - nStartTicks

    Randomize(2)                'Set to a known state

```

```

nStartTicks = GetSystemTicks() 'Start ticks
For i& = 1 To nMinIts           'Control the number of iterations
    x = 10000 * Rnd()           'Generate the random data
    y = 10000 * Rnd()           'Generate the random data
    GCD_1(x, y)                 'Do the work we really care about
Next
nEndTicks = GetSystemTicks()
nGCD_1_Ticks = nEndTicks - nStartTicks - nLoopTicks

Randomize(2)                   'Set to a known state
nStartTicks = GetSystemTicks() 'Start ticks
For i& = 1 To nMinIts           'Control the number of iterations
    x = 10000 * Rnd()           'Generate the random data
    y = 10000 * Rnd()           'Generate the random data
    GCD_2(x, y)                 'Do the work we really care about
Next
nEndTicks = GetSystemTicks()
nGCD_2_Ticks = nEndTicks - nStartTicks - nLoopTicks

s = "Looping " & nMinIts & " iterations takes " & nLoopTicks & _
    " ticks" & CHR$(10) & _
    "Calling GCD_1 takes " & nGCD_1_Ticks & " ticks or " & _
    Format(nMinIts * 100 / nGCD_1_Ticks, "####00.00") & _
    " Iterations per second" & CHR$(10) & _
    "Calling GCD_2 takes " & nGCD_2_Ticks & " ticks or " & _
    Format(nMinIts * 100 / nGCD_2_Ticks, "####00.00") & _
    " Iterations per second"

MsgBox s, 0, "Compare GCD"
End Sub

```

One problem in writing timing routines is determining how many iterations to do. I frequently use computers of different speeds. The results in Figure 52 are based on my home computer running the macro in Listing 120. The macro in Listing 120 makes a specific number of iterations. Sometimes I use a solution that limits the iterations based on time rather than number of iterations. This complicates the measurement of overhead and is left as an interesting, but not overly difficult, problem for the reader.



Figure 52. The improvement is about 10 percent.

6.12. Long time intervals and special dates

It's easy to obtain elapsed time over long intervals by subtracting date values. To determine precise dates and intervals, you can creatively use the component pieces. For example, given the date, what is the first day of the month? This is easy because the first day of every month is day 1. Use the functions Year and Month to extract the year and month, and then reassemble the date using DateSerial and set the day to 1. The sample macro also calls WeekDayText shown in Listing 110.

Listing 121. First day of the month.

```
Function FirstDayOfMonth(d As Date) As Date
    FirstDayOfMonth() = DateSerial(Year(d), Month(d), 1)
End Function

Sub FirstDayOfThisMonth()
    Dim d As Date
    d = FirstDayOfMonth(Now())
    MsgBox "First day of this month (" & d & ") is a " & WeekDayText(d)
End Sub
```

To find the last day of a month, find the first day of the next month and then subtract 1 from the number. If the current month is December, set the month to January and increment the year by 1.

Listing 122. Last day of the month.

```
Function LastDayOfMonth(d As Date) As Date
    Dim nYear As Integer
    Dim nMonth As Integer
    nYear = Year(d)           'Current year
    nMonth = Month(d) + 1     'Next month, unless it was December.
    If nMonth > 12 Then        'If it is December then nMonth is now 13
        nMonth = 1           'Roll the month back to 1
        nYear = nYear + 1     'but increment the year
    End If
    LastDayOfMonth = CDate(DateSerial(nYear, nMonth, 1)-1)
End Function

Sub LastDayOfThisMonth()
    Dim d As Date
    d = LastDayOfMonth(Now())
    MsgBox "Last day of this month (" & d & ") is a " & WeekDayText(d)
End Sub
```

It's easy to find the first day of the year for any given date; it's always January 1 of that year. Use the Year function to obtain the current year and then set the day and month each equal to 1. Finding the last day of the year for any given date is only marginally more difficult. First, find the first day of the next year by incrementing the year by 1 and setting the month and day equal to 1. Subtracting 1 from the first day of next year provides the last day of this year.

```
d = Now
Print DateSerial(Year(d), 1, 1)           '01/01/2003
Print CDate(DateSerial(Year(d)+1, 1, 1)-1) '12/31/2003
```

Use the WeekDay function to find the first and last days of a week. Subtract the day of the week and add 1 to take the date to Sunday at the beginning of the current week.

```
d = Date
Print CDate(CDb1(d) - WeekDay(d) + 1) '8/10/2003 is a Sunday
Print CDate(CDb1(d) - WeekDay(d) + 7) '8/16/2003 is a Saturday
```

You can use similar date manipulations to solve other date-related problems, such as determining the work week, how many days until your anniversary, or the age of a person in years, months, and days.

6.13. Conclusion

Although dates in LO Basic are straightforward and easy to use, you must take care with dates prior to October 15, 1582. The jump from the Gregorian and Julian calendar also may potentially cause unexpected problems. Also take care when the underlying implementation, a Double, becomes a negative number; this happens around December 30, 1899. This chapter also discussed methods of timing routines and determining specific dates.

7. String Routines

This chapter introduces the subroutines and functions supported by OpenOffice.org Basic that are related to strings. This includes functions to manipulate strings, convert other data types to strings, and to perform special formatting.

Text data is stored in strings as a sequence of 16-bit unsigned integer Unicode version 2.0 values. The Unicode Worldwide Character Standard is a set of binary codes representing textual or script characters designed because ASCII, the original standard, can handle only 256 distinct characters. The first 128 characters (numbered 0 through 127) correspond to the letters and symbols on a standard U.S. keyboard. The next 128 characters (numbered 128 through 255) consist of special characters such as accent marks, Latin-based characters, and a few symbols. The remaining 65,280 values — of which only about 34,000 are currently used — are used for a wide variety of worldwide text characters, mathematical symbols, accent marks (diacritics), and technical symbols.

OpenOffice.org has a large number of functions that allow you to manipulate strings. These string-manipulation operations range from converting uppercase to lowercase (or vice versa) to selecting substrings out of a longer string. The functions listed in Table 48 are the string functions covered in this chapter. The functions listed in Table 49 are related to strings as well as either numerical or array manipulations; these are covered in other chapters.

Table 48. *These string-related functions are covered in this section.*

Function	Description
ASC(str)	Return the integer ASCII value of the first character in the string. This supports 16-bit Unicode values as well.
CHR(n)	Convert an ASCII number to a character.
CStr(obj)	Convert standard types to a string.
Format(obj, format)	Fancy formatting; works only for strings.
Hex(n)	Return the hexadecimal representation of a number as a string.
InStr(str, str) InStr(start, str, str) InStr(start, str, str, mode)	Attempt to find string 2 in string 1. Returns 0 if not found and starting location if it is found. The optional start argument indicates where to start looking. The default value for mode is 1 (case-insensitive comparison). Setting mode to 0 performs a case-sensitive comparison.
InStrRev(str, find, start, mode)	Return the position of the first occurrence of one string within another, starting from the right side of the string. Only available with “Option VBASupport 1”. Start and mode are optional.
Join(s()) Join(s(), str)	Concatenate the array elements, separated by the optional string delimiter, and return the value as a string. The default delimiter is a single space. Inverse of the Split function.
LCase(str)	Return a lowercase copy of the string.
Left(str, n)	Return the leftmost n characters from the string.
Len(str)	Return the length of the string.
LSet str1 = str2	Left-justify a string into the space taken by another string.
LTRim(str)	Return a copy of the string with all leading spaces removed.
Mid(str, start)	Return the substring, starting at start. If the length is omitted, the entire end of the string is returned. If the final string argument is included, this replaces the specified portion of the first

Function	Description
Mid(str, start, len) Mid(str, start, len, str)	string with the last string.
Oct(n)	Return the octal representation of a number as a string.
Replace(str, find, repl, start, count, mode)	Search str for find and replace it with repl. Optionally, specify the start, count, and mode.
Right(str, n)	Return the rightmost n characters.
RSet str1 = str2	Right-justify a string into the space taken by another string.
RTrim(str)	Return a copy of the string with all trailing spaces removed.
Space(n)	Return a string with the number of specified spaces.
Split(str) Split(str, str)	Split a string into an array based on an optional delimiter. Inverse of the Join function.
Str(n)	Convert a number to a string with no localization.
StrComp(s1, s2) StrComp(s1, s2, mode)	Compare two strings returning -1, 0, or 1 if the first string is less than, equal to, or greater than the second in alphabetical order. Set the optional third argument to zero for a case-insensitive comparison. The default is 1 for a case-sensitive comparison.
StrConv(str, mode[, local])	Converts a string based on the mode: 1=upper, 2=lower, 4=wide, 8=narrow, 16=Katakana, 32=Hiragana, 64=to unicode, 128=from unicode.
String(n, char) String(n, ascii)	Return a string with a single character repeated multiple times. The first argument is the number of times to repeat; the second argument is the character or ASCII value.
StrReverse	Reverse a string. Must use “Option VBASupport 1”, or precede it with CompatibilityMode(True).
Trim(str)	Return a copy of the string with all leading and trailing spaces removed.
UCase(str)	Return an uppercase copy of the string.
Val(str)	Convert a string to a double. This is very tolerant to non-numeric text.

The subroutines and functions related to string handling in Basic are listed in Table 48. Some of these functions (see Table 49) have in-depth coverage in other chapters, because they are directly related to the content in those chapters. They are covered briefly near the end of this chapter, in the section titled “Converting data to a string.”

Table 49. *These string-related functions are covered in other chapters.*

Function	Covered In	Description
Join(s()) Join(s(), str)	5 Array Routines	Concatenate the array elements, separated by the optional string delimiter, and return the value as a string.
Split(str) Split(str, str)	5 Array Routines	Split a string into an array based on an optional delimiter.
CStr(obj)	4 Numerical Routines	Convert standard types to a string.
Str(n)	4 Numerical Routines	Convert a number to a string with no localization.
Hex(n)	4 Numerical Routines	Return the hexadecimal representation of a number as a string.
Oct(n)	4 Numerical Routines	Return the octal representation of a number as a string.

Function	Covered In	Description
Val(str)	4 Numerical Routines	Convert a string to a double. This is very tolerant to non-numeric text.

7.1. ASCII and Unicode values

In the early days of computers there were different types of data-processing equipment, and there was no common method of representing text. To alleviate this problem, the American National Standards Institute (ANSI) proposed the American Standard Code for Information Interchange (ASCII) in 1963. The standard was finalized in 1968 as a mapping of 128 characters, numbers, punctuation, and control codes to the numbers from 0 to 127 (see Table 50). The computer-minded reader may notice that this requires 7 bits and does not use an entire byte.

Table 50. *The original 128 ASCII characters.*

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Table 50 lists the original 127 ASCII characters. The top row and the left column are used to identify the hexadecimal ASCII value. For example, the capital letter A has an ASCII value of 41 in hexadecimal format, and z has an ASCII value of 5A. If more than one letter occupies a box, that value represents a special command character (see Table 51). Some of these special commands are designed for communications, some for file formats, and some are even available on the keyboard.

Table 51. *Non-printable ASCII characters.*

Hex	DEC	Symbol	Description
00	0	NUL	Null, usually signifying nothing
01	1	SOH	Start of heading
02	2	STX	Start of text
03	3	ETX	End of text
04	4	EOT	End of transmission — not the same as ETB
05	5	ENQ	Enquiry
06	6	ACK	Acknowledge — I am here or data successfully received
07	7	BEL	Bell — Causes teletype machines and many terminals to ring a bell
08	8	BS	Backspace — Moves the cursor or print head backward (left) one space
09	9	TAB	Horizontal tab — Moves the cursor (or print head) right to the next tab

Hex	DEC	Symbol	Description
			stop
0A	10	LF	Line feed or new line — Moves the cursor (or print head) to a new line
0B	11	VT	Vertical tab
0C	12	FF	Form feed — Advances paper to the top of the next page
0D	13	CR	Carriage return — Moves the cursor (or print head) to the left margin
0E	14	SO	Shift out — Switches the output device to an alternate character set
0F	15	SI	Shift in — Switches the output device back to the default character set
10	16	DLE	Data link escape
11	17	DC1	Device control 1
12	18	DC2	Device control 2
13	19	DC3	Device control 3
14	20	DC4	Device control 4
15	21	NAK	Negative acknowledge
16	22	SYN	Synchronous idle
17	23	ETB	End of transmission block — not the same as EOT
18	24	CAN	Cancel
19	25	EM	End of medium
1A	26	SUB	Substitute
1B	27	ESC	Escape — This is the Esc key on your keyboard
1C	28	FS	File separator
1D	29	GS	Group separator
1E	30	RS	Record separator
1F	31	US	Unit separator
7F	127	DEL	Delete — This is the Del key on your keyboard

For most computers, the smallest easily stored and retrieved piece of data is a byte, which is composed of 8 bits. The characters in Table 50 require only 7 bits. To avoid wasting space, the Extended ASCII characters were introduced; these used the numbers 128 through 255. Although these characters introduce special, mathematical, graphic, and foreign characters, it just wasn't enough to satisfy international use. Around 1986, Xerox started working to extend the character set to work with Asian characters. This work eventually led to the current Unicode set, which uses 16-bit integers and allows for 65,536 unique characters.

OOo stores characters as 16-bit unsigned integer Unicode values. The ASC and CHR functions convert between the integer value and the character value, for example, between 65 and A. Use the ASC function to determine the numerical ASCII value of the first character in a string. The return value is a 16-bit integer allowing for Unicode values. Only the first character in the string is used;

the rest of the characters are ignored. A run-time error occurs if the string has zero length. This is essentially the inverse of the CHR\$ function, which converts the number back into a character.

TIP The CHR function is frequently written as CHR\$. In Visual Basic, CHR\$ returns a string and can't handle null input values, and CHR returns a variant that's able to accept and propagate null values. In Basic, they are the same; they both return strings and they both generate a run-time error with a null input value.

Use the CHR function to convert a 16-bit ASCII value to the character that it represents. This is useful when you want to insert special characters into a string. For example, CHR(10) is the new-line character. The CHR function is the inverse of the ASC function. Although the ASC function returns the Unicode numbers, these numbers are frequently generically referred to as “the ASCII value.” Strictly speaking, this is incorrect, but it's a widely used slang expression. The numbers correspond directly to the ASCII values for the numbers 0 through 255, and having used the terminology for years, programmers aren't likely to stop. So, when you see the term “ASCII value” in this book, think “Unicode value.”

Listing 123. *Demonstrate new line.*

```
Sub ShowChrAsc
    Dim s$
    Print CHR$(65)           'A
    Print ASC("Andrew")     '65
    s = "1" & CHR$(10) & "2" 'New line between 1 and 2
    MsgBox s
End Sub
```

TIP Use the MsgBox statement to print strings that contain CHR\$(10) or CHR\$(13) — they both cause Basic to print a new line. The Print statement displays a new dialog for each new line. MsgBox, however, properly displays new lines in a single dialog.

While attempting to decipher the internal functions of OpenOffice.org, I frequently find strings that contain characters that aren't immediately visible, such as trailing spaces, new lines, and carriage returns. Converting the string to a sequence of ASCII characters simplifies the process of recognizing the true contents of the string. See Listing 124 and Figure 53.

Listing 124. *Convert a string to ASCII.*

```
Sub ExampleStringToASCII
    Dim s As String
    s = "AB"""" """"BA"
    MsgBox s & CHR$(10) & StringToASCII(s), 0, "String To ASCII"
End Sub

Function StringToASCII(sInput$) As String
    Dim s As String
    Dim i As Integer
    For i = 1 To Len(sInput$)
        s = s & CStr(ASC(Mid(sInput$, i, 1))) & " "
    Next
    StringToASCII = s
End Function
```

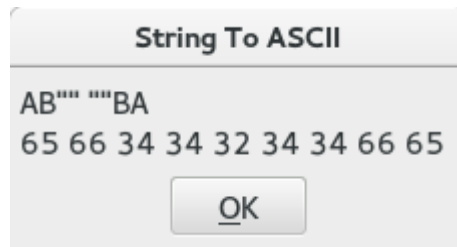


Figure 53. A string followed by its corresponding ASCII values: A=65, B=66, "=34, and so on ...

On more than one occasion, I needed to know exactly how OOo stored data in a text document. One common example is trying to manipulate new lines and new paragraphs in a manner not easily supported by regular expressions. The subroutine in Listing 125 displays the currently selected text as a string of ASCII values. The important thing to learn in this chapter is how to view the ASCII values associated with the text. This will show the characters used between paragraphs, for example. The methods to properly retrieve and manipulate selected text are covered later. To test this macro, select run the macro.

Listing 125. Display the selected text as ASCII characters.

```
Sub SelectedTextAsASCII()
    Dim vSelections           'Multiple disjointed selections
    Dim vSel                  'A single selection
    Dim vCursor               'OOo document cursor
    Dim i As Integer          'Index variable
    Dim s As String           'Temporary utility string variable
    Dim bIsSelected As Boolean 'Is any text selected?
    bIsSelected = True        'Assume that text is selected

    'The current selection in the current controller.
    'If there is no current controller, it returns NULL.
    'ThisComponent refers to the current document
    vSelections = ThisComponent.getCurrentSelection()
    If IsNull(vSelections) OR IsEmpty(vSelections) Then
        bIsSelected = False
    ElseIf vSelections.getCount() = 0 Then
        bIsSelected = False
    End If

    If NOT bIsSelected Then      'If nothing is selected then say so
        Print "Nothing is selected" 'and then exit the subroutine
        Exit Sub
    End If

    'The selections are numbered from zero
    'Print the ASCII values of each
    For i = 0 To vSelections.getCount() - 1
        vSel = vSelections.getByIndex(i)
        vCursor = ThisComponent.Text.CreateTextCursorByRange(vSel)
        s = vCursor.getString()
        If Len(s) > 0 Then
```

```

        MsgBox StringToASCII(vCursor.getString()), 0, "ASCII of Selection " & i
    ElseIf vSelections.getCount() = 1 Then
        Print "Nothing is selected"
    End If
Next
End Sub

```

7.2. Standard string functions

The comparison operators (=, <, <=, >, >=, and <>) work with strings and numbers performing a case-sensitive comparisons. This means that the strings “a” and “A” are not equal. The built-in StrComp function can compare strings with and without case sensitivity. The StrComp function, which defaults to a case-sensitive comparison, returns -1, 0, or 1 if the first string argument is less than, equal to, or greater than the second string argument. Set the optional third argument to zero for a case-insensitive comparison.

TIP Use StrComp(string1, string2, 0) to alphabetize strings using a locale based case-sensitive compare.

The following pseudo code shows a simplified version of how StrComp could work for a case-sensitive comparison. A case-insensitive comparison would convert both strings to all uppercase before performing the comparison; I have no idea if StrComp is locale specific.

```

Let s1 = string1
Let s2 = string2
Let min_len = minimum(Len(s1), Len(s2))
For i = 1 To min_len
    If ASC(Mid(s1, i, 1)) < ASC(Mid(s2, i, 1)) Then
        set return value to -1
        Exit Function
    End If
    If ASC(Mid(s1, i, 1)) > ASC(Mid(s2, i, 1)) Then
        set return value to 1
        Exit Function
    End If
Next
If Len(s1) < Len(s2) Then
    set return value to -1
    Exit Function
End If
If Len(s1) > Len(s2) Then
    set return value to 1
    Exit Function
End If
set return value to 0
Exit Function

```

The numerical Unicode value of the first character in the first string is compared to the numerical Unicode value of the first character in the second string. If the first character is numerically less than the second character, -1 is returned. If the first character is numerically greater than the second character, 1 is returned. If the first character in each string is the same, the second

character in each string is compared. If the corresponding numerical Unicode value of each character is the same and the strings are the same length, 0 is returned. If corresponding characters all match, but the strings are of different lengths, the shorter string is considered less than the longer string.

Listing 126. Demonstrate StrComp.

```
Print StrComp( "A", "AA") ' -1 because "A" < "AA"
Print StrComp("AA", "AA") ' 0 because "AA" = "AA"
Print StrComp("AA", "A") ' 1 because "AA" > "A"

Print StrComp( "a", "A") ' 1 because "a" > "A"
Print StrComp( "a", "A", 1) ' 1 because "a" > "A"
Print StrComp( "a", "A", 0) ' 0 because "a" = "A" if case is ignored
```

Use the UCase and LCase functions to return a copy of the string with all characters in uppercase or lowercase.

Listing 127. Demonstrate UCase and LCase.

```
S$ = "Las Vegas"
Print LCase(s) REM Returns "las vegas"
Print UCase(s) REM Returns "LAS VEGAS"
```

If numerous comparisons will be made, using LCase or UCase is sometimes faster than performing a case-insensitive comparison each time. And sometimes, it is simply easier to use.

```
If LCase(Right(sFileName, 3)) = "odt" Then
```

Use StrConv(string, mode, locale_id) to convert a string with more flexibility than the individual methods UCase and LCase. The supported modes (shown in Table 52) are bit values that can be used together; for example, 1+64=65 causes all characters to be converted to uppercase and to Unicode. The final argument, locale_id, is an optional integer locale identifier that is not currently supported (as of OOo version 3.2.1)

Table 52. Modes supported by the StrConv statement.

Mode	Description
0	No change.
1	Convert all characters to uppercase.
2	Convert all characters to lowercase.
4	Convert narrow (half-width) characters in the string to wide (full-width) characters.
8	Convert wide (full-width) characters in the string to narrow (half-width) characters.
16	Convert Hiragana characters in the string to Katakana characters.
32	Convert Katakana characters in the string to Hiragana characters.
64	Convert all characters to Unicode.
128	Convert all characters from Unicode.

Use the LTrim, RTrim, and Trim functions to return copies of a string with leading, trailing, or both leading and trailing spaces removed. I usually do this with data retrieved from files and databases, and directly from users. The original string is unchanged, as are all internal spaces. Some trim routines in other programming languages trim all sorts of invisible characters, such as carriage returns, new-line characters, and tabs. In Basic, only the space character with an ASCII value of 32 is trimmed.

Listing 128. Demonstrate LTrim and RTrim.

```
s = "  hello  world  "
Print "(" & LTrim(s) & ")" " '(hello  world  )
Print "(" & RTrim(s) & ")" " ' ( hello  world)
Print "(" & Trim(s) & ")" " '(hello  world)
```

Use the Len function to return the number of characters in the string. If the argument is not a string, it is converted to a string first. It's probably safer to use CStr to convert nonstring arguments to strings, rather than to rely on the automatic behavior. For example, types such as Byte will not convert automatically as expected. The value held in the Byte data type is treated as an ASCII value and converted to a single character. The CStr function avoids this problem.

Listing 129. Demonstrate Len.

```
Print Len("") '0
Print Len("1") '1
Print Len("123") '3
Print Len(12) '2 the number is converted to a string
```

To create a string with a single character repeated multiple times, use the String function. The first argument is an integer indicating how many times the character is repeated. Zero is a valid value for the first argument, returning an empty string. The second argument is the character to repeat. Just like the ASC function, the String function uses the first character from the string and ignores the rest. If the second argument is a number, it's treated as an ASCII value and the character is created from the number.

Listing 130. Demonstrate String.

```
Print String(2, 65) 'AA 65 is ASCII for A
Print String(2, "AB") 'AA Only the first character is used
Print ASC(String(2)) '0 Bug: Created string with two ASCII 0 characters
Print Len(Space(4)) '4 Four spaces
```

Use the function InStr to find where (and if) one string is contained inside another. The InStr function can take four arguments. The first argument is an optional integer that indicates the first character to check. This defaults to 1, the first character, if it is not included. InStr then searches the second argument to see if it contains the third argument. The fourth optional argument determines if the comparison is case sensitive (0) or case insensitive (1). The default search is case insensitive. You can't use the fourth argument unless you also use the first argument.

TIP

The StrComp function uses a 0 to indicate a case-insensitive comparison and a 1 — the default — to indicate a case-sensitive comparison. The InStr function, however, uses a 0 to indicate a case-sensitive comparison and a 1 — the default — to indicate a case-insensitive comparison. The only similarity is that the value 1 is the default.

Listing 131. Demonstrate InStr.

```
Print InStr("CBAABC", "abc")           '4 default to case insensitive
Print InStr(1, "CBAABC", "b")          '2 first argument is 1 by default
Print InStr(2, "CBAABC", "b")          '2 start with second character
Print InStr(3, "CBAABC", "b")          '5 start with third character
Print InStr(1, "CBAABC", "b", 0)        '0 case-sensitive comparison
Print InStr(1, "CBAABC", "b", 1)        '2 case-insensitive comparison
Print InStr(1, "CBAABC", "B", 0)        '2 case-sensitive comparison
```

InStrRev is only available while VB compatibility mode is true. The start location is argument 3, as opposed to the first argument in InStr. A start location of -1 means start at the right most character; I wish that -2 was the second character from the right, but it causes a run-time exception.

Listing 132. Demonstrate InStrRev.

```
Sub ExampleInStrRev
    CompatibilityMode(True)
    Print InStrRev("CBAABC", "ABC")      '4 default to case sensitive
    Print InStrRev("CBAABC", "abc")      '0 default to case sensitive
    Print InStrRev("CBAABC", "abc", -1, 1) '4 force case insensitive
    Print InStrRev("CBAABC", "B", 1)      '0 start with first character
    Print InStrRev("CBAABC", "B", 2)      '2 start with second character
    Print InStrRev("CBAABC", "B", -1)     '5 start with last character
    Print InStrRev("CBAABC", "B", 5)      '5 start with fifth character
    Print InStrRev("CBAABC", "B", 4)      '2 start with fourth character
    Print InStrRev("CBAABC", "b", -1, 0)  '0 case-sensitive comparison
    Print InStrRev("CBAABC", "b", -1, 1)  '5 case-insensitive comparison
    Print InStrRev("CBAABC", "B", -1, 0)  '5 case-sensitive comparison
End Sub
```

Prior to OOo 2.0, the return value from InStr is an integer, which is limited to values from -32,768 through 32,767. A string can be up to 65,535 characters in length, which causes problems when InStr searches large strings. Starting with OOo 2.0, however, the return type is a long.

Listing 133. Demonstrate InStr and a long string.

```
Dim s1 As String
s1 = String(44000, "*") & "XX"          'This string has 44002 characters.
Print InStr(s1, "XX")                    '44001
```

7.3. Locale and strings

Use **Tools | Options | Language Settings | Languages** to view the Locale used by OO. My Locale setting is “Default – English (USA)”. After changing the Locale, you must exit and restart for the changes to be used. The following macro returns different results based on the configured locale. Notice the difference when run as English – US and Turkish locales.

Listing 134. *Some string functions use Locale.*

```
Sub LocaleStringTests
    Dim s$
    ' compare Turkish dotless i with upper case i
    ' Turkish Locale returns 0, English returns -1
    s = "Compare dotless i with upper case i : " & StrComp("i", "I", 0) & CHR$(10)

    ' compare lower case i to upper case i
    ' English Locale returns 0, Turkish Locale returns 1.
    s = s & "Compare i with upper case i : " & StrComp("i", "I", 0) & CHR$(10)

    s = s & "Lower Case I = " & LCase("I") & CHR$(10)
    s = s & "Upper Case i = " & UCase("i")
    MsgBox s
End Sub
```

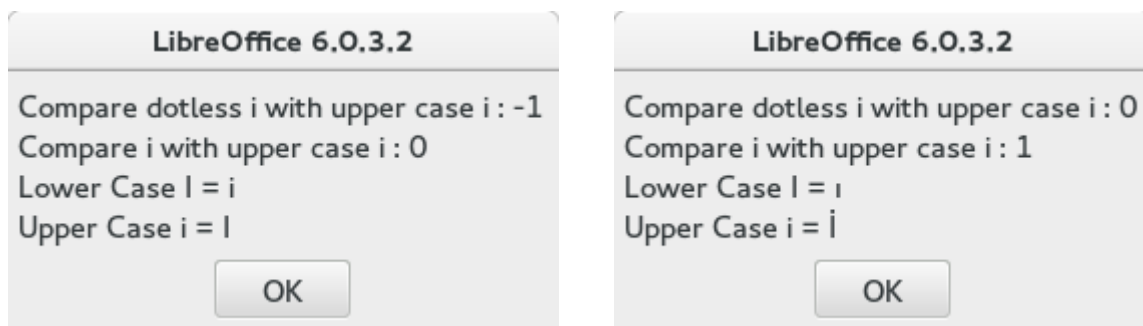


Figure 54. *English – US Locale on the Left, Turkish Locale on the Right.*

I did not explore other areas that may also rely on Locale specific settings.

7.4. Substrings

Use the Left function to retrieve a portion of the string from the start. The first argument is the string from which to extract the characters, and the second argument indicates how many characters to return. Analogously, the Right function returns strings from the end of a string. If the requested length is zero, an empty string is returned. If the requested length is too large, the entire string is returned.

```
Print Left("12345", 2) '12
Print Left("12345", 8) '12345
Print Right("12345", 2) '45
```

The length argument for the Left and Right functions is a long, correctly handling strings, which can contain up to 65,535 characters (see Listing 135).

Listing 135. *Strings can contain up to 2 billion characters.*

```
Dim s1 As String
s1 = String(44002, "*") 'This string has 44002 characters
Print Len(s1) '44002
Print Len(Left(s1, 44000)) '44000 (fixed in OOo 2.0)
```

```
Print Len(Right(s1, 44000)) '44000
```

Use the Mid function to extract arbitrary substrings and to replace substrings in an existing string. In general, the string functions return a new string rather than modifying the existing string. For example, the Trim function returns a new string with leading and trailing spaces removed, rather than removing leading and trailing spaces from the existing string. The Mid function, however, can be used to modify the string rather than simply returning a new one. In its simplest form, the Mid function has functionality to the Right function. The first argument is a string, and the second argument is a starting position. The optional third argument indicates the length of the string to return.

Listing 136. Demonstrate Mid.

```
Print Mid("123456", 3)      '3456
Print Mid("123456", 3, 2)   '34
s1 = String(44000, "*") & "XX"
Print Mid(s1, 44000)        '*XX      No problem with large arguments
Print Len(Mid(s1, 2, 40000)) '40000   No problem with large arguments
```

The Mid function can provide the same functionality as the Left function.

```
Left(s, n) = Mid(s, 1, n)
```

The Mid function takes an optional fourth argument, a string, that replaces the specified substring in the first argument. In other words, if four arguments are present, the first three arguments specify a substring in the string and the fourth argument replaces the substring. This may shorten the length of a string, but in Basic this will never cause the string to become longer. If the final argument is longer than the specified substring, only the first characters are used from the final argument to replace the substring.

Listing 137. Demonstrate Mid to replace strings.

```
s = "123456789"
Mid(s, 3, 5, "")      'Replace five characters with nothing
Print s               '1289

s = "123456789"
Mid(s, 3, 5, "XX")    'Replace five characters with two
Print s               '12XX89

s = "123456789"
Mid(s, 3, 5, "ABCDEFG") 'Cannot add more than you replace from the middle
Print s               '12ABCDE89

s = "123456789"
Mid(s, 7, 12, "ABCDEFG") 'You can add more than you remove from the end
Print s               '123456ABCDEFG
```

7.5. Replace

The ReplaceInString function (see Listing 138) emulates the Mid function, with two exceptions: It puts the entire new string in place, even if it's longer than the substring that it replaces, and it doesn't modify the original string.

Listing 138. A general replace string method.

```
REM This function is similar to Mid with four arguments.
REM This function does not modify the original string.
REM This function handles replacement text larger than n.
Function ReplaceInString(s$, i&, n&, sNew$) As String
    If i <= 1 Then
        'Place the string in front.
        'The only question is how much string must be removed.

        If n < 1 Then                                'Remove nothing
            ReplaceInString = sNew & s
        ElseIf n >= Len(s) Then                      'Remove everything
            ReplaceInString = sNew
        Else                                          'Remove a portion from the left
            ReplaceInString = sNew & Right(s, Len(s) - n)
        End If

    ElseIf i + n > Len(s) Then
        'Replacing past the end, then extract the leftmost parts
        'Mid works to find if the length argument is larger than the
        'string, so this is fine! Append the new text to the end.

        ReplaceInString = Mid(s, 1, i - 1) & sNew

    Else
        'Replace somewhere in the middle of the string.
        'First, obtain the leftmost text.
        'Second, insert the new text, if any.
        'Finally, obtain the rightmost text.

        ReplaceInString = Mid(s, 1, i - 1) & sNew & Right(s, Len(s) - i - n + 1)
    End If
End Function
```

But wait, there is an undocumented statement Replace(strng, find, rpl, start, Count, mode), which finds all occurrences of find and replaces them with rpl. The last three arguments are optional.

The start value indicates what part of the string to return, not where to start replace. A value of one for the start value returns all of the characters. A start value of three ignores the first two characters in the return string.

The count indicates the maximum number of times to replace the found text. A value of -1 replaces all values.

A mode value of 1, the default, indicates a case-insensitive compare is used to find matching text. A mode value of 0 causes a case-sensitive compare.

7.6. Aligning strings with LSet and RSet

Use the LSet and RSet statements to left-justify and right-justify strings into the space taken by another string. This is useful, for example, to create column headings that are left or right justified with leading or trailing spaces. The RSet and LSet functions use the same syntax.

```
LSet string_1 = expression
RSet string_1 = expression
```

The string on the left-hand side can contain any data as long as it is the length that you want. The right-hand side must evaluate to a string; that string will be displayed in the space defined by the length of the left-hand string. Unlike the behavior for many functions in Basic, the expression is not automatically converted to a string.

Listing 139. RSet.

```
Dim s As String      'String variable to contain the result
s = String(10, "*")  'The result is 10 characters wide
RSet s = CStr(1.23)  'The number is not automatically converted to a string
Print "$" & s        '$      1.23
```

The important thing about the string on the left is its length — the width of the field in which the value of interest will be displayed. The easiest way to get a string of a specified length is to use the String function. The character that you use is not important because all of the extra characters in the final result are replaced with spaces.

Listing 140. LSet.

```
Dim s As String      'String variable to contain the result
s = String(10, "X")  'The result is 10 characters wide
LSet s = CStr(1.23)  'The number is not automatically converted to a string
Print s & "%"        '1.23   %
```

If the string on the left is shorter than the string expression on the right, the expression is truncated to fit. Both LSet and RSet chop characters from the end of the expression to make it fit within the defined string length.

Listing 141. LSet and RSet truncate.

```
Dim s As String      'String variable to contain the result
s = String(4, "X")   'The result will be four characters wide
LSet s = CStr(21.23) 'Truncated on the right
Print "$" & s & "%"   '$21.2%
RSet s = CStr(21.23) 'Truncated on the right
Print "$" & s & "%"   '$21.2%
```

The code in Listing 142 demonstrates the behavior of the LSet and RSet commands. The results are displayed in Figure 55.

Listing 142. Complete LSet and RSet example.

```
Sub ExampleLSetAndRSet
```

```

Dim s As String
Dim sVar As String
Dim sTmp As String

sTmp = "12345"
sVar = String(10, "*")
LSet sVar = sTmp
s = "LSet " & String(10, "*") & " = " & sTmp & _
    " == >" & sVar & "<" & CHR$(10)

sVar = String(10, "*")
RSet sVar = sTmp
s = s & "RSet " & String(10, "*") & " = " & sTmp & _
    " == >" & sVar & "<" & CHR$(10) & CHR$(10)

sVar = String(2, "*")
LSet sVar = sTmp
s = s & "LSet " & String(2, "*") & " = " & sTmp & _
    " == >" & sVar & "<" & CHR$(10)

sVar = String(2, "*")
RSet sVar = sTmp
s = s & "RSet " & String(2, "*") & " = " & sTmp & _
    " == >" & sVar & "<" & CHR$(10)

MsgBox s, 0, "RSet and LSet"
End Sub

```

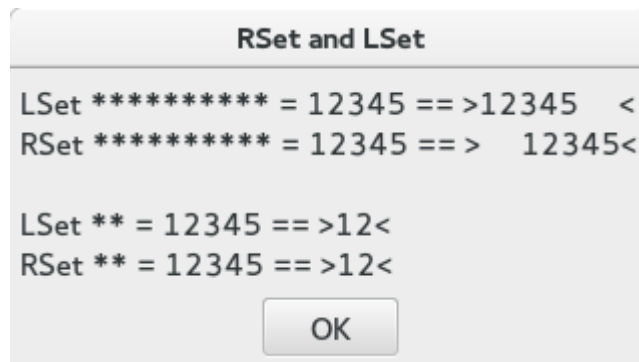


Figure 55. RSet and LSet justify strings.

TIP

In Visual Basic, LSet allows you to overlay data from one user-defined type with data from another, overlaying all the bytes from one data structure on top of the other; ignoring the underlying structure. In LO Basic, LSet only manipulates strings.

7.7. Fancy formatting with Format

You can convert a number to a string formatted according to the optional format string. And you can include multiple formats in a single format string. See Table 53. The current locale influences the returned formatted string. Set the locale using **Tools | Options | Language Settings | Languages**. If the format string is omitted, Format produces output similar to the CStr function.

Listing 143. Simple Format statement.

```
Print Format(1223, "00.00")           '1223.00
Print Format(1234.56789, "###00.00")  '1234.57
```

Each individual format is separated by a semicolon (;). The first format is used for positive numbers, the second for negative numbers, and the third for zero. If only one format code is present, it applies to all numbers.

Listing 144. Format string may specify formatting for positive , negative, and zero numbers.

```
Dim s As String
s = "P 00000.000;N ####.00;Z 0.0"
Print Format(-12.3, s)   'N 12.30
Print Format(0, s)       'Z 0.0
Print Format(12.3, s)    'P 000012.300
```

Table 53. Numeric Format specifiers.

Code	Description
0	If the number has a digit at the position of the 0 in the format code, the digit is displayed; otherwise a zero appears. This means that leading and trailing zeros are displayed, leading digits are not truncated, and trailing decimals are rounded.
#	This works like the 0, but leading and trailing zeros are not displayed.
.	The decimal placeholder determines the number of decimal places to the left and right of the decimal separator. Although the period is used in the format string regardless of locale, the output correctly uses the locale-specific decimal separator.
%	Multiply the number by 100 and insert the percent sign (%) where it appears in the format code.
E- E+ e- e+	If the format code contains at least one numerical digit placeholder (0 or #) to the right of the symbol, the number is formatted in the scientific notation. The letter E or e is inserted between the number and the exponent. The number of placeholders for digits to the right of the symbol determines the number of digits in the exponent. If the exponent is negative, a minus sign (-) is displayed directly before an exponent. If the exponent is positive, a plus sign (+) is only displayed before exponents with E+ or e+.
,	The comma is a placeholder for the thousands separator. It separates thousands from hundreds in a number with at least four digits. The thousands delimiter is displayed if the format code contains the placeholder surrounded by digit placeholders (0 or #).
- + \$ () space	Plus signs (+), minus signs (-), dollar signs (\$), spaces, or brackets entered directly in the format code are displayed as the literal character.
\	The backslash displays the next character in the format code. In other words, it prevents the next character from being seen as a special character. The backslash is not displayed unless you enter a double backslash (\\) in the format code. Characters that must be preceded by a backslash in the format code in order to be displayed as literal characters are the date- and time-formatting characters (a, c, d, h, m, n, p, q, s, t, w, y, /, :), numeric-formatting characters (#, 0, %, E, e, comma, period), and string-formatting characters (<, >). You may also enclose characters in double quotation marks.
General Number	Numbers are displayed as entered.
Currency	A currency symbol is placed in front of the number, and negative numbers are in brackets.
Fixed	At least one digit is displayed in front of the decimal separator. Two decimals are displayed.
Percent	Multiply the number by 100 and append a percent sign (%).
Standard	Displays numbers with a locale-specific thousands separator. Two decimals are displayed.
Scientific	Displays numbers in scientific notation. Two decimals are displayed.

The format function has been dramatically improved over the years and most of the bugs have been fixed. Format specifiers related to numbers are shown in Table 53.

Listing 145. Demonstrate numeric format specifiers.

```
Sub ExampleFormat
    MsgBox Format(6328.2, "##,##0.00")           REM 6,328.20
    MsgBox Format(123456789.5555, "##,##0.00")    REM 123,456,789.56
    MsgBox Format(0.555, ".##")                  REM .56
    MsgBox Format(123.555, "#.##")                REM 123.56
    MsgBox Format(123.555, ".##")                REM 123.56
    MsgBox Format(0.555, "0.##")                 REM 0.56
    MsgBox Format(0.1255555, "%#.##")            REM %12.56
    MsgBox Format(123.45678, "##E-####")         REM 1E2
    MsgBox Format(.0012345678, "0.0E-####")      REM 1.2E-003
    MsgBox Format(123.45678, "#.e-###")          REM 1e002
    MsgBox Format(.0012345678, "#.e-###")        REM 1e-003
    MsgBox Format(123.456789, "#.## is ###")      REM 123.46
    MsgBox Format(8123.456789, "General Number") REM 8123.456789
    MsgBox Format(8123.456789, "Fixed")           REM 8123.46
    MsgBox Format(8123.456789, "Currency")       REM 8,123.46 $ (broken)
    MsgBox Format(8123.456789, "Standard")       REM 8,123.46
    MsgBox Format(8123.456789, "Scientific")      REM 8.12E+03
    MsgBox Format(0.00123456789, "Scientific")   REM 1.23E-03
    MsgBox Format(0.00123456789, "Percent")      REM 0.12%
End Sub
```

Format specifiers related to date and time formatting are shown in Table 54. For reasons that I do not understand, they are not included with the standard documentation.

Table 54. Date and time format specifiers.

Code	Description
q	The quarter of the year (1 through 4).
qq	The quarter of the year as 1st quarter through 4th quarter
y	The day in the year (1 through 365).
yy	Two-digit year.
yyyy	Complete four-digit year.
m	Month number with no leading zero.
mm	Two-digit month number; leading zeros are added as required.
mmm	Month name abbreviated to three letters.
mmmm	Full month name as text.
mmmmm	First letter of month name.
d	Day of the month with no leading zero.
dd	Day of the month; leading zeros are added as required.
ddd	Day as text abbreviated to three letters (Sun, Mon, Tue, Wed, Thu, Fri, Sat).
dddd	Day as text (Sunday through Saturday).

Code	Description
dddd	Full date in a short date format.
dddddd	Full date in a long format.
w	Day of the week as returned by WeekDay (1 through 7).
ww	Week in the year (1 though 52).
h	Hour with no leading zero.
hh	Two-digit hour; leading zeros are added as required.
n	Minute with no leading zero.
nn	Two-digit minute; leading zeros are added as required.
s	Second with no leading zero.
ss	Two-digit second; leading zeros are added as required.
tttt	Display complete time in a long time format.
c	Display a complete date and time.
/	Date separator. A locale-specific value is used.
:	Time separator. A locale-specific value is used.

The date and time format specifiers are now implemented. I am aware of at least one bug that is demonstrated in Listing 146 with the format string “d/mmmm/yyyy h:nn:ss”; Figure 56 shows that “nn” does not expand properly on the last line.

Listing 146. *Demonstrate date and time format specifiers.*

```

Sub FormatDateTimeStrings
    Dim i%
    Dim d As Date
    d = now()
    Dim s$
    Dim formats
    formats = Array("q", "qq", "y", "yy", "yyyy", _
                    "m", "mm", "mmm", "mmm", "mmmm", _
                    "d", "dd", "ddd", "ddd", "dddd", "dddd", _
                    "w", "ww", "h", "hh", "n", "nn", "nnn", "s", "ss", _
                    "tttt", "c", "d/mmmm/yyyy h:nn:ss")
    For i = LBound(formats) To UBound(formats)
        s = s & formats(i) & " => " & Format(d, formats(i)) & CHR$(10)
    Next
    MsgBox s
End Sub

```

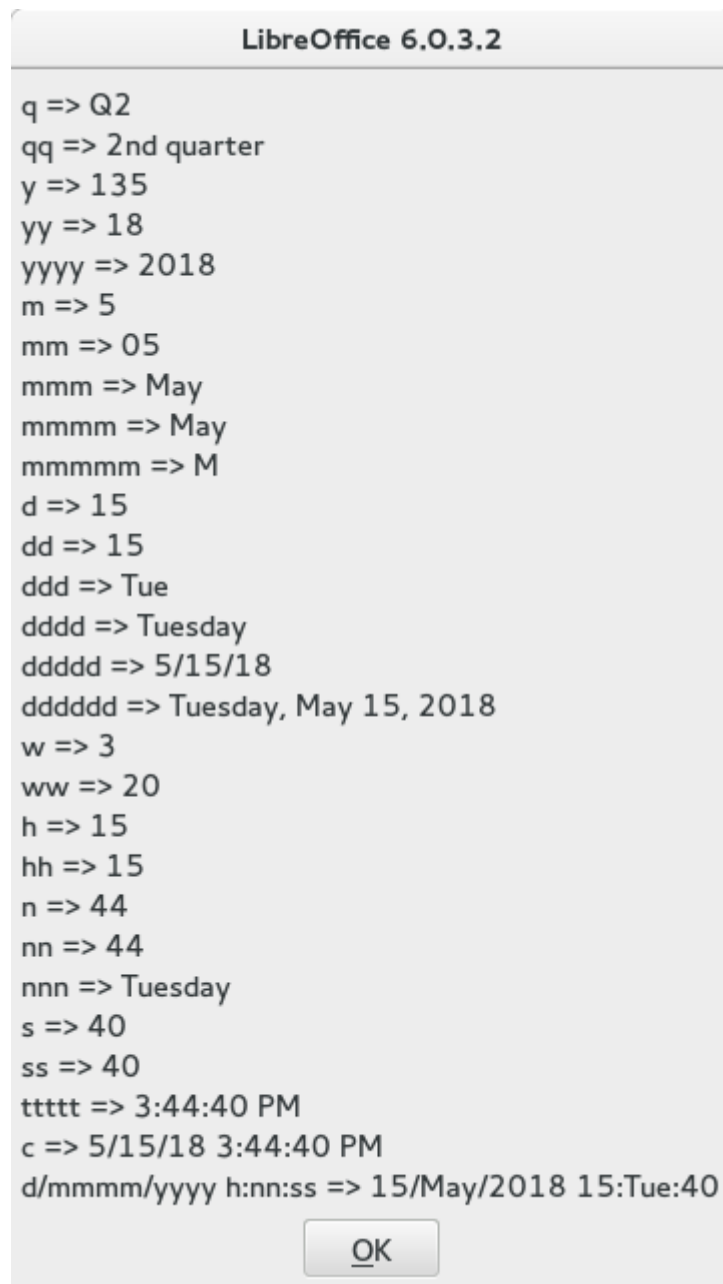


Figure 56. Date and time format specifiers.

Format specifiers related to strings are shown in Table 55.

Table 55. String format specifiers.

Code	Description
<	String in lowercase.
>	String in uppercase.

Other string format specifiers used to be documented, but have never been implemented (see Table 56). I include them because the specifiers in Table 54 and Table 55 used to be documented but not implemented; now they are implemented but not documented.

Table 56. String format specifiers.

Code	Description
@	Character placeholder. If the input character is empty, place a space in the output string. For example, “(@@@)” formats to “()” with an empty string.
&	Character placeholder. If the input character is empty, place nothing in the output string. For example, “(&&&)” formats to “()” with an empty string.
!	Normally, character placeholders are filled right to left; the ! forces the placeholders to be filled left to right.

As of this writing with OOo version 3.2.1, only the upper / lower case string format specifiers are implemented.

Listing 147. String format specifiers.

```
Sub FormatStrings
    Dim i%
    Dim s$
    Dim formats
    formats = Array("<", ">", _
        "@@", " (@@@)", "[ @@@@]", _
        "&&", " (&&&)", "[ &&&&]", _
    )
    For i = LBound(formats) To UBound(formats)
        s = s & formats(i) & " => (" & Format("On", formats(i)) & ") " & CHR$(10)
    Next
    MsgBox s
End Sub
```

7.8. Converting data to a string

Basic contains functions that convert other data types to a string. Although the Format function is the most versatile method of converting a number to a string, that level of control is typically not required. The Str function converts a number to a string with no localization, and the Val function converts it back to a number.

The Hex and Oct functions convert a Long to its corresponding hexadecimal or octal notation. The leading “&H” and “&O” are not included. To convert back to a number, these strings must be manually prepended to the string.

The CStr function is able to intelligently convert almost any data type to a string in a locale-specific way (see Table 57). The Str function is limited to numbers and does not perform a locale-specific conversion.

Table 57. Converting data types with CStr.

Type	Converted to String
Boolean	True or False
Date	Formatted date string such as 06/08/2010
Null, uninitialized object	Run-time error
Empty, uninitialized variant	Zero-length string
any numeric value	Number as a string

Listing 148. CStr with a few data types.

```
Sub ExampleCStr
    On Error Goto Handler
    Dim b As Boolean
    Dim o As Object
    Dim v As Variant ' This is empty
    Dim d As Double  : d = PI()

    Print "Boolean (" & CStr(b) & ")"
    Print "Date (" & CStr(Now) & ")"
    Print "Empty Variant (" & CStr(v) & ")"
    Print "Double (" & CStr(d) & ")"
    Print "Null object (" & CStr(o) & ")"
    Exit Sub
Handler:
    Print "Encountered error: " & Error
    Resume Next
End Sub
```

The CStr function is useful when you need to explicitly convert a value to a string to avoid incorrect default conversions to other types. For example, the first operand to the addition operator determines if the result is a string or a number. This is also an argument against the use of the addition operator (+) for string concatenation rather than the operator specifically designed for string concatenation (&).

```
Print 3 + "4"      '7
Print CStr(3) + "4" '34
```

The Join function concatenates all of the elements in a one-dimensional array into a single string. If no delimiter is specified, a space separates each element.

```
Print Join(Array(3, 4, 5))      '3 4 5
Print Join(Array(3, 4, 5), "X") '3X4X5
```

The Split function is used to split a string into pieces based on an optional delimiter. This is essentially the opposite of the Join function and is the fastest method to parse a string into a series of substrings based on a delimiter.

```
Split("3 4 5")      'returns the array (3, 4, 5)
Split("3X4X5", "X") 'returns the array (3, 4, 5)
```

7.9. Advanced searching (regular expressions)

The usual search methods are StrComp (Listing 126), InStr (Listing 131), and InStrRev (Listing 132). Advanced searching can be done using the TextSearch service. Searching can be done using ABSOLUTE, REGEXP, or APPROXIMATE mode. I will not take the time to pursue the TextSearch service in depth; for example, to investigate replacing text.

Listing 149. using the TextSearch service.

```
Sub StringTextSearch
    Dim oTextSearch           ' TextSearch service.
    Dim sStrToSearch As String ' String to search.
    Dim sMatchString As String ' String that was found.
    Dim aSearchResult         ' com.sun.star.util.SearchResult
    Dim rank As Long
    Dim iMatchStartPos As Long
    Dim iMatchLen As Long
    Dim aSrcOpt As New com.sun.star.util.SearchOptions
    Dim s$
    Dim enLocale As New com.sun.star.lang.Locale

    enLocale.Language = "en"
    enLocale.Country = "US"

    oTextSearch = CreateUnoService("com.sun.star.util.TextSearch")
    s = ""

    With aSrcOpt
        'https://api.libreoffice.org/docs/idl/ref/SearchFlags_8idl.html
        .searchFlag = com.sun.star.util.SearchFlags.REG_EXTENDED
        .Locale = enLocale
        'Supports ABSOLUTE, REGEXP, and APPROXIMATE
        .algorithmType = com.sun.star.util.SearchAlgorithms.REGEXP
        .searchString = "a+"

        'This does not work.
        '.transliterateFlags = com.sun.star.il8n.TransliterationModulesNew.IGNORE_CASE

        'This works
        .transliterateFlags = com.sun.star.il8n.TransliterationModulesNew.UPPERCASE_LOWERCASE
    End With

    oTextSearch.setOptions(aSrcOpt)

    sStrToSearch = "aaa hello AAA"
    aSearchResult = oTextSearch.searchForward(sStrToSearch, 0, Len(sStrToSearch)-1 )
    ' Print aSearchResult.subRegExpressions

    REM subRegExpressions has value zero if no match...
    Do While aSearchResult.subRegExpressions > 0
        'Print "" + LBound(aSearchResult.startOffset) + ":" + UBound(aSearchResult.startOffset)
        rank = aSearchResult.subRegExpressions - 1
        iMatchStartPos = aSearchResult.startOffset(rank) + 1
```

```

iMatchLen = aSearchResult.endOffset(rank) - aSearchResult.startOffset(rank)
sMatchString = Mid(sStrToSearch, iMatchStartPos, iMatchLen)
s = s & "(" + LBound(aSearchResult.startOffset) & ":" & _
        UBound(aSearchResult.startOffset) & ")" => " & sMatchString & CHR$(10)
If Len(sStrToSearch) <= aSearchResult.endOffset(rank)+1 Then
    Exit Do
End If

aSearchResult = oTextSearch.searchForward(sStrToSearch, _
        aSearchResult.endOffset(rank)+1, Len(sStrToSearch)-1 )

Loop
MsgBox s
End Sub

```

7.10. Conclusion

It pays to know the different functions supported by Basic. Before I was aware of the Split function, I spent a lot of time writing a macro that parsed a string into pieces. I rewrote my code using the Split function and the macro was significantly faster.

There are a lot of very powerful capabilities for formatting text in Basic. Among other things, the use of the Unicode character set allows processing of nearly any language in the world. There are also a number of good functions for joining, splitting, and formatting text strings.

8. File Routines

This chapter introduces the subroutines and functions supported by Basic that are related to files and directories. After reading this chapter you'll be able to create, delete, rename, and move files and directories. You'll learn methods that inspect files, both open and closed, and directories. This chapter also explains the idiosyncrasies and bugs related to reading and writing files, along with differences between operating systems.

Basic includes functions that allow you to interact with the file system (see Table 58). You can perform simple and complex tasks such as creating and deleting directories, or even opening and parsing files. In this chapter I'll spend a fair amount of time on directories, file attributes, and the different file types. I will examine how files are organized and manipulated, how the different file types are structured, and which functions read and write data for those different file types. I was happy with how easily I was able to write macros to move and rename files. On the other hand, the functions to manipulate binary and random files feel rough around the edges.

Table 58. *File functions in Basic.*

Function	Description
ChDir(path)	Change the currently logged directory or drive. Deprecated; do not use.
ChDrive(path)	Change the currently logged drive. Deprecated; do not use.
Close #n	Close a previously opened file or files. Separate file numbers with a comma.
ConvertFromURL(str)	Convert a path expressed as a URL to a system-specific path.
ConvertToURL(str)	Convert a system-specific path to a URL.
CurDir CurDir(drive)	Return the current directory as a system-specific path. If the optional drive is specified, the current directory for the specified drive is returned.
Dir(path) Dir(path, attr)	Return a listing of files based on an included path. The path may contain a file specification — for example, “/home/andy/*.txt”. Optional attributes determine if a listing of files or directories is returned.
EOF(number)	Return True if the file denoted by “number” is at the end of the file.
FileAttr(number, 1)	Return the mode used to open the file given by “number”. The second argument specifies if the file-access or the operating-system mode is desired, but only the file mode is currently supported.
FileCopy(src, dest)	Copy a file from source to destination.
FileDateTime(path)	Return the date and time of the specified file as a string.
FileExists(path)	Return True if the specified file or directory exists.
FileLen(path)	Return the length of the specified file as a long.
FreeFile()	Return the next available file number for use.
Get #number, variable Get #number, pos, variable	Read a record from a relative file, or a sequence of bytes from a binary file, into a variable. If the position argument is omitted, data is read from the current position in the file. For files opened in binary mode, the position is the byte position in the file.
GetAttr(path)	Return a bit pattern identifying the file type.
GetPathSeparator()	Return the system-specific path separator.
Input #number, var	Sequentially read numeric or string records from an open file and assign the data to one or more variables. The carriage return (ASC=13), line feed (ASC=10), and comma act as delimiters. Input cannot read commas or quotation marks (") because they delimit the text. Use the Line Input statement if you must do this.

Function	Description
Kill(path)	Delete a file from disk.
Line Input #number, var	Sequentially read strings to a variable line-by-line up to the first carriage return (ASC=13) or line feed (ASC=10). Line end marks are not returned.
Loc(number)	Return the current position in an open file.
LOF(number)	Return the size of an open file, in bytes.
MkDir(path)	Create the directory.
Name src As dest	Rename a file or directory.
Open path For Mode As #n	Open a data channel (file) for Mode (Input = read, Output = write)
Put #n, var Put #n, pos, var	Write a record to a relative file or a sequence of bytes to a binary file.
Reset	Close all open files and flush all files to disk.
RmDir(path)	Remove a directory.
Seek #n, pos	Set the position for the next writing or reading in a file.
SetAttr(path, attr)	Set file attributes.
Write #n, string	Write data to a file.

8.1. Using URL notation to specify a file

Many of the functions in Table 58 specify a file or path. These functions accept both system-specific names and Uniform Resource Locator (URL) notation. This is the same notation used by your Web browser. Table 59 shows examples.

Table 59. URL examples.

System	System Path	URL Path
Windows	c:\Temp\help.txt	file:///c:/Temp/help.txt
Windows	c:\My Documents	file:///c:/My%20Documents
Unix	/home/andy/Temp/help.txt	file:///home/andy/Temp/help.txt
Unix	/home/andy/My Documents	file:///home/andy/My%20Documents

TIP

The statement “Shell(“C:\Prog Files\calc.exe”,2)” failed because there is a space in the path. The Shell statement passes the string to the command shell, which interprets the portion of the path before the space as the program to run. URL notation avoids this problem.

One advantage of URL notation is that special characters are encoded. Arguments that are passed to a shell, for example, frequently have problems with paths that contain a space. In URL notation, spaces are encoded as “%20” (see Table 59). Use the functions ConvertToURL to convert a system-specific path to URL notation and ConvertFromURL to convert to a system-specific path.

Listing 150. Converting to and from a URL.

```
Sub ToFromURL
    Print ConvertToURL("/home/andy/logo.miff")
    Print ConvertFromURL("file:///home/andy/logo.miff") 'This requires UNIX
    Print ConvertToURL("c:\My Documents") 'This requires Windows
```



```
Print ConvertFromURL("file:///c:/My%20Documents") 'This requires windows
End Sub
```

Special characters, such as the space, are encoded with a percent sign (%) followed by the ASCII value of the character encoded as a two-digit hexadecimal number. The space character has an ASCII value of 32, which is 20 in hexadecimal format. This is why a space is encoded as %20.

Listing 151. Special URL characters.

```
Sub URLSpecialEncoding
Print ConvertFromURL("file:///c:/ABC/abc") ' /ABC/abc (UNIX)
Print ConvertFromURL("file:///c:/ABC/abc") ' /ABC/abc (Windows)
End Sub
```

TIP On my Linux computer, ConvertFromURL("file:///c:/ABC/abc") does not properly decode.

URL notation is system independent, so URL paths work as well on an Apple computer as they do on a Windows computer. To create a system-specific path, use the function GetPathSeparator to obtain the system-specific path separator. Listing 152 demonstrates how to use GetPathSeparator to build a complete path. Windows-based computers use “\” as the path separator, and Unix-based computers use “/” as the path separator. URL notation uses “/” as the separator regardless of the operating system.

Listing 152. Use GetPathSeparator() rather than “\” or “/”.

```
sPathToFile = "C:\temp"
sBookName = "OOME.odt"
sPathToBook = sPathToFile & GetPathSeparator() & sBookName
```

TIP Visual Basic for Applications (VBA) does not support the function GetPathSeparator, but it does have the property Application.PathSeparator, which always returns a backslash, even on a Macintosh computer. VBA also does not support ConvertToURL or ConvertFromURL.

8.2. Directory manipulation functions

Some functions apply equally well to directories as well as files. This section is concerned with those that apply only to directories.

The function CurDir, with a drive specifier as the argument, returns the current directory for the specified drive. See Listing 153 and Figure 57. If the argument is omitted, the current directory for the current drive is returned. The drive specifier is ignored on Unix systems. The initial value of the current directory is system dependent and may change depending upon how OOo is started. If you start OOo from a command-line prompt, you’ll likely have a different current directory than if you start OOo from a menu or another application. For some operating systems, using File | Open to open an existing document sets the current directory to the directory containing the opened document (I have seen this in Windows). In some operating systems, such as Linux, this does not affect the current directory. Do not rely on this behavior!

Listing 153. Print the current directory.

```
Sub ExampleCurDir
```

```

MsgBox "Current directory on this computer is " & _
    CurDir, 0, "Current Directory Example"
End Sub

```

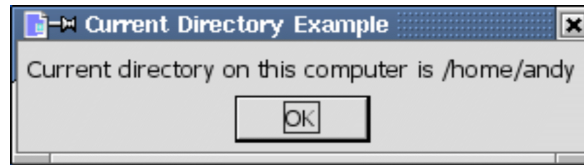


Figure 57. *CurDir returns the current directory.*

The functions ChDir and ChDrive, although present in Basic, do nothing and will likely be removed from the language. Their original purpose was to change the current drive and directory, but this was a system wide change, which is dangerous in multitasking environments like we use today. The initial current directory is dependent upon the operating system and how LO was opened. The initial values, therefore, cannot be assumed.

Use the Mkdir function to create a directory, and Rmdir to remove a directory. In Listing 154, a directory path is created from the argument to Mkdir. If an absolute path is not provided, the created directory is relative to the current directory as obtained with the function CurDir. The function Rmdir removes the directory, all directories below it, and all files contained in the directories. This macro calls OOMEWorkDir in Listing 165.

Listing 154. *Create and then remove directories in the OOME Work Directory.*

```

Sub ExampleCreateRmDirs
    If NOT CreateOOMEWorkDir() Then
        Exit Sub
    End If
    Dim sWorkDir$
    Dim sPath$
    sWorkDir = OOMEWorkDir()
    sPath = sWorkDir & "a" & GetPathSeparator() & "b"
    Mkdir sPath
    Print "Created " & sPath
    RmOOMEWorkDir()
    Print "Removed " & sWorkDir
End Sub

```

The code in Listing 154 uses absolute paths. It's possible to use relative paths, but I strongly discourage it. The behavior of the current directory is operating-system dependent.

File-related functions that also work with directories include Dir, FileDateTime, FileExists, FileLen, GetAttr, and Name. These are discussed later.

8.3. File manipulation functions

This section explores functions that deal with inspecting and manipulating entire files, rather than the contents of those files. Some of these functions have a dual purpose, acting on both files and directories. In each case, the function accepts at least one argument that identifies a file or directory. The following things are true about arguments that identify files or directories:

If the path is not present, the current directory — as returned by CurDir — is used.

The system representation and the URL notation are both allowed. For example, “C:\tmp\foo.txt” and “file:///c:/tmp/foo.txt” refer to the same file.

Unless it is explicitly stated, a single file or directory must be uniquely identified. The only function that accepts a file specification is Dir, which returns a listing of files matching the file specification.

Each file and directory has attributes (see Table 60). Each attribute represents a single bit in a number, allowing each item in a path to have multiple attributes set at the same time. Some attributes have been deprecated to be more system dependent. Not all systems support hidden or system files, for example. Use GetAttr to return the attributes.

Table 60. File and directory attributes.

Deprecated	Attribute	Description
No	0	Normal; no bits set
No	1	Read-Only
Yes	2	Hidden
Yes	4	System
No	8	Volume
No	16	Directory
No	32	Archive bit (file changed since last backed up)

The function in Listing 155 accepts an attribute from the GetAttr function and returns an easy-to-understand string. If no bits are set, the attributes indicate a normal file.

Listing 155. Print attributes as a string.

```
REM uses bitwise comparison to read the attributes
Function FileAttributeString(x As Integer) As String
    Dim s As String
    If (x = 0) Then
        s = "Normal"
    Else
        If (x AND 16) <> 0 Then s = "Directory"           'Directory bit 00010000 set
        If (x AND 1)  <> 0 Then s = s & " Read-Only"      'read-only bit 00000001 set
        If (x AND 2)  <> 0 Then s = s & " Hidden"         'Deprecated
        If (x AND 4)  <> 0 Then s = s & " System"         'Deprecated
        If (x AND 8)  <> 0 Then s = s & " Volume"         'Volume bit 00001000 set
        If (x AND 32) <> 0 Then s = s & " Archive"        'Archive bit 00100000 set
    End If
    FileAttributeString = s
End Function
```

TIP Listing 155 performs bit operations (explained later) to determine which attributes are set.

Use the GetAttr function to get the attributes of a file, and use SetAttr to set the attributes. The first argument to the function SetAttr is the name of the file — relative or absolute — and the second argument is a number representing the attributes to set or clear. In other words, after

calling `SetAttr(name, n)`, the function `GetAttr(name)` should return the integer `n`. For example, calling `SetAttr` with the attribute set to 32 sets the archive bit and clears all the others so `GetAttr` returns 32. To set more than one bit at the same time, use the OR operator to combine attributes. Use `SetAttr(fileName, 1 OR 32)` to set both the archive bit and the read-only bit. `SetAttr` works on directories as well as files.

TIP Attributes favor the Windows environment. On Unix-based operating systems such as Linux and Sun, setting attributes affects the user, group, and world settings. Setting the attribute to 0 (not read-only) corresponds to “`rw-rw-rw-`”. Setting the attribute to 1 (read-only) corresponds to “`r--r--r--`”.

Use the `FileLen` function to determine the length of a file. The return value is a long. The function in Listing 156 obtains the file length and then creates a pretty string to display the length. The file length is returned in bytes — K, MB, G, or T — depending on the length. This produces a more easily understood result than a simple number.

Listing 156. *Display a number in a nice readable form such as 2K rather than 2048.*

```
Function PrettyFileLen(path$) As String
    PrettyFileLen = nPrettyFileLen(FileLen(path))
End Function

Function nPrettyFileLen(ByVal n As Double) As String
    Dim i As Integer      'Count number of iterations
    Dim v() As Variant    'Holds abbreviations for Kilobytes, Megabytes, ...
    v() = Array("bytes", "K", "MB", "G", "T") 'Abbreviations

    REM Every time that the number is reduced by 1 kilobyte,
    REM the counter is increased by 1.
    REM Do not decrease the size to less than 1 kilobyte.
    REM Do not increase the counter by more than the size of the array.
    Do While n > 1024 AND i+1 < UBound(v())
        n = Fix(n / 1024) 'Truncate after the division
        i = i + 1         'Started at i=0 (bytes) increment to next abbreviation
    Loop
    nPrettyFileLen = CStr(n) & v(i)
End Function
```

Use the `FileExists` function to determine if a file or directory exists. Use `FileDateTime` to return a string with the date and time that a file was created or last modified. The returned string is in a system-dependent format. On my computer, the format is “MM/DD/YYYY HH:MM:SS”. The returned string can be passed directly to the function `CDate`. The `GetFileInfo` macro in Listing 157 uses all of the file and directory inspection functions to return information in an easy-to-read format. Also see Figure 58.

Listing 157. *Get information about a file.*

```
Function GetFileInfo(path) As String
    Dim s As String
    Dim iAttr As Integer
    s = "The path " & path & " is "
    If Not FileExists(path) Then
```

```

    GetFileInfo = s & " does not exist"
    Exit Function
End If
s = s & " exists" & CHR$(10)
s = s & "Date and Time = " & FileDateTime(path) & CHR$(10)
iAttr = GetAttr(path)
REM The length of a directory is always zero
If (iAttr AND 16) = 0 Then
    s = s & "File length = " & PrettyFileLen(path) & CHR$(10)
End If
s = s & "Attributes = " & FileAttributeString(iAttr) & CHR$(10)
GetFileInfo = s
End Function

```

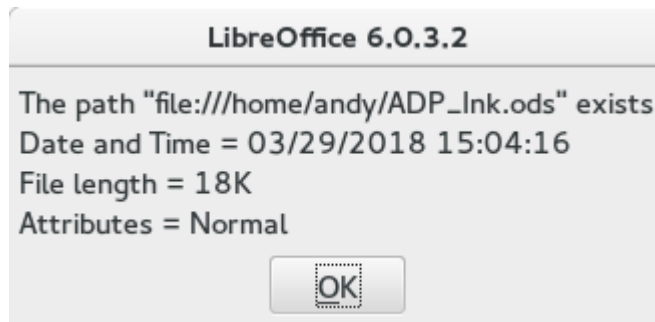


Figure 58. You can learn a lot about a file by using the file-inspection functions.

Use the Kill statement to delete a file from the disk. A run-time error occurs if the file does not exist.

```
Kill("C:\temp\BadFile.txt")
```

Use the FileCopy function to copy files. The first argument is the file to copy and the second argument is the destination file. See Table 61. The FileCopy function is able to recursively copy entire directories, but it can't handle file specifications. Surprisingly, if the first argument is a file, the second argument must also be a file — I expected that I could copy a file to a directory with FileCopy("C:\auto.bat", "C:\bak\").

Table 61. Arguments to FileCopy.

Valid	Source	Destination	Comment
Yes	File	File	Copy the file. The names do not have to be the same.
Yes	Directory	Directory	Recursively copy all files and directories contained in one directory to another directory.
No	file spec		File specifications (wildcards, for example, *.*) are not allowed.
No	File	Directory	If the source is a file, the destination must also be a file.

```

FileCopy("C:\auto.bat", "C:\auto.bak")      'Copy file
FileCopy("C:\auto.bat", "C:\tmp\auto.bat")  'Copy file
FileCopy("C:\logs", "C:\bak")               'Copy directory

```

TIP

Do not recursively copy a directory into itself — this creates an infinite loop. For example, `FileCopy("C:\logs", "C:\logs\bak")` will never finish because the “bak” subdirectory immediately becomes part of the contents of “logs”, which then has to be copied as well. Bad idea.

Use the `Name` statement to rename a file or directory. This statement has an unusual syntax: It places the keyword `As` between the source and destination names.

```
Name "C:\Joe.txt" As "C:\bill.txt"      'Rename a file
Name "C:\logs" As "C:\oldlogs"         'Rename a directory
Name "C:\Joe.txt" As "C:\tmp\joe.txt"  'Move the file to the tmp directory
Name "C:\logs" As "C:\bak\logs"        'Move the logs directory
```

TIP

A common power-user trick is to use the `Name` command to move a file or directory from one location to another.

8.4. File attributes, bitmasks, and binary numbers

It isn’t necessary to understand binary numbers and bitmasks to use either file attributes or bitmasks in Basic, so don’t panic; simply skip the parts that make your head spin. Understanding this material, however, makes it simpler to understand what is happening with file attributes and how to use them.

The file and directory attributes in Table 60 were strategically chosen to have a nice property when written in base 2 (see Table 62) — each attribute has only one bit set. Zero is a special case — it has no bits set.

Table 62. File and directory attributes.

Decimal Attribute	Binary Attribute	Description	Comment
00	0000 0000	Normal	No bits set
01	0000 0001	Read-Only	Bit 1 set
02	0000 0010	Hidden	Bit 2 set
04	0000 0100	System	Bit 3 set
08	0000 1000	Volume	Bit 4 set
16	0001 0000	Directory	Bit 5 set
32	0010 0000	Archive	Bit 6 set

Use `GetAttr` to obtain the attributes of a file or path. If the file or path is a directory, then bit 5 is set. If the file or path is read-only, then bit 1 is set. A returned attribute of 0 means that no bits are set and that this is a normal file. Consider an attribute value of 33, which in binary is 0010 0001. Bit 1 is set, so this is read-only. Bit 6 is set, so this file has changed since it was last archived. You can see that you don’t need to know how to convert a decimal number into a binary number. However, you do need to know how to write a macro to determine which bits are set and which bits are not set. Use the `AND` operator to determine which bits are set. With `AND`, two things must

both be true for the answer to be true. For example, my flashlight works if it has a light bulb AND it has batteries.

The AND operator works with numbers by performing this logical operation on each of the bits. For example, “3 AND 5” represented as base 2 is “0011 AND 0101 = 0001”. Bit 1 — the bit in the rightmost position — in each number is equal to 1, so bit 1 in the result is also 1. All of the other bits do not have corresponding 1s in the same position, so all of the other bits in the result equal zero.

Now I’ll apply this idea to the problem at hand. If the numeric value of an attribute is not zero, then at least one property is set. Given this, you can then check each attribute as illustrated by the example in Table 63.

Table 63. Check attribute value 33 (100001) for each file property.

Read-Only	Hidden	System	Volume	Directory	Archive
10 0001	10 0001	10 0001	10 0001	10 0001	10 0001
AND 00 0001	AND 00 0010	AND 00 0100	AND 00 1000	AND 01 0000	AND 10 0000
(1) 00 0001	(0) 00 0000	(0) 00 0000	(0) 00 0000	(0) 00 0000	(32)10 0000

To do this in Basic, use code similar to the following:

```

If TheAttribute = 0 Then
    REM No attributes set
Else
    If (TheAttribute AND 1) = 1 Then ... 'Read-Only file: bit 1 is set
    If (TheAttribute AND 16) = 16 Then ... 'Directory: bit 5 is set.
    If (TheAttribute AND 4) <> 0 Then ... 'Another way to code the same logic.
End If

```

Each file and directory has an attribute defined as these bit patterns. If a bit in the attribute that corresponds to a particular property is set, then the file has that property. Performing the AND operator with the individual bit positions determines if the file has that property. The function FileAttributeString in Listing 155 uses this method.

To set the archive bit and read-only bit on a file, combine the bits and call the function once. Use the OR operator to combine bit patterns. With the OR operator, if either bit is set, the resulting bit is a 1. To set the read-only and the archive bits, use “1 OR 32”. If you set the attributes to 1, then all of the other attributes will be cleared and only the read-only bit will be set.

8.5. Obtaining a directory listing

Use the Dir function to obtain a directory listing. The first argument is a file specification. Although a file or directory may be uniquely identified, file specs (also called wildcards) are allowed. For example, the command Dir("C:\temp*.txt") returns a list of all files that have the extension TXT. The second argument specifies attributes, for which there are two valid values: 0 (the default) returns files; set the second argument to 16 to retrieve a list of directories.

TIP

Most operating systems contain two special directory names, represented by a single period (.) and a double period (..). A single period references the current directory, and two periods reference the parent directory. These special directories are included in the directory list as returned by the Dir function. If you write a macro that looks at each directory recursively but you don't take these two into consideration, your macro will erroneously run forever.

The first call to Dir starts reading a directory and returns the first file that matches. Each additional call, which takes no arguments, returns the next file that matches.

```
sFileName = Dir(path, attribute)    'Get the first one
Do While (sFileName <> "")          'While something found
    sFileName = Dir()               'Get the next one
Loop
```

If the path uniquely identifies a file or directory, only one entry is returned. For example, the command Dir("C:\tmp\autoexec.bat") returns the single file "autoexec.bat". Less obviously, the command Dir("C:\tmp") returns the single directory "tmp". To determine what a directory contains, the path must either contain a file specifier (C:\tmp*.*) or the path must contain a trailing path separator (C:\tmp\). The code in Listing 158 performs a simple listing of the current directory; it uses the function GetPathSeparator to obtain the path separator in an operating-system-independent way.

Listing 158. List the files in the current directory.

```
Sub ExampleDir
    Dim s As String                'Temporary string
    Dim sFileName As String        'Last name returned from DIR
    Dim i As Integer               'Count number of dirs and files
    Dim sPath                      'Current path with path separator at end
    sPath = CurDir & GetPathSeparator() 'With no separator, DIR returns the
    sFileName = Dir(sPath, 16)     'directory rather than what it contains
    i = 0                          'Initialize the variable
    Do While (sFileName <> "")      'While something returned
        i = i + 1                 'Count the directories
        s = s & "Dir " & CStr(i) & _
            " = " & sFileName & CHR$(10) 'Store in string for later printing
        sFileName = Dir()         'Get the next directory name
    Loop
    i = 0                          'Start counting over for files
    sFileName = Dir(sPath, 0)      'Get files this time!
    Do While (sFileName <> "")
        i = i + 1
        s = s & "File " & CStr(i) & " = " & sFileName & " " & _
            PrettyFileLen(sPath & sFileName) & CHR$(10)
        sFileName = Dir()
    Loop
    MsgBox s, 0, ConvertToURL(sPath)
End Sub
```

Sample output from Listing 158 is shown in Figure 59. First, the directories are listed. The first two directories, "." and "..", represent the following:

```
file:///home/andy/My%20Documents/OpenOffice/
```


file:///home/andy/My%20Documents/

The inclusion of “.” and “..” is a common source of problems. A listing of directories contains these two directories, which should usually be ignored.

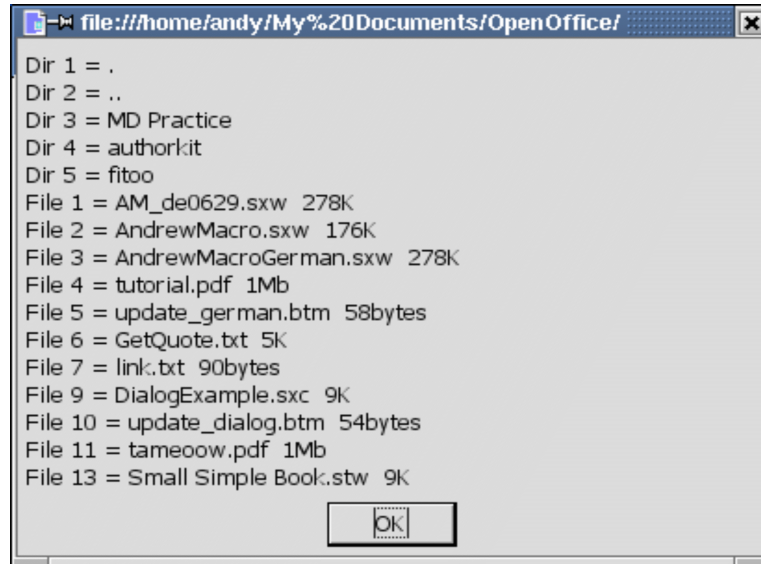


Figure 59. Directory listing of the current directory.

8.6. Open a file

LO uses low-level, system-specific methods to manipulate files. The operating system maintains a list of the open files and identifies them with a number, which is called a “file handle.” In the world of Basic, this is usually called the “file number” or “data channel.”

To open a file, you must tell it what file number (file handle) to use. Use the FreeFile function to obtain an unused file number, which will be used when opening a file, referencing the open file, and when closing the file. The Open statement is used to open a file before it is usable for reading or writing. The Open statement requires a file number, which should always be obtained from the FreeFile function. Use the Close statement when you are finished with the file. You can close multiple files in a single statement by providing a comma-separated list of numbers after the Close statement. Use the Reset function to close all of the open files at one time without having to explicitly list the file numbers. All open file numbers are closed and their data is flushed to disk.

```
n = FreeFile()  
Open FileName For Mode [Access ioMode] [Lock Mode] As #n [Len=DataLen]  
Close #n
```

“FileName” is the name of the file that you want to open. If the file name does not include the path, the current directory is assumed. “For Mode” specifies the state of the file when it is opened and how you intend to use the file (see Table 64).

Table 64. Valid “For Mode” values and the resulting configuration if Access is not used.

For Mode	File Pointer	File Exists	No File	Read	Write	Comment
For Append	end	Open	Create	Yes	Yes	Sequential access
For Input	start	Open	error	Yes	No	Sequential access
For Output	start	Delete	Create	Yes	Yes	Sequential access
For Binary	start	Delete	Create	Yes	Yes	Random access
For Random	start	Delete	Create	Yes	Yes	Random access

Each mode has its own set of behaviors as shown in Table 64. Consider the row For Input, this can be read to say: When a file is opened “For Input”:

1. The file pointer is positioned at the start of the file.
2. If the file exists, it is opened (and not deleted).
3. If the file does not exist, an error is generated.
4. The file is opened with read access, but not write access (assuming that Access is not explicitly provided).
5. Sequential access is used for reading the file.

Unfortunately, the precise implementation is dependent on the operating system and even the compiler used to create your version of OOO; for example, on some systems, you can write to a file opened for Input.

When a file is open, a pointer into the file is maintained. This pointer identifies where the next read or write operation will occur. For example, if the file pointer is at the start of the file, the next “read” command will read the first thing in the file. If the file pointer is at the end of the file, the next “write” command will append data to the end of the file. You have some control over the initial position of the file pointer when the file is opened, and you can move this file pointer around when the file is open. All of the “For” modes, except “For Append,” position the file pointer at the start of the file.

You can access a file sequentially or randomly. A sequential file is similar to a video tape. Although you can fast forward and rewind the tape to a specific location on the tape, the entire tape moves past the read/write head. You then press Play or Record and the data is either sequentially read from or written to the tape. A random file is similar to a music CD. Although you can play the CD sequentially, it isn’t required; you can quickly jump to any song and play it. To make the analogy more accurate, however, each song on the CD must be the same size. This is the disadvantage to the “For Random” mode.

Consider storing names of different lengths in a file on the disk. Storing one name per line in the file is efficient with respect to space. You can use a new-line character between each name. To find a specific name in the file, you start at the beginning and read until you find the person’s name. On the other hand, if you know that the longest name is 100 characters, you can store each name

on the disk, and store enough spaces after the name to use a total of 100 characters for each name. This wastes space, but it allows you to quickly move between names on the disk, because of the regular file structure. To read or write the 1000th name in the file, you simply move directly to that record. You have wasted space in this design, but you have gained speed performance. All of the “For” modes,” except “For Binary” and “For Random,” specify sequential file access. Random files use this ability to fix the record length to the maximum size of interest in order to permit very rapid file access and retrieval.

The access modes in Table 65 affect the default treatment of a file when it is opened. When an access mode is specified, it also verifies that you have access to either read or write the file. If you do not have write access to a file opened with “Access Write,” a run-time error occurs. The access mode affects every open “For” mode except “For Append” — which never deletes an existing file when it is opened.

Table 65. *Valid Access ioModes.*

Access ioMode	Description
Access Read	Do not delete an existing file. Verify that you have read access.
Access Write	Delete an existing file. Verify that you have write access.
Access Read Write	Delete an existing file. Verify that you have read and write access.

Using “Access Write” while opening a file “For Input” allows you to write to the file after it is opened; first the file is erased and then a new file is created. After the file is open, different operating systems enforce the access rights differently. As of OOo version 1.1.1, opening a binary or random file with “Access Read” allows you to write to the file when using Windows, but not on Linux. It is always safe to open a file “For Append” and then move the file pointer to the start of the file manually.

TIP The only safe way to open a file for both reading and writing without erasing the contents of the file is to open the file “For Append,” and then move the file pointer to the start of the file.

To limit access to a file while it’s open, use the “Lock” modes (see Table 66). This prevents others from reading and/or writing to the file while you have it open. This is primarily used in multi-user environments because you can’t control what others might try to do while you are using the file.

Table 66. *Valid protected keywords.*

Lock Mode	Description
Lock Read	Others cannot read the file while it’s open, but they can write to it.
Lock Write	Others cannot write the file while it’s open, but they can read from it.
Lock Read Write	Others cannot read or write the file while it is open.

Use the Len keywords to specify the size of each record when the file is opened “For Random” (discussed later).

8.7. Information about open files

Basic has functions that return file information by using the file name (see Listing 157). It is also possible to obtain information about open files from the file number. The FileAttr function returns how the file associated with the given file number was opened. Table 67 lists the return values and their meanings.

```
FileAttr(n, 1) 'How the file was opened in BASIC using Open For ...
```

Table 67. Description of FileAttr() return values.

Return Value	Description
1	Open For Input
2	Open For Output
4	Open For Random
8	Open for Append
16	Open For Binary
32	Documentation incorrectly states that this is for open for binary.

TIP

FileAttr is incorrectly documented. FileAttr(n, 2) does not return a system file handle, if the second argument is not 1, the return value is always 0. Another problem is that the included help incorrectly states that Binary has a return value of 32.

Use the EOF function to determine if the end of the file has been reached. A typical use is to read all of the data until “End Of File.”

```
n = FreeFile           'Always find the next free file number
Open FileName For Input As #n 'Open the file for input
Do While NOT EOF(n)    'While NOT End Of File
    Input #n, s         'Read some data!
    REM Process the input here!
Loop
```

Use the LOF function to determine the length of an open file. This number is always in bytes.

```
LOF (n)
```

Use the Loc function to obtain the current location of the file pointer. This number is not always accurate and the return value has a different meaning depending on how the file was opened. Loc returns the actual byte position for files opened in Binary mode. For Random files, Loc returns the record number of the last record read or written. For sequential files opened with Input, Output, or Append, however, the number returned by Loc is the current byte position divided by 128. This is done to maintain compatibility with other versions of BASIC.

```
Loc (n)
```

TIP

If a file is opened in a mode other than Random, and Basic considers the file a text file, Loc returns the line number that will be read next. I cannot decide if this is a bug or just incomplete documentation. Sometimes the return values from Loc are just wrong. For example, if you open a file for output and then write some text, Loc returns 0.

The Seek function, when used with only one argument, returns the next position that will be read or written. This is similar to the Loc function except that for sequential files, the absolute byte position is always returned. If you want to save the position in a file and return to it later, use the Seek function to obtain the current file pointer; then you can use the Seek function to return the file pointer to the original location.

```
position = Seek(n)      'Obtain and save the current position.
statements              'Do arbitrary stuff.
Seek(n, position)      'Move the file pointer back to the original position.
```

The argument for setting the file pointer using the Seek function is the same as the value returned by the Seek function. For Random files, the position is the number of the object to read, not the byte position. For sequential files, the position is the byte position in the file. The macro in Listing 159 returns information about an open file from the file number, including the open mode, file length, and file pointer location. Listing 160 uses Listing 159, and the result is shown in Figure 60.

Listing 159. *Return information about an open file as a string.*

```
Function GetOpenFileInfo(n As Integer) As String
    Dim s As String
    Dim iAttr As Integer
    On Error GoTo BadFileNumber
    iAttr = FileAttr(n, 1)
    If iAttr = 0 Then
        s = "File handle " & CStr(n) & " is not currently open" & CHR$(10)
    Else
        s = "File handle " & CStr(n) & " was opened in mode:"
        If (iAttr AND 1) = 1 Then s = s & " Input"
        If (iAttr AND 2) = 2 Then s = s & " Output"
        If (iAttr AND 4) = 4 Then s = s & " Random"
        If (iAttr AND 8) = 8 Then s = s & " Append"
        If (iAttr AND 16) = 16 Then s = s & " Binary"
        iAttr = iAttr AND NOT (1 OR 2 OR 4 OR 8 OR 16)
        If iAttr AND NOT (1 OR 2 OR 4 OR 8 OR 16) <> 0 Then
            s = s & " unsupported attribute " & CStr(iAttr)
        End If
        s = s & CHR$(10)
        s = s & "File length = " & nPrettyFileLen(LOF(n)) & CHR$(10)
        s = s & "File location = " & CStr(LOC(n)) & CHR$(10)
        s = s & "Seek = " & CStr(Seek(n)) & CHR$(10)
        s = s & "End Of File = " & CStr(EOF(n)) & CHR$(10)
    End If
AllDone:
    On Error GoTo 0
    GetOpenFileInfo = s
    Exit Function
BadFileNumber:
    s = s & "Error with file handle " & CStr(n) & CHR$(10) & _
        "The file is probably not open" & CHR$(10) & Error()
    Resume AllDone
End Function
```

TIP

The position argument passed to the Seek function is one-based, not zero-based. This means that the first byte or record is 1, not 0. For example, Seek(n, 1) positions the file pointer to the first byte or record in the file.

The macro in Listing 160 opens a file for output. A large amount of data is written to the file to give it some size. At this point, the Loc function returns 0 and EOF returns True. The Seek function is used to move the file pointer to a position in the file that allows some data to be read. The Loc function still returns 0. One hundred pieces of data are read from the file in order to advance the value returned by the Loc function. Finally, the file is deleted from the disk. Figure 60 shows information based on a file number.

Listing 160. Create delme.txt in the current directory and print file information.

```
Sub WriteExampleGetOpenFileInfo
    Dim FileName As String           'Holds the file name
    Dim n As Integer                'Holds the file number
    Dim i As Integer                'Index variable
    Dim s As String                 'Temporary string for input
    FileName = ConvertToURL(CurDir) & "/delme.txt"
    n = FreeFile()                  'Next free file number
    Open FileName For Output Access Read Write As #n 'Open for read/write
    For i = 1 To 15032              'Write a lot of data
        Write #n, "This is line ", CStr(i), "or", i 'Write some text
    Next
    Seek #n, 1022                   'Move the file pointer to location 1022
    For i = 1 To 100                'Read 100 pieces of data; this will set Loc
        Input #n, s                 'Read one piece of data into the variable s
    Next
    MsgBox GetOpenFileInfo(n), 0, FileName
    Close #n
    Kill(FileName)                  'Delete this file, I do not want it
End Sub
```

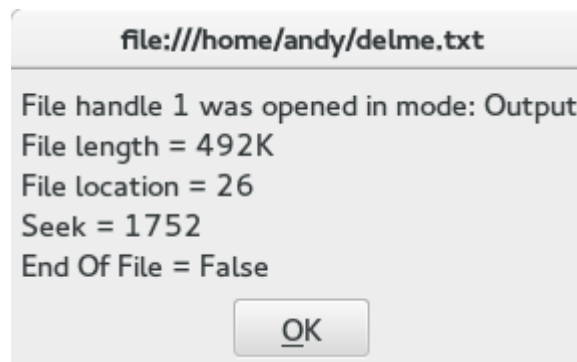


Figure 60. Information based on a file number.

8.8. Reading and writing data

Files opened for Random and Binary data use the statements Put and Get for writing and reading data. Files opened in any other mode use the Line Input, Input, Print, and Write statements for reading and writing. If no expressions are entered, a blank line is written to the file. The Write

statement accepts multiple arguments to print to the file and it automatically adds delimiters as it writes. In the created file, each expression is separated by a comma. Strings are enclosed in double quotation marks, numbers are not enclosed in anything, and dates and Boolean values are enclosed between octothorpe (#) characters.

```
Write #n, expression1, expression2, expression3, ...
Print #n, expression1, expression2, expression3, ...
```

TIP The character “#” has many names, including number sign, pound sign, hash, sharp, crunch, hex, grid, pigpen, tic-tac-toe, splat, crosshatch, and octothorpe, to name a few.

The Print statement does not write any useful delimiters. Instead, it writes spaces between each expression. Numbers typically use 13 spaces.

```
Write #n, i, "the time # is", Now, CDBl(1.221), CBool(0)
Print #n, i, "the time # is", Now, CDBl(1.221), Cbool(0)
```

The code above produces the text below.

```
0,"the time # is",#07/01/2010 21:05:49#,1.221,#False#
0          the time # is 07/01/2010 21:05:49 1.221          False
```

Listing 161 demonstrates the difference between Write and Print.

Listing 161. Demonstrate Write versus Print.

```
Sub ExampleWriteOrPrint
    Dim FileName As String           'Holds the file name
    Dim n As Integer                'Holds the file number
    Dim i As Integer                'Index variable
    Dim s As String                 'Temporary string for input
    Dim sTemp$
    FileName = ConvertToURL(CurDir) & "/delme.txt"
    n = FreeFile()                  'Next free file number
    Open FileName For Output Access Read Write As #n 'Open for read/write

    Write #n, i, "the time # is", Now, CDBl(1.221), CBool(0)
    Print #n, i, "the time # is", Now, CDBl(1.221), CBool(0)

    Seek #n, 1                     'Move the file pointer to location 1
    Line Input #n, s
    Line Input #n, sTemp
    s = s & CHR$(10) & sTemp
    MsgBox s
    Close #n
    Kill(FileName)                 'Delete this file, I do not want it
End Sub
```

As its name implies, the Line Input statement reads an entire line of text (see Listing 161), but it does not return the delimiter. Each line is delimited by either a carriage return (ASCII value 13) or a line-feed character (ASCII value 10). These two delimiters work to read lines on every operating system supported by LO.

```
Line Input #n, stringVar          'Read an entire line but not the delimiter.
```

The Input statement reads text based on the following delimiters: comma, carriage return, or line-feed characters. The Input statement can read multiple variables of differing types in a single command. Changing Line Input to Input in Listing 161 causes only the first two items to be read rather than two lines.

```
Input #n, var1, var2, var3, ...
```

The Write command adds appropriate delimiters automatically so that you can read string and numeric data into the appropriate variable types. The Input command automatically removes commas and double quotation marks from the input when these characters are used as the delimiters. See Listing 162 and Figure 61 for input examples.

Listing 162. Use Input to read text written with Write.

```
Sub ExampleInput
    Dim sFileName As String
    Dim n As Integer
    Dim t As String, d As Double, s As String
    sFileName = ConvertToURL(CurDir) & "/delme.txt"
    n = FreeFile()
    Open sFileName For Output Access Read Write As #n
    Write #n, 1.33, Now, "I am a string"
    Seek(n, 1)
    Input #n, d, t, s
    close #n
    Kill(sFileName)
    s = "string (" & s & ")" & CHR$(10) & _
        "number (" & d & ")" & CHR$(10) & _
        "time   (" & t & ") <== read as a string" & CHR$(10)
    MsgBox s, 0, "Example Input"
End Sub
```

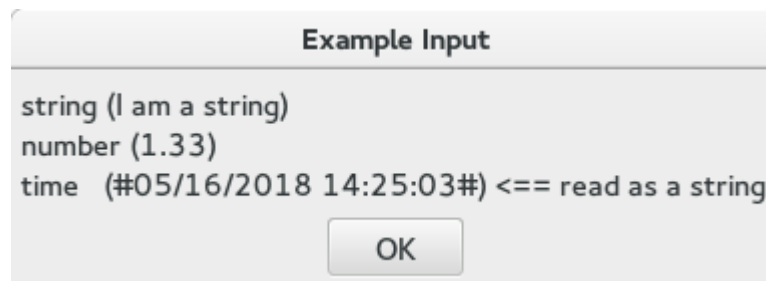


Figure 61. Input cannot read time delimited with “#”.

TIP

A friend in Germany had different results, because the number 1.33 is written as 1,33. While reading the values, the comma is seen as a delimiter rather than as part of the number.

Unfortunately, the delimiters produced by the Write statement are not supported by the Input statement. Numbers and simple strings read with no problems. Date and Boolean values delimited with the # character, however, fail. These values must be read into string variables and then parsed.

Do not use the Input statement if you do not have a lot of control over the input text file. Double-quotation marks and commas in text strings are assumed to be text delimiters. The end result is that the text is not properly parsed when it is read. If your input data may contain these characters, use the Line Input command and then manually parse the text. If you must read the carriage return or line-feed characters, the file should be read as a binary file.

A binary file is a random file with a block length of zero. Use the Put statement to write random and binary files. The simplest case involves putting a simple variable directly to the file.

```
int_var = 4 : long_var = 2
Put #n,,int_var      '04 00      (two bytes written)
Put #n,,long_var     '02 00 00 00 (four bytes written)
```

The first argument is the file number and the third argument is the variable or data to write. The second argument is the position in the file where the data should be written. If you omit the position, as shown in the example, you must still include the comma.

```
Put #n,,variable      'Write to the next record or byte position
Put #n, position, variable 'Specify the next record or byte position
```

Random files assume that the position identifies a record number. Binary files assume that the position identifies an absolute byte position. If the position is not specified, the data is written at the current file pointer, which is advanced with the data that is written.

If the data variable is a Variant, an integer identifying the data type precedes the data. This integer is the same integer returned by the VarType function, to be detailed later.

```
v = 4                'A Variant variable
Put #n,,v            '02 00 04 00 (first two bytes says type is 2)
Put #n,,4            '02 00 04 00 (first two bytes says type is 2)
Put #n,,CInt(4)      '02 00 04 00 (first two bytes says type is 2)
```

A string stored as a Variant includes the VarType if it is “Put” to a file that was opened as any type other than Binary. When an array is Put to a file, each element of the array is written. If the array contains a String Variant, it includes the VarType even if the file type is Binary. When Put places a string as a Variant, it actually writes the VarType, the string length, and then the string.

```
v() = Array("ABCD") 'ASCII in hexadecimal is 41 42 43 44
Put #n,,v()          '08 00 04 00 41 42 43 44 (08 00 = type) (04 00 = length)
```

When data is Put to a file, the current file pointer is saved and then all of the data is written. If a non-zero block length is used, the file pointer is positioned one block length past the saved file position regardless of how much data was written. For example, if the block length is 32 and the current file position is 64, the file pointer is positioned to byte 96 after the data is written. If more than 32 bytes are written, part of the next record is overwritten. If fewer than 32 bytes are written, then the previous data is left unchanged. Because of this, some people initialize every record that will be written when the file is created. Numeric values are usually initialized to zero, and string values are usually initialized to spaces.

On the other hand, even though I don't recommend it, you can use the Put statement with a file opened in a sequential mode. Likewise, you can use the Write, Line Input, and Input statements for files opened in Binary or Random mode. The actual bytes written for writing methods used for files of the "wrong" file structure are not documented, and I had difficulty understanding the output. I finally read the source code to determine what is written in each case, but undocumented behavior determined in this way should not be assumed to be stable, reliable behavior for Basic. If you want to use these methods for other file structures than those documented, I recommend that you test the output for your specific data. When a piece of data is written to a file, the specific context is used to determine what to write. See Table 68.

Table 68. Summary of what the Put statement writes.

Type	Bytes	Comment
Boolean	1	Basic stores a Boolean in an integer. The True value has all of the bits set, which incorrectly cast down to one byte, causing a run-time error. False writes with no problem.
Byte	3	Although the byte variable uses only one byte when writing the data, byte variables are supported only when stored in a Variant, so the data is preceded by two bytes of type information.
Currency	8	Internally stored as a Double.
Date	8	Internally stored as a Double.
Double	8	
Integer	2	
Long	4	
Object	Error	Run-time error: Only the basic types are supported.
Single	4	
String	Len(s)	Each character is one byte. Characters with an ASCII value larger than 255 are written with incomplete data. In Binary mode, the Put statement will not write characters with an ASCII value of zero. This is written fine in Random mode and the string is preceded by the string length.
Variant	Varies	Two bytes of type information are written, followed by the data. A string also includes the length of the string in two bytes.
Empty	4	An empty Variant variable writes two bytes of type information indicating an integer value, and then it writes two bytes with zeros.
Null	Error	Only an object can contain the value Null, and the Put statement does not work with objects.

TIP Basic only supports using Get and Put with the standard data types.

TIP Numbers written to binary files are written in reverse byte order.

TIP The Put statement cannot write a Boolean with a True value, and it doesn't properly write strings with Unicode values greater than 255; writing then trying to read the value caused OOo to crash when I last tested (years ago).

TIP

The Get statement fails for binary files if the position is not provided.

Use Get to read Binary data and Random files. The syntax for the Get statement is similar to the Put statement.

```
Get #n,,variable           'Read from next record or byte position
Get #n, position, variable 'Specify the next record or byte position
```

If the argument to the Get statement is a Variant, the type information is always read, regardless of the context. When a string is read, it is assumed to be preceded by an integer that contains the length of the string. This required string length is not written automatically to binary files but it is to random files. Listing 163 shows an example of reading and writing a binary file. Also see Figure 62.

Listing 163. Create and then read a binary file.

```
Sub ExampleReadWriteBinaryFile
    Dim sFileName As String 'File name from which to read and write
    Dim n As Integer        'File number to use
    Dim i As Integer        'Scrap Integer variable
    Dim l As Long           'Scrap Long variable
    Dim s As String         'Scrap String variable
    Dim s2 As String        'Another scrap String variable
    Dim v                   'Scrap Variant variable
    sFileName = ConvertToURL(CurDir) & "/delme.txt"
    If FileExists (sFileName) Then
        Kill(sFileName)
    End If
    n = FreeFile()
    Open sFileName For Binary As #n
    i = 10 : Put #n,,i      '0A 00
    i = 255 : Put #n,,i     'FF 00
    i = -2 : Put #n,,i      'FE FF
    l = 10 : Put #n,,l      '0A 00 00 00
    l = 255 : Put #n,,l     'FF 00 00 00
    l = -2 : Put #n,,l      'FE FF FF FF

    REM Put string data, precede it with a length
    i = 8 : Put #n,,i       '08 00 (about to put eight characters to the file)
    s = "ABCD"
    Put #n,,s               '41 42 43 44 (ASCII for ABCD)
    Put #n,,s               '41 42 43 44 (ASCII for ABCD)

    REM Put data contained in a Variant
    Put #n,,CInt(10)         '02 00 0A 00
    i = -2 : Put #n,,CInt(i) '02 00 FE FF      (Functions return a Variant)
    Put #n,,CLng(255)        '03 00 FF 00 00 00 (Functions return a Variant)
    v = 255 : Put #n,,v      '02 00 FF 00      (This IS a Variant)
    v = "ABCD" : Put #n,,v   '41 42 43 44      (Not in an array)
    v = Array(255, "ABCDE") 'The string contains type information and length
    Put #n,,v()              '02 00 FF 00 08 00 05 00 41 42 43 44 45
    close #n
```

```

REM now, read the file.
s = ""
n = FreeFile()
Open sFileName For Binary Access Read As #n
Get #n, 1, i : s = s & "Read Integer " & i & CHR$(10)
Get #n, 3, i : s = s & "Read Integer " & i & CHR$(10)
Get #n, 5, i : s = s & "Read Integer " & i & CHR$(10)
Get #n, 7, l : s = s & "Read Long " & l & CHR$(10)
Get #n, 11, l : s = s & "Read Long " & l & CHR$(10)
Get #n, 15, l : s = s & "Read Long " & l & CHR$(10)
Get #n, 19, s2 : s = s & "Read String " & s2 & CHR$(10)
close #n
MsgBox s, 0, "Read Write Binary File"
End Sub

```

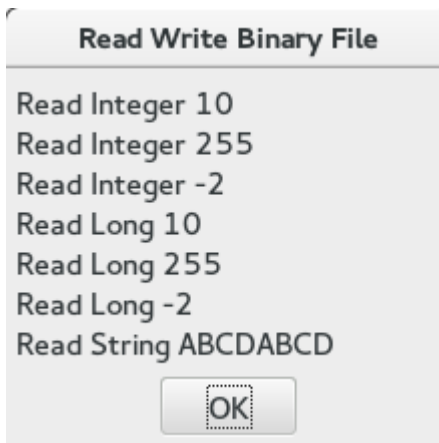


Figure 62. Use Get and Put to read and write binary files.

Random files are usually used to store a user-defined data type, but this is not supported in Basic; use “Open FileName For Random” to open a file for random access. The code in Listing 164 writes numerous types and sizes of data to the file with a block length of 8. If the Put statement had no bugs, then after writing the first block, the file pointer would be positioned to the second block for writing. It is instead positioned to the end of the file. To avoid this bug, explicitly include the position to write in the statement. If the position points to a position beyond the end of the file, the file pointer is moved to the end of the file. This is the primary reason why the code in Listing 164 initializes the file to all zeros before starting; the file is initialized with locations for the subsequent operations. Notice that the string includes the string length before the text when it is Put to the file. The output is essentially the same as Listing 163 shown in Figure 62.

Listing 164. Write and then read a random access file.

```

Sub ExampleReadWriteRandomFile
    Dim sFileName As String 'File name from which to read and write
    Dim n As Integer        'File number to use
    Dim i As Integer        'Scrap Integer variable
    Dim l As Long           'Scrap Long variable
    Dim s As String         'Scrap String variable
    Dim s2 As String        'Another scrap String variable

```

```

sFileName = ConvertToURL(CurDir) & "/delme.txt"

REM Now the file is initialized so it can be used!
REM Must use Access Read so that the file is not created new
REM I cannot write this as a binary file because then ASCII
REM zeros are not written.
n = FreeFile()
Open sFileName For Random As #n Len = 8
REM First, create a file with all zeros with enough room
REM for 20 8-byte records.
s = String(8 * 20-2, 0) 'String has 158 characters with ASCII value 0
Put #n,1,s 'Written as Random so Len(s) is written first
i = 0 : Put #n,1,i 'Write over the length with zeros.

REM Now write the data
i = 10 : Put #n,1,i '0A 00
i = 255 : Put #n,2,i 'FF 00
i = -2 : Put #n,3,i 'FE FF
l = 10 : Put #n,4,l '0A 00 00 00
l = 255 : Put #n,5,l 'FF 00 00 00
l = -2 : Put #n,6,l 'FE FF FF FF

REM Put string data, precede it with a length (integer value) automatically
s = "ABCD" : Put #n,7,s '04 00 41 42 43 44 (Length, then ASCII for ABCD)
close #n

REM Now read the file.
s = ""
n = FreeFile()
Open sFileName For Random Access Read As #n Len=8
Get #n, 1, i : s = s & "Read Integer " & i & CHR$(10)
Get #n, 2, i : s = s & "Read Integer " & i & CHR$(10)
Get #n, 3, i : s = s & "Read Integer " & i & CHR$(10)
Get #n, 4, l : s = s & "Read Long " & l & CHR$(10)
Get #n, 5, l : s = s & "Read Long " & l & CHR$(10)
Get #n, 6, l : s = s & "Read Long " & l & CHR$(10)
Get #n, 7, s2 : s = s & "Read String " & s2 & CHR$(10)
close #n
MsgBox s, 0, "Read Write Random File"
End Sub

```

8.9. File and directory related services

Some of the OO methods for file manipulation are buggy and unreliable. You may want to consider some of the built-in OO services; services are discussed later.

8.9.1. Path Settings and Singletons

Most of the macros in this chapter use the CurDir function to choose a place to store files. The PathSettings service provides read/write access (and the ability to register a listener) for the paths properties used by LO. But wait, the PathSettings service was deprecated with LO 4.3 and you should now use the “thePathSettings” singleton.

What is a Singleton?

A singleton restricts the creation of an object to a single object. So, no matter how many people ask for a singleton, they should always get the same singleton. For our purposes, the PathSettings service, which has been deprecated, is just an instance of the “thePathSettings” singleton. This is how you create a service.

```
oPathSettings = CreateUnoService( "com.sun.star.util.PathSettings" )
```

To get a copy of a singleton, you need to know the name of the singleton, and then you get it from the default context.

```
oContext = getProcessServiceManager().DefaultContext
oPathSettings = oContext.GetValueByName("/singletons/com.sun.star.util.thePathSettings")
```

Note that the singleton name is preceded by the text “/singletons/”.

Back to path settings

Although the documentation is not clear on this, my examples indicate that the path is returned as a URL. On the other hand, the PathSettings service uses the PathSubstitution service, which specifically states that it returns URLs. OOMEWorkDir in Listing 165 demonstrates obtaining the Work directory.

Listing 165. Determine the work directory to use.

```
Function OOMEWorkDir() As String
    Dim s$
    Dim oPathSettings
    Dim oContext

    ' Deprecated, use the singleton instead.
    'oPathSettings = CreateUnoService("com.sun.star.util.PathSettings")
    oContext = getProcessServiceManager().DefaultContext
    oPathSettings = oContext.GetValueByName("/singletons/com.sun.star.util.thePathSettings")
    s$ = oPathSettings.Work
    If s = "" Then
        s = GetPathSeparator()
    ElseIf Right(s,1) <> "/" AND Right(s,1) <> "\\\" Then
        If Left(s, 5) = "file:" Then
            s = s & "/"
        Else
            s = s & GetPathSeparator()
        End If
    End If
    OOMEWorkDir() = s & "OOMEWork" & GetPathSeparator()
End Function
```

A macro that creates temporary files or directories for example purposes will use Listing 166 to create and remove the working directory.

Listing 166. Create and remove the OOME work directory.

```
Function CreateOOMEWorkDir() As Boolean
    CreateOOMEWorkDir() = False
```

```

Dim s$
s = OOMEWorkDir()
If NOT FileExists(s) Then
    Mkdir s
End If
CreateOOMEWorkDir() = FileExists(s)
End Function

Function RmOOMEWorkDir() As Boolean
    RmOOMEWorkDir() = False
    Dim s$
    s = OOMEWorkDir()
    If FileExists(s) Then
        Rmdir s
    End If
    RmOOMEWorkDir() = NOT FileExists(s)
End Function

```

The documentation lists properties that are supported. By inspecting the object, I found more properties than those that are documented.

Table 69. Documented PathSettings properties.

Property	Number	Which Directory
Addin	Single	Contains spreadsheet add-ins that use the old add-in API.
AutoCorrect	Multiple	Contains the settings for the AutoCorrect dialog.
AutoText	Multiple	Contains the AutoText modules.
Backup	Single	Where automatic document backups are stored.
Basic	Multiple	Contains Basic files used by the AutoPilots.
Bitmap	Single	Contains the external icons for the toolbars.
Classification	Single	Not documented.
Config	Single	Contains configuration files. This property is not visible in the path options dialog and cannot be modified.
Dictionary	Single	Contains the dictionaries.
Favorite	Single	Contains the saved folder bookmarks.
Filter	Single	Where the filters are stored.
Fingerprint	Single	Not documented
Gallery	Multiple	Contains the Gallery database and multimedia files.
Graphic	Single	Displayed when the dialog for opening a graphic or saving a new graphic is used.
Help	Single	Contains the OOo help files.
Iconset	Single	Not documented
Linguistic	Single	Contains the OOo spellcheck files.
Module	Single	Contains the OOo modules.
Palette	Single	Contains the palette files that contain user-defined colors and patterns (*.SOB and *.SOF).
Plugin	Multiple	Contains the Plugins.

Property	Number	Which Directory
Storage	Single	Where information about mail, news, and FTP servers is stored.
Temp	Single	Contains the OOO temp-files.
Template	Multiple	Contains the OOO document templates.
UIConfig	Multiple	Global directories for user interface configuration files. The user interface configuration is merged with the user settings stored in the directory specified by UserConfig.
UserConfig	Single	Contains the user settings, including the user interface configuration files for menus, toolbars, accelerators and status bars.
UserDictionary	Single	Contains the custom dictionaries.
Work	Single	The work folder. This path can be modified according to the user's needs and can be seen in the Open or Save dialog.

To see the path settings on your computer, run the macro in Listing 167. On my computer, I find numerous extra paths such as Work_internal, Work_user, Work_writable.

DisplayPathSettings (see Listing 167) demonstrates numerous advanced techniques that are not explained in this chapter.

- Creating and using an OOO service.
- Creating a new document.
- Inserting text into a document.
- Setting paragraph style.
- Inserting paragraph breaks into a text object.

The following macro was originally written by Danny Brewer, who did much to advance the knowledge of OOO macros before he went on to other things. I modified the macro so that it declares all variables and handles property types returned as array values.

Listing 167. Display the PathSettings in a new text document.

```
Sub DisplayPathSettings
    Dim oPathSettings ' PathSettings service.
    Dim oPropertySetInfo ' Access the service properties.
    Dim aProperties ' Contains all of the service properties.
    Dim oDoc ' Reference a newly created document.
    Dim oText ' Document's text object.
    Dim oCursor ' Cursor in the text object.
    Dim oProperty ' A property of the service.
    Dim cPropertyName$ ' Property name.
    Dim cPropertyValue ' Property value may be an array or multiple strings.
    Dim aPaths ' The paths as an array.
    Dim cPath$ ' A single path from the array.
    Dim j As Integer ' Index variable.
    Dim i As Integer ' Index variable.
```



```

Dim oContext
oContext = getProcessServiceManager().DefaultContext
oPathSettings = oContext.getValueByName("/singletons/com.sun.star.util.thePathSettings")
'oPathSettings = CreateUnoService( "com.sun.star.util.PathSettings" )

' Example of how to get a single property you are after.
'oPathSettings.Work

' Get information about the properties of the path settings.
oPropertySetInfo = oPathSettings.getPropertySetInfo()

' Get an array of the properties.
aProperties = oPropertySetInfo.getProperties()

' Create an output document.
oDoc = StarDesktop.loadComponentFromURL( "private:factory/swriter", _
                                         "_blank", 0, Array() )

oText = oDoc.getText()
oCursor = oText.createTextCursor()

oText.insertString( oCursor, "Path Settings", False )
oCursor.ParagraphStyleName = "Heading 1"
oText.insertControlCharacter( oCursor, _
                             com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, False )

' Iterate over the array of properties,
' and write information about each property to the output.
For i = LBound( aProperties ) To UBound( aProperties )
    oProperty = aProperties( i )
    cPropertyName = oProperty.Name
    cPropertyValue = oPathSettings.getPropertyValue( cPropertyName )

    oText.insertString( oCursor, cPropertyName, False )
    oCursor.ParagraphStyleName = "Heading 3"
    oText.insertControlCharacter( oCursor, _
                                com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, False )
    If IsArray(cPropertyValue) Then
        ' Multiple URLs are sometimes returned as an array.
        aPaths = cPropertyValue
    ElseIf Len( cPropertyValue ) > 0 Then
        ' Multiple URLs are sometimes separated by a semicolon.
        ' Split them up into an array of strings.
        aPaths = Split( cPropertyValue, ";" )
    Else
        aPaths = Array()
    End If
    For j = LBound( aPaths ) To UBound( aPaths )
        cPath = aPaths( j )
        oText.insertString( oCursor, cPath, False )
        oText.insertControlCharacter( oCursor, _
                                    com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, False )
    Next

```

```

oText.insertControlCharacter( oCursor, _
    com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, False )
Next i
End Sub

```

Setting a path value is trivially done using either direct assignment, or by using the property set information object. OOo requires these values to be correct, so, if you write bad values, it will negatively affect OOo.

```

oPathSettings.Work = ConvertToUrl("C:\MyWorkDir")
oPathSettings.setPropertyValue("Work", "C:\MyWorkDir")

```

The properties supported by the PathSetting service are stored, at least on my computer, in the file: `openoffice.org/basis3.2/share/registry/schema/org/openoffice/Office/Common.xcs`. The documentation still references the old filename `Common.xml`, which caused me some level of consternation.

8.9.2. Path Substitution

My file `Common.xcs` contains entries such as `$(userpath)/store` and `$(work)`, which are very different from what is returned by the PathSettings service. This is because the path setting service changes shortcuts such as `$(userpath)` to the real value before returning a string. Analogously, it substitutes the shortcuts back into the string before storing the value. You can call the PathSubstitution service directly to make your own substitutions.

Table 70. Path substitution variables.

Name	Description
\$(inst)	Installation path of the OOo Basis layer.
\$(prog)	Program path of the OOo Basis layer.
\$(brandbaseurl)	Installation path of the the OOo Brand layer.
\$(user)	The user installation directory.
\$(work)	The work directory of the user; "MyDocuments" for Windows, the user's home-directory for Linux.
\$(home)	The user's home directory; Documents and Settings for Windows, the user's home-directory for Linux.
\$(temp)	The current temporary directory.
\$(path)	The value of PATH environment variable.
\$(lang)	The country code used by OOo; 01=english.
\$(langid)	The language code used by OOo; 1033=english us.
\$(vlang)	The language used by OOo as a string. Like "en-US" for a US English OOo.

Use `getSubstituteVariableValue` to convert one name at a time. If the name is not known, a run-time error occurs.

Listing 168. Substitute one variable with PathSubstitution.

```

Sub UsePathSubstitution()

```

```

Dim oPathSub      ' PathSubstitution service.
Dim names         ' List of names to substitute.
Dim subName$      ' Single name to check.
Dim i As Integer  ' Index variable.
Dim s$           ' Accumulate the value to print.

names = Array("inst", "prog", "user", "username", _
              "work", "home", "temp", "path", _
              "langid", "vlang")

oPathSub = CreateUnoService( "com.sun.star.util.PathSubstitution" )

' Use getSubstituteVariableValue with a single variable.
' Runtime error if the name is not know.

'Print oPathSub.getSubstituteVariableValue("${inst}")
For i = LBound(names) To UBound(names)
    subName = "$(" & names(i) & ")"
    s = s & names(i) & " = "
    s = s & oPathSub.getSubstituteVariableValue(subName) & CHR$(10)
Next
MsgBox s, 0, "Supported Names"
End Sub

```

Use substituteVariables to substitute multiple values at the same time. Use reSubstituteVariables to place the variable names back into a regular string.

Listing 169. ReSubstitute variables.

```

Sub UsePathReSubstitution()
    Dim oPathSub      ' PathSubstitution service.
    Dim s$           ' Accumulate the value to print.
    Dim sTemp$

    ' There are two variables to substitute.
    ' False means do not generate an error
    ' if an unknown variable is used.
    s = "${temp)/OOME/ or $(work)"
    sTemp = oPathSub.substituteVariables(s, False)
    s = s & " = " & sTemp & CHR$(10)

    ' This direction encodes the entire thing as though it were a single
    ' path. This means that spaces are encoded in URL notation.
    s = s & sTemp & " = " & oPathSub.reSubstituteVariables(sTemp) & CHR$(10)
    sTemp = "${temp)"
    s = s & sTemp & " = " & oPathSub.substituteVariables(sTemp, False) & CHR$(10)
    sTemp = "${work)"
    s = s & sTemp & " = " & oPathSub.substituteVariables(sTemp, False) & CHR$(10)
    MsgBox s
End Sub

```

As you can see from the figure, substituteVariables only works with a single substitutions; it used to work with multiple. Also, reSubstituteVariables no longer functions at all.

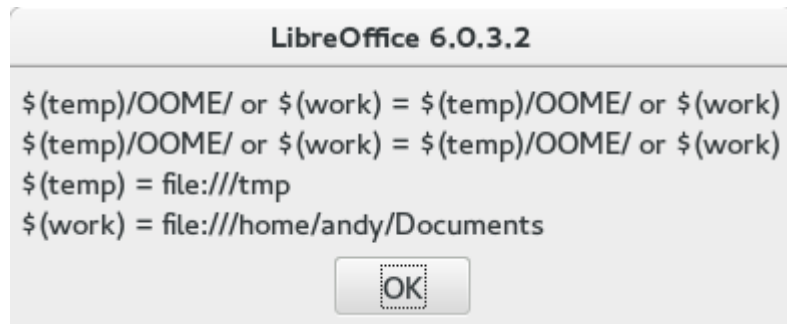


Figure 63. PathSubstitution service.

8.9.3. Simple File Access

Internally, LO uses the SimpleFileAccess service, rather than the file methods used by Basic, for file operations. The methods supported by the SimpleFileAccess service are shown in the following table.

Table 71. Methods supported by SimpleFileAccess.

Method	Description
copy(fromURL, toURL)	Copy a file.
move(fromURL, toURL)	Move a file.
kill(url)	Delete a file or directory, even if the folder is not empty.
isFolder(url)	Return true if the URL represents a folder.
isReadOnly(url)	Return true if the file is read-only.
setReadOnly(url, bool)	Set file as read-only if the boolean argument is true, otherwise, clear the read-only flag.
createFolder(url)	Creates a new Folder.
getSize(url)	Returns the size of a file as a long integer.
getContentType(url)	Return the content type of a file as a string. On my computer, an odt file has type application/vnd.sun.staroffice.fsyst-file.
getDateTimeModified(url)	Return the last modified date for the file as a com.sun.star.util.DateTime structure, which supports the properties: HundredthSeconds, Seconds, Minutes, Hours, Day, Month, and Year.
getFolderContents(url, bool)	Returns the contents of a folder as an array of strings. Each string is the full path as a URL. If the bool is True, then files and directories are listed. If the bool is False, then only files are returned.
exists(url)	Return true if a file or directory exists.
openFileRead(url)	Open file to read, return an input stream.
openFileWrite(url)	Open file to write, return an output stream.
openFileReadWrite(url)	Open file to read and write, return a stream.
setInteractionHandler(handler)	Set an interaction handler to be used for further operations. This is a more advanced topic and I will not discuss this here.
writeFile(toUrl, inputStream)	Overwrite the file content with the given data.

8.9.4. Streams

A stream supports reading and writing data from some input source to some output source that may go beyond a file system. As an example, I use streams to transfer entire files between the regular file system and a field in a database. In other words, streams are powerful and worth knowing and understanding. This section does not cover all of the stream capabilities and does not even touch on Markable streams and Object streams. Streams support all sorts of fancy things such as listeners that are automatically called when specific events occur.

TIP The `readLine()` method does not remove the end of line character if the end of file is reached.

Table 72. Stream methods.

Method	Stream	Description
<code>available()</code>	<code>InputStream</code>	Returns the number of available bytes as a long.
<code>closeInput()</code>	<code>InputStream</code>	Close the input stream.
<code>closeOutput()</code>	<code>OutputStream</code>	Close the stream.
<code>flush()</code>	<code>OutputStream</code>	Flush buffers.
<code>getLength()</code>	<code>XSeekable</code>	Get length of the stream.
<code>getPosition()</code>	<code>XSeekable</code>	Return the stream offset as a 64-bit integer.
<code>isEOF()</code>	<code>TextInputStream</code>	Returns true if the end of file has been reached.
<code>readBoolean()</code>	<code>DataInputStream</code>	Read an 8 bit value and return a byte. 0 means FALSE; all other values mean TRUE.
<code>readByte()</code>	<code>DataInputStream</code>	Read and return an 8 bit value and return a byte.
<code>readBytes(byteArray, long)</code>	<code>InputStream</code>	Read the specified number of bytes and return the number of bytes read. If the number of bytes requested was not read, then the end of file has been reached.
<code>readChar()</code>	<code>DataInputStream</code>	Read and return a 16-bit Unicode character.
<code>readDouble()</code>	<code>DataInputStream</code>	Read and return a 64-bit IEEE double.
<code>readFloat()</code>	<code>DataInputStream</code>	Read and return a 32-bit IEEE float.
<code>readHyper()</code>	<code>DataInputStream</code>	Read and return a 64-bit big endian integer.
<code>readLine()</code>	<code>TextInputStream</code>	Read text until a line break (CR, LF, or CR/LF) or EOF is found and returns it as string (without CR, LF).
<code>readLong()</code>	<code>DataInputStream</code>	Read and return a 32-bit big endian integer.
<code>readShort()</code>	<code>DataInputStream</code>	Read and return a 16-bit big endian integer.
<code>readSomeBytes(byteArray, long)</code>	<code>InputStream</code>	Read up to the specified number of bytes and return the number of bytes read. If the number of bytes read is zero, then the end of file has been reached.
<code>readString(charArray, boolean)</code>	<code>TextInputStream</code>	Read text until one of the given delimiter characters or EOF is found and returns it as string. The boolean argument determines if the delimiter is returned (false) or removed (true).
<code>readUTF()</code>	<code>DataInputStream</code>	Read and return a string of UTF encoded characters.
<code>seek (INT64)</code>	<code>XSeekable</code>	Change the stream pointer to the specified location.
<code>setEncoding(string)</code>	<code>TextInputStream</code>	Set the character encoding (see

Method	Stream	Description
		http://www.iana.org/assignments/character-sets).
skipBytes(long)	InputStream	Skips the specified number of bytes.
truncate()	XTruncate	Set the size of the file to zero.
writeBoolean(boolean)	DataOutputStream	Write a boolean as an 8 bit value. 0 means FALSE; all other values mean TRUE.
writeByte(byte)	DataOutputStream	Write an 8 bit value and return a byte.
writeBytes(byteArray())	OutputStream	Write all bytes to the stream.
writeChar(char)	DataOutputStream	Write a 16-bit Unicode character.
writeDouble(double)	DataOutputStream	Write a 64-bit IEEE double.
writeFloat(float)	DataOutputStream	Write a 32-bit IEEE float.
writeHyper(INT64)	DataOutputStream	Write a 64-bit big endian integer.
writeLong(long)	DataOutputStream	Write a 32-bit big endian integer.
writeShort(short)	DataOutputStream	Write a 16-bit big endian integer.

There are many different types of stream services and interfaces (see Table 72). A simple stream returned by SimpleFileAccess only supports reading and writing raw bytes and you must convert the data to an array of bytes. It is more common to manually create an appropriate stream (such as DataOutputStream or TextInputStream) and use that to wrap the simple stream returned by SimpleFileAccess.

Listing 170. *Using SimpleFileAccess to read and write text files.*

```
Sub ExampleSimpleFileAccess
    Dim oSFA          ' SimpleFileAccess service.
    Dim sFileName$    ' Name of file to open.
    Dim oStream       ' Stream returned from SimpleFileAccess.
    Dim oTextStream   ' TextStream service.
    Dim sStrings      ' Strings to test write / read.
    Dim sInput$       ' The string that is read.
    Dim s$            ' Accumulate result to print.
    Dim i%            ' Index variable.

    sStrings = Array("One", "UTF:Ãä", "1@3")

    ' File to use.
    sFileName = CurDir() & "/delme.out"

    ' Create the SimpleFileAccess service.
    oSFA = CreateUnoService("com.sun.star.ucb.SimpleFileAccess")

    'Create the Specialized stream.
    oTextStream = CreateUnoService("com.sun.star.io.TextOutputStream")

    'If the file already exists, delete it.
    If oSFA.exists(sFileName) Then
```

```

    oSFA.kill(sFileName)
End If

' Open the file for writing.
oStream = oSFA.openFileWrite(sFileName)

' Attach the simple stream to the text stream.
' The text stream will use the simple stream.
oTextStream.setOutputStream(oStream)

' Write the strings.
For i = LBound(sStrings) To UBound(sStrings)
    oTextStream.writeString(sStrings(i) & CHR$(10))
Next

' Close the stream.
oTextStream.closeOutput()

oTextStream = CreateUnoService("com.sun.star.io.TextInputStream")
oStream = oSFA.openFileRead(sFileName)
oTextStream.setInputStream(oStream)
For i = LBound(sStrings) To UBound(sStrings)
    sInput = oTextStream.readLine()
    s = s & CStr(i)

    ' If the EOF is reached then the new line delimiters are
    ' not removed. I consider this a bug.
    If oTextStream.isEOF() Then
        If Right(sInput, 1) = CHR$(10) Then
            sInput = Left(sInput, Len(sInput) - 1)
        End If
    End If

    ' Verify that the read string is the same as the written string.
    If sInput <> sStrings(i) Then
        s = s & " : BAD "
    Else
        s = s & " : OK "
    End If
    s = s & "(" & sStrings(i) & ")"
    s = s & "(" & sInput & ")" & CHR$(10)
Next
oTextStream.closeInput()
MsgBox s
End Sub

```



Figure 64. Text files with SimpleFileAccess.

8.9.5. Pipes

A pipe is an output stream and an input stream. Data written to the outputstream is buffered until it is read from the input stream. The Pipe service allows an outputstream to be converted into an input stream at the cost of an additional buffer. It is simple to create and close a pipe. Although CreatePipe in Listing 171 creates data streams, a very simple change would use a text stream instead.

Listing 171. *Create and close a pipe.*

```
Function CreatePipe() As Object
    Dim oPipe      ' Pipe Service.
    Dim oDataInp   ' DataInputStream Service.
    Dim oDataOut   ' DataOutputStream Service.

    oPipe      = createUNOService ("com.sun.star.io.Pipe")
    oDataInp   = createUNOService ("com.sun.star.io.DataInputStream")
    oDataOut   = createUNOService ("com.sun.star.io.DataOutputStream")
    oDataInp.setInputStream(oPipe)
    oDataOut.setOutputStream(oPipe)
    CreatePipe = oPipe
End Function

Sub ClosePipe(oPipe)
    oPipe.Successor.closeInput
    oPipe.Predecessor.closeOutput
    oPipe.closeInput
    oPipe.closeOutput
End Sub
```

TestPipes in Listing 172 uses a pipe to convert a byte array to a double and a double to a byte array.

Listing 172. *Convert a byte array to a double and a double to a binary array.*

```
Sub TestPipes
    Dim oPipe      ' Pipe service.
    Dim d As Double
    Dim i As Integer
    Dim s$

    oPipe = CreatePipe()
```



```

' First, write a series of bytes that represents 3.14159
oPipe.Predecessor.writeBytes(Array(64, 9, 33, -7, -16, 27, -122, 110))
d = 2.6      '4004CCCCCCCCCCCD
oPipe.Predecessor.writeDouble(d)

' Now, read the pipe.
d = oPipe.Successor.readDouble()
s = "Read the array of bytes as: " & CStr(d) & CHR$(10) & CHR$(10)

' Now read the double that was written as a series of bytes.
s = s & "2.6 = "
Do While oPipe.Successor.available() > 0
    i = oPipe.Successor.readByte()
    REM In case the byte was negative
    i = i AND 255
    If i < 16 Then s = s & "0"
    s = s & Hex(i) & " "
Loop
ClosePipe(oPipe)
MsgBox s
End Sub

```

8.10. Conclusion

The file and directory functions in Basic are able to manipulate directories and files. With the exception of reading and writing binary and random files, the directory and file manipulation functions work with few surprises. On the other hand, some of the functions are broken and have been broken for years; you may need to use more advanced methods, such as Streams and SimpleFileAccess for anything other than simple file reading and writing.

9. Miscellaneous Routines

This chapter introduces the subroutines and functions supported by OpenOffice.org Basic that do not easily fit into another larger category — for example, routines related to flow control, user input, user output, error handling, inspecting variables, color, and display — **as well as** a few routines that you should not use.

I was tempted to call this chapter “Leftovers” because it contains the routines that were left over after I grouped the others into chapters. Although the word “leftovers” frequently has a negative connotation, this is certainly not the case for the routines discussed in this chapter. The eclectic mix of routines includes some of the more interesting and useful routines that are varied enough to prevent boredom from lulling you off to sleep.

9.1. Display and color

The Basic functions related to color manipulations and determining screen metrics are shown in Table 73. The screen metrics provide the size of each pixel so that you can write macros to create objects at a given size and position objects more precisely.

Table 73. *Display- and color-related functions in Basic.*

Function	Description
Blue(color)	Get the blue component
GetGuiType	Get the GUI type: Mac, Windows, Unix
Green(color)	Get the green component
QBColor(dos_color)	Return RGB for standard color
Red(color)	Get the red component
RGB(red, green, blue)	RGB to colorNumber
TwipsPerPixelX	Width of each pixel in twips
TwipsPerPixelY	Height of each pixel in twips

9.1.1. Determine the GUI type

The GetGuiType function returns an integer corresponding to the graphical user interface (GUI). In other words, you can find out what type of computer is running the macro ... well, sort of. This function only mentions the GUI type, not the operating system — for example, just Windows, not Windows 98 or Windows XP. The function GetGuiType is only included for backward compatibility with previous versions of Basic.

One of my associates ran OpenOffice.org as a server on his computer at home. He connected to his home computer from work as a client. The value returned by GetGuiType was not defined while OOO is running in a client/server environment.

Table 74 shows the return values, as documented by the OOO help and seen in the source code as of version 3.2.1.

Table 74. Return values from *GetGuiType*.

#	OOo Help	Source Code
1	Windows	Windows (sometimes OS/2, which runs Windows)
2	Not mentioned	OS/2
3	Not mentioned	Used to be Mac, not returned.
4	Unix	Unix
-1	Mentioned as an undefined value	Unsupported OS

The macro in Listing 173 demonstrates the *GetGuiType* function.

Listing 173. Display the GUI type as a string.

```
Sub DisplayGuiType()
    Dim s As String
    Case 1
        s = "Windows" ' Documented and in code.
    Case 2
        s = "OS/2" ' No longer documented or set in code.
    Case 3
        s = "Mac OS" ' No longer documented or set in code., I expect Mac to return 4.
    Case 4
        s = "UNIX" ' Documented and in code.
    Case Else
        ' -1 is Not Documented, but is returned in the code if not Windows or Unix.
        s = "Unknown value " & CStr(GetGuiType()) & CHR$(10) & _
            "Probably running in Client/Server mode"
    End Select
    MsgBox "GUI type is " & s, 0, "GetGuiType()"
End Sub
```

The value -1 is returned if the type is not known, but that is not specifically documented. This probably means that you are running in a client / server mode, but, I have not checked that.

9.1.2. Determine pixel size (in twips)

Basic has two functions to determine the size of each display pixel (dot) in twips: *TwipsPerPixelX* and *TwipsPerPixelY*. The word “twip” is short for “twentieth of a PostScript point.” There are 72 PostScript points in an inch, thus 1440 twips in an inch.

In 1886, the American Typefounders Association proposed a unit of measure for typesetting called the “American Printer’s Point.” There are approximately 72.27000072 Printer’s Points in an inch. Years later while developing the PostScript page description language for Adobe Systems, Jim Warnock and Charles Geschke defined the PostScript point as exactly 72 points to an inch. When dot-matrix printers were released, they could print at either 10 or 12 characters per inch. Twips were created as a unit of measure that worked well for both dot-matrix printers and PostScript points.

TIP

There are 1440 twips in an inch. This number is important because OOo uses twips for many measurements.

Twips are the standard on which all Microsoft Windows graphics routines are based. Twips are used in the Rich Text Format, printer drivers, screen drivers, and many other products and platforms — including OpenOffice.org. The macro in Listing 174 obtains the number of twips per pixel in both the X and Y direction (horizontal and vertical) and displays the number of pixels per inch.

Listing 174. *Determine how many pixels per inch.*

```
Sub DisplayPixelSize
    Dim s As String
    s = s & TwipsPerPixelX() & " Twips per X-Pixel or " & _
        CStr(1440 \ TwipsPerPixelX()) & " X-Pixels per Inch" & CHR$(10)
    s = s & TwipsPerPixelY() & " Twips per Y-Pixel or " & _
        CStr(1440 \ TwipsPerPixelY()) & " Y-Pixels per Inch"
    MsgBox s, 0, "Pixel Size"
End Sub
```



Figure 65. *Number of pixels per inch on my computer.*

Unfortunately, it is not clear to me what is returned when there are multiple monitors with different values. The source code returns values from the “Default Device”.

9.1.3. Use color functions

Colors on computer monitors, digital cameras, scanners — and those seen by the human eye — are produced by adding the three primary colors of light: red, green, and blue (RGB). When printing or painting, color is produced by absorbing some colors and reflecting others. Color printing uses a different set of colors, called primary pigments: cyan, magenta, yellow, and black (CMYK). These two different systems are based on real physical models. The RGB model is based on how light combines to form colors. The CMYK model is based on what happens when you mix paint of different colors.

Basic uses the RGB model, allowing for 256 different shades of each of the primary colors. This number is stored as a Long Integer. Use the functions Red, Green, and Blue to extract the red, green, and blue components from a color in OOo. Use the RGB function to combine the individual color components and obtain the long integer used by OOo. The RGB function accepts three arguments, each representing one of the primary colors. Each of the color components must be a

value from 0 through 255. The RGB function performs no validity checking, so consider the results undefined if you break the rules. In summary, Basic represents RGB colors as a single integer; the functions Red, Green, and Blue extract the red, green and blue components; and the function RGB accepts the red, green, and blue components and returns the Basic representation of the RGB color.

```
Dim nRed As Integer           'Can only be 0 through 255
Dim nGreen As Integer         'Can only be 0 through 255
Dim nBlue As Integer          'Can only be 0 through 255
Dim nOOoColor As Long         'Can only be 0 through 16,581,375
nOOoColor = RGB(128, 3, 101) '8,389,477
nRed = Red(nOOoColor)         '128
nGreen = Green(nOOoColor)     '3
nBlue = Blue(nOOoColor)       '101
```

In the old days of DOS, BASIC supported 16 colors. In Table 75, which shows the color name and the DOS the number used by DOS to represent the color. The OOo Color column contains the corresponding number as represented by OOo. The Red, Green, and Blue columns contain the values returned by the corresponding Basic functions. The QBColor function is designed to accept the DOS Color as an argument and return the corresponding OOo Color.

Listing 175. Demonstrate QBColor values.

```
Sub DisplayQBColor
    Dim i%
    Dim s$
    For i = 0 To 15
        s = s & i & " = " & QBColor(i) & " = ("
        s = s & Red(QBColor(i)) & ", "
        s = s & Green(QBColor(i)) & ", "
        s = s & Blue(QBColor(i)) & ")"
        s = s & CHR$(10)
    Next
    MsgBox s
End Sub
```

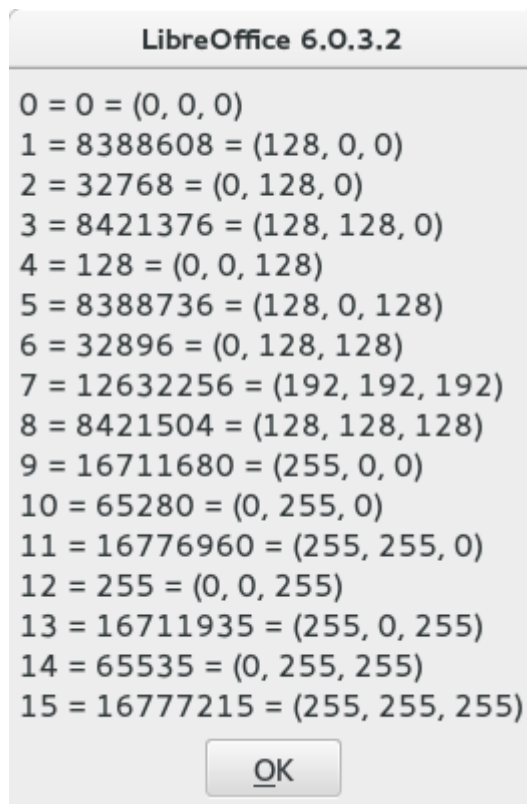


Figure 66. Results from QGColor.

Table 75. Color representation in LO.

DOS Color	OOo Color	Red	Green	Blue
0	0	0	0	0
4	128	0	0	128
2	32768	0	128	0
6	32896	0	128	128
1	8388608	128	0	0
5	8388736	128	0	128
3	8421376	128	128	0
8	8421504	128	128	128
7	12632256	192	192	192
12	255	0	0	255
10	65280	0	255	0
14	65535	0	255	255
9	16711680	255	0	0
13	16711935	255	0	255
11	16776960	255	255	0
15	16777215	255	255	255

9.2. Flow control

The flow-control functions listed in Table 76 either direct flow or provide functionality similar to flow control. For example, the IIF (immediate if) function provides functionality that would otherwise require an If-Then-Else statement.

Table 76. Basic functions related to flow control.

Function	Description
Choose(number, argument_list)	Flow control
IIF(condition, TrueExpression, FalseExpression)	Flow control
Stop	Stop execution now
Wait(milliseconds)	Pause the macro for a short time
WaitUntil(aTime)	Wait until a time has been reached.
Declare	Declare a DLL that you want to call
DDEExecute(nDDEChannel, command)	Execute a DDE command
DDEInitiate(Server, Topic)	Open a DDE channel
DDEPoke(nDDEChannel, item, data)	Set data on the DDE server through the channel
DDERequest(nDDEChannel, item)	Post a DDE request over an open channel
DDETerminateAll()	Close all DDE connections
FreeLibrary	Free a DLL library
Shell	Run a command through the command shell

9.2.1. Return an argument

The IIF function (Immediate If) and the Choose function both return an argument based on the value of the first argument. The IIF function returns one of two values based on a conditional expression. This works as a great one-line If-Then-Else statement.

```
max_age = IIF(johns_age > bills_age, johns_age, bills_age)
```

The IIF function takes three arguments. The first argument is a Boolean value, which determines what argument to return; one of the next two arguments is returned. Listing 176 shows how you can write the IIF function yourself.

Listing 176. The IIF function, if you wrote it yourself.

```
Function myIIF(conditional, true_arg, false_arg) As Variant
    If CBool(conditional) Then
        myIIF = true_arg
    Else
        myIIF = false_arg
    End If
End Function
```

All arguments to a routine are evaluated before the routine is called. Because IIF is a function, all arguments are evaluated when the function runs. With an If statement, the conditional code runs only if necessary. See Listing 177.

Listing 177. *If the denominator is zero, the division does not occur.*

```
If denominator <> 0 Then
    result = numerator / denominator
else
    result = 0
End If
```

In Listing 177, if the denominator is zero, the division is not done and zero is returned instead. In case your mathematical skills are a little rusty, it is not valid to divide a number by zero; a run-time error occurs. On the other hand, if your mathematical skills are not rusty, you know that returning zero when the denominator is zero isn't really correct, either. The macro in Listing 178 demonstrates that the functions f1 and f2 are both called, even though only the value from f2 is returned. In other words, IIF(x <> 0, 1/x, 0) causes a division-by-zero error if x is zero.

Listing 178. *All IIF arguments are called.*

```
Sub ExampleIIF
    REM This demonstrates that ALL expressions are evaluated
    REM This prints
    REM "I am in function f1"
    REM "I am in function f2"
    REM "f2"
    Print IIF(1>2, f1(), f2())
End Sub
Function f1$
    Print "I am in function f1"
    f1 = "f1"
End Function
Function f2$
    Print "I am in function f2"
    f2 = "f2"
End Function
```

The Choose function is similar to the IIF function in that it returns an argument based on the value in the first argument. However, it differs because it can have more than two possible return values, and the first argument is an integer, rather than a Boolean, that indicates which of those possibly many arguments to return.

```
Choose (expression, Select_1[, Select_2, ... [,Select_n]])
```

The Choose statement returns a null if the expression is less than 1 or greater than the number of selection arguments. Choose returns Select_1 if the expression evaluates to 1 and Select_2 if the expression evaluates to 2. The result is similar to storing the selections in an array with a lower bound of 1 and then indexing into the array. Each argument is evaluated regardless of which one is returned. See Listing 179.

Listing 179. *Demonstrate the Choose statement.*

```
Sub ExampleChoose
    Dim i%, v
    i% = CStr(InputBox("Enter an integer 1-4 (negative number is an error)"))
    v = Choose(i, "one", "two", "three", "four")
    If IsNull(v) Then
        Print "V is null"
    Else
        Print CStr(v)
    End If
End Sub
```

9.2.2. Pause or end the macro

The Stop command causes the macro to stop running. That's it, it is done, finished! You must start again. The Wait statement, on the other hand, only pauses macro execution (see Listing 180). After the specified number of milliseconds, the macro starts running again. A run-time error occurs if the number of milliseconds to wait is negative.

Listing 180. *Demonstrate the wait function.*

```
Sub ExampleWait
    On Error Goto BadInput
    Dim nMillis As Long
    Dim nStart As Long
    Dim nEnd As Long
    Dim nElapsed As Long
    nMillis = CLng(InputBox("How many milliseconds to wait?"))
    nStart = GetSystemTicks()
    Wait(nMillis)
    nEnd = GetSystemTicks()
    nElapsed = nEnd - nStart
    MsgBox "You requested to wait for " & nMillis & " milliseconds" & _
        CHR$(10) & "Waited for " & nElapsed & " milliseconds", 0, "Example Wait"
BadInput:
    If Err <> 0 Then
        Print Error() & " at line " & Erl
    End If
    On Error Goto 0
End Sub
```

In all of my experiments, the Wait statement has been accurate. The macro waits and then starts when it should.

WaitUntil is new and provides more compatibility with VB. The following code waits two seconds.

```
WaitUntil Now + TimeValue("00:00:02")
```

9.2.3. Dynamic Link Libraries

While a macro runs, it may directly call numerous subroutines and functions. A macro can also call routines and applications that are not related to OpenOffice.org. A Dynamic Link Library (DLL) is a collection of routines, or programs, that can be called when required. Each DLL is

packaged into a single file with the extension DLL — the file does not always have the suffix “.dll”, but it usually does. There are two very nice things about DLL files: Many programs can share a single DLL, and they are not usually loaded into memory until they are needed. The usage of DLLs promotes code reuse and does not waste memory. To tell Basic about a routine in a DLL, use the Declare statement.

TIP DLLs are not supported on Linux and I use Linux, so, I have not tested using DLLs in the latest versions of LO.

```
Declare Sub Name Lib "LibName" (arguments)
Declare Function Name Lib "LibName" (arguments) As Type
```

LibName is the name of the DLL that contains the routine named Name. It is common to use a DLL that you did not write, so you often have no control over the name of the routine that you call. Naming can be a problem if your macro already contains a routine with the same name or if you call another routine by the same name in another DLL. As a result, the Declare statement supports the Alias keyword, which you can use to overcome this hurdle. In this case, the RealName is the name used by the DLL, and myName is the name used by your macro.

```
Declare Sub myName Lib "LibName" Alias "RealName" (arguments)
Declare Function myName Lib "Libname" Alias "RealName" (arguments) As Type
```

For functions, the type declaration should use the standard types. You must, of course, know the type so that you can declare it. The argument list contains the arguments that are passed to the external routine. You must use the keyword ByVal if an argument is passed by value rather than by reference. Listing 181 calls a DLL.

TIP By default Basic passes arguments by reference. This means that if the called subroutine changes the argument, it is also changed in the calling program. The ByVal keyword causes an argument to be passed by value rather than by reference.

Listing 181. *Call a DLL. This will only work on windows (and if the DLL is present).*

```
Declare Sub MyMessageBeep Lib "user32.dll" Alias "MessageBeep" ( Long )
Declare Function CharUpper Lib "user32.dll" Alias "CharUpperA" _
    (ByVal lpsz As String) As String

Sub ExampleCallDLL
    REM Convert a string to uppercase
    Dim strIn As String
    Dim strOut As String
    strIn = "i Have Upper and Lower"
    strOut = CharUpper(strIn)
    MsgBox "Converted:" & CHR$(10) & strIn & CHR$(10) & _
        "To:" & CHR$(10) & strOut, 0, "Call a DLL Function"
    REM On my computer, this plays a system sound
    Dim nBeepLen As Long
    nBeepLen = 5000
    MyMessageBeep(nBeepLen)
    FreeLibrary("user32.dll" )
```

End Sub

A DLL is not loaded until a routine in the DLL is called. To remove the DLL from memory, use the FreeLibrary statement. The FreeLibrary statement accepts one argument: the name of the DLL to unload.

9.2.4. Calling external applications

Use the Shell statement to run an external application. The Shell command is disabled for users connecting by a virtual portal unless they happen to be the same user that started OOo in the first place. This statement does not obtain any information from an application; it simply runs another application or command.

```
Shell(Pathname, Windowstyle, Param, bSync)
```

TIP The Shell command is a potential security hole.

The only required argument is the first; the rest are optional. The first argument is the full path name of the external application. The application path may be in URL notation, but need not be. The Shell statement has problems if the path or application name contains a space. You can solve this problem the same way that your Web browser solves it: Substitute “%20” for each space. The ASCII value of a space is 32, which is 20 in hexadecimal. This technique can also be used to substitute other characters if they cause you problems.

```
Shell("file:///C:/Andy/My%20Documents/oo/tmp/h.bat",2) 'URL notation uses /
Shell("C:\Andy\My%20Documents\oo\tmp\h.bat",2) 'Windows notation uses \
```

The second argument (optional) indicates the window style of the started application. Table 77 lists valid values for the Shell window style argument.

Table 77. Window styles for the Shell statement.

Style	Description
0	Focus is on the hidden program window.
1	Focus is on the program window in standard size.
2	Focus is on the minimized program window.
3	Focus is on the maximized program window.
4	Standard size program window, without focus.
6	Minimized program window, but focus remains on the active window.
10	Full-screen display.

The third argument (optional) is a string that is passed to the application. Each space in the argument string is read by the called application as delimiting a separate argument. To pass an argument with an embedded space, place an extra set of double quotation marks around the arguments.

```
Shell("/home/andy/foo.ksh", 10, ""one argument" another") ' two arguments
```

TIP

The string ""one argument"" another" is correct and as intended; think about it!

The final optional argument determines if the Shell command returns immediately while the application is running (the default behavior) or if it waits until the application is finished. Setting the final argument to True causes the Shell statement to wait.

```
Sub ExampleShell
    Dim rc As Long
    rc = Shell("C:\andy\TSEProWin\g32.exe", 2, "c:\Macro.txt")
    Print "I just returned and the returned code is " & rc ' rc = 0
    Rem These two have spaces in their names
    Shell("file:///C:/Andy/My%20Documents\oo\tmp\h.bat", 2)
    Shell("C:\Andy\My%20Documents\oo\tmp\h.bat", 2)
End Sub
```

The Shell function returns a long with the value zero. If the program does not exist, a run-time error occurs and nothing is returned. When some applications run, they return a value that can be used to indicate an error. This value is not available from the Shell command. Intuitively, it isn't possible to obtain the final return value from an application when the Shell function returns before the application is finished running.

TIP

In Visual Basic, the arguments for the Shell function are different: Shell(path, window_style, bsync, Timeout). The Timeout value indicates how long to wait for the external application to finish. Arguments follow the application name as part of the same string, delimited by spaces.

VB does not use a separate argument to specify the arguments to send to the application launched by the Shell command. Instead, the arguments follow the name of the application, separated by spaces, inside the same quotation marks that contain the function path and name. This method also works with Basic, as an alternative way to specify the Shell command arguments. If only function, arguments, and window style are required, this alternative way to write the Shell command allows you to have identical statements for VB or OOo routines. If you want to specify bsync or Timeout arguments, the VB and OOo environments are not compatible.

```
Shell("/home/andy/foo.ksh hello you") ' two arguments, "hello" and "you"
```

9.2.5. Dynamic Data Exchange

TIP

All DDE commands are deprecated and have been for over 10 years. I do not recommend that you use them (see https://bz.apache.org/ooo/show_bug.cgi?id=39574) .

Dynamic Data Exchange (DDE) is a mechanism that allows information to be shared between programs. Data may be updated in real time or it may work as a request response.

Although the DDE commands accepted by a DDE server are specific to the individual server, the general syntax is the same. Most DDE commands require a Server, Topic, and Item. The Server is the DDE name for the application that contains the data. The Topic, usually a file name, contains

the location of the referenced Item. The example in Listing 182 uses the DDE function in a Calc spreadsheet to extract data in cell A1 from an Excel spreadsheet.

Listing 182. *Use DDE as a Calc function to extract cell A1 from a document.*

```
=DDE("soffice";"/home/andy/tstdoc.xls";"a1") 'DDE in Calc to obtain a cell
='file:///home/andy/TST.sxc'#$sheet1.A1      'Direct reference to a cell
```

The second line shows how a cell can be directly referenced in another Calc spreadsheet without using DDE. Basic supports DDE commands directly (see Table 78).

Table 78. *DDE commands supported by Basic.*

Command	Description
DDEExecute(nDDEChannel, command)	Execute a DDE command.
DDEInitiate(Server, Topic)	Open a DDE channel and return the channel number.
DDEPoke(nDDEChannel, item, data)	Set data on the DDE server.
DDERequest(nDDEChannel, item)	Post a DDE request over an open channel.
DDETerminateAll()	Close all DDE connections.

First, the DDEInitiate function makes a connection to the DDE server. The first argument is the name of the server — for example, “soffice” or “excel”. The second argument specifies the channel to use. A common value for a channel is a file name. The opened channel is identified by an integer, which is returned by the DDEInitiate command. A channel number of 0 indicates that the channel was not opened. Attempting to establish a DDE connection to OOo for a file that is not currently open returns 0 for the channel. See Listing 183.

Listing 183. *Use DDE to access a Calc document.*

```
Sub ExampleDDE
    Dim nDDEChannel As Integer
    Dim s As String
    REM OOo must have the file open or the channel will not be opened
    nDDEChannel = DDEInitiate("soffice", "c:\TST.sxc")
    If nDDEChannel = 0 Then
        Print "Sorry, failed to open a DDE channel"
    Else
        Print "Using channel " & nDDEChannel & " to request cell A1"
        s = DDERequest(nDDEChannel, "A1")
        Print "Returned " & s
        DDETerminate(nDDEChannel)
    End If
End Sub
```

The usable commands and syntax for each server are server dependent, so a complete coverage of DDE is beyond the scope of this book.

TIP

When I last tested Listing 183, it ran, returned a value, then crashed OOo.

9.3. User input and output

Basic provides a very simple mechanism for presenting information to the user and obtaining information from the user at run time (see Table 79). These routines aren't used to access files; they're used strictly for user input from the keyboard and output to the screen.

Table 79. User input and output functions.

Function	Description
InputBox(Msg, Title, Default, x_pos, y_pos)	Request user input as a String.
MsgBox (Message, Type, Title)	Display a message in a nice dialog.
Print expression1; expression2, expression3;...	Print expressions.

9.3.1. Simple output

Use the Print statement for simple, single-line output. A list of expressions follows the Print statement. If two expressions are separated by a semicolon, the expressions are printed adjacent to each other. If two expressions are separated by a comma, the expressions are printed with a tab between them; you cannot change the size of the tabs.

```
Print expression1, expression2, ... ' Print with tabs between expressions
Print expression1; expression2; ... ' Print with nothing between expressions
Print 1,Now;"hello","again";34.3    ' Mixing semicolons and commas is fine
```

Arguments are converted to a locale-specific string representation before they are printed. In other words, dates and numbers appear as you expect based on the locale set in your configuration (Tools | Options | Language Settings | Languages). Boolean values, however, always print the text True or False.

The help included with OOO lists two special expressions that work with the Print statement: Spc and Tab. In Basic, the Spc function works the same as the Space function. It accepts one numeric argument and returns a string composed completely of spaces. The Tab function, although documented, does not exist. See Listing 184 for a Print example.

Listing 184. Demonstrate the Spc() function.

```
Sub ExamplePrint
    Print "It is now";Spc(12);Now()
End Sub
```

The Print statement is usually used for simple, single-line output while debugging, because it allows you to stop a macro from running by clicking the Cancel button (see Figure 67). This is a great tool for debugging. Place a Print statement before or after lines that may cause a problem. If the values don't appear to be correct, you can click the Cancel button to stop the macro.

TIP

The print statement is nice because you can cause a macro to stop running by clicking Cancel. An advantage, and a disadvantage, however, is that it leaves the user in the IDE at the print statement.

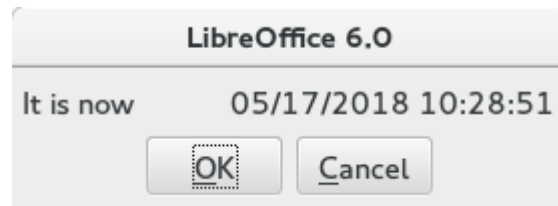


Figure 67. The Spc function returns a string with spaces.

When using many Print statements for debugging purposes, print explanatory information with the data to remind yourself what it is.

```
Print "Before the loop, x = ";x
For i=0 To 10
    Print "In the loop, i = ";i;" and x = ";x
```

When you print a string that contains a new-line character (ASCII 10 or 13), a new dialog is displayed for each new line. The code in Listing 185 displays three consecutive dialogs with text that reads “one”, “two”, and “three”. The Print dialog is able to print using more than one line. If a single line of text becomes too long, the line wraps and appears on more than one line. In other words, although Print will wrap by itself, the user has no way to force a new line in the dialog.

Listing 185. *New lines print in their own dialog.*

```
Print "one" & CHR$(10) & "two" & CHR$(13) & "three" ' Displays three dialogs
```

The Print statement uses a simple, defined protocol for numeric formatting. Positive numeric expressions contain a leading space. Negative numeric expressions contain a leading minus sign. Numbers with a decimal are printed in exponential notation if they become too large.

The Print statement displays a print dialog each time, unless the statement ends with either a semicolon or a comma. In this case it stores the text from each Print statement and adds to it until it encounters a Print statement that doesn’t end with a semicolon or a comma.

```
Print "one", 'Do not print yet, ends with a comma
Print "two" 'Print "one    two"
Print "three", 'Do not print yet, ends with a comma
Print "four"; 'Do not print yet, ends with a semicolon
Print          'Print "three    four"
```

9.3.2. Multi-line output

The MsgBox statement provides more control over the dialog that is displayed than does the Print statement, but can print only one string expression at a time. String expressions that contain a new-line character (ASCII 10 or 13) are printed in the same dialog. Each new-line character starts a new line in the dialog.

Listing 186. *Display a simple message box with a new line.*

```
Sub ExampleMsgBoxWithReturn
    MsgBox "one" & CHR$(10) & "two"
End Sub
```



Figure 68. This simple MsgBox dialog contains only an OK button.

The dialog in Figure 68 is very simple. The MsgBox function accepts two new arguments, as shown in Listing 187. The DialogTitle is displayed as the title line of the dialog. Valid values for DialogType are shown in Table 80. The DialogType determines which buttons are displayed in the dialog, which button is the default button, and which icon is displayed on the dialog.

Listing 187. The MsgBox function can accept a type and a dialog title.

```
MsgBox(Message)
MsgBox(Message, DialogType)
MsgBox(Message, DialogType, DialogTitle)
```

Table 80. Valid values for DialogType.

Value	Description
0	Display OK button only.
1	Display OK and Cancel buttons.
2	Display Abort, Retry and Ignore buttons.
3	Display Yes, No, and Cancel buttons.
4	Display Yes and No buttons.
5	Display Retry and Cancel buttons.
16	Add the Stop icon to the dialog.
32	Add the Question icon to the dialog.
48	Add the Exclamation Point icon to the dialog.
64	Add the Information icon to the dialog.
128	First button in the dialog is the default button. This is the default behavior.
256	Second button in the dialog is the default button.
512	Third button in the dialog is the default button.

Listing 188. Demonstrate the MsgBox type behavior.

```
Sub MsgBoxExamples()
    Dim i%
    Dim values
    values = Array(0, 1, 2, 3, 4, 5)
    For i = LBound(values) To UBound(values)
        MsgBox ("Dialog Type: " + values(i), values(i))
    Next
    values = Array(16, 32, 48, 64, 128, 256, 512)
    For i = LBound(values) To UBound(values)
```



```

    MsgBox ("Yes, No, Cancel, with Type: " + values(i), values(i) + 3)
Next
End Sub

```

You can use more than one `DialogType` at the same time to achieve your desired buttons, icon, and default behavior. The display choices are encoded in the first four bits (values 0-15 are binary 0000-1111); the icons and default behavior are encoded in higher bits (for example, 64 is 01000000 in binary). To combine attributes, use OR or add the values together. (This is similar to the behavior described for file attributes.)

Although you can display a dialog with a Cancel button, this won't cause the macro to stop running, as happens with the Print statement. Instead, the `MsgBox` function returns an integer that identifies the selected button (see Table 81). Clicking the Cancel button returns the value 2, and the Abort button returns a 3, which can then be tested by your code; then you (in your code) can decide whether or not you really want to cancel the macro.

Table 81. Return values from `MsgBox`.

Value	Description
1	OK
2	Cancel
3	Abort
4	Retry
5	Ignore
6	Yes
7	No

In other words, if you want the macro to stop when a user clicks the Cancel button, you must check the return value, as demonstrated in Listing 189. The message contains a new-line character so the message contains two lines of text. The dialog type requests three buttons and an icon, and sets the second button to be default (see Error: Reference source not found). The macro does different things based on the selected button.

Listing 189. Demonstrate how `MsgBox` works.

```

Sub ExampleMsgBox
    Dim nReturnCode As Integer
    Dim nDialogType As Integer
    Dim sMessage As String
    sMessage = "An error occurred!" & CHR$(10) & "Do the important work anyway?"
    REM 3 means Yes, No, Cancel
    REM 48 displays an Exclamation Point icon
    REM 256 second button is default.
    nDialogType = 3 OR 48 OR 256
    nReturnCode = MsgBox(sMessage, nDialogType, "My Title")
    If nReturnCode = 2 Then
        Print "Stopping the macro now!"
    Stop
End Sub

```

```

ElseIf nReturnCode = 6 Then
    Print "You chose Yes"
ElseIf nReturnCode = 7 Then
    Print "You chose No"
Else
    Print "I will never get here!", nReturnCode
End If
Print "Ready to exit the subroutine"
End Sub

```

9.3.3. Prompting for input

Use the `InputBox` function to prompt users for input. You can specify the dialog title. If a Default value is provided, it's displayed in the input box. The displayed dialog contains a text-input box, an OK button, and a Cancel button. The `InputBox` function returns a string to the calling statement. Clicking Cancel returns a zero-length string.

```

InputBox(Message)
InputBox(Message, Title)
InputBox(Message, Title, Default)
InputBox(Message, Title, Default, x_pos, y_pos)

```

The position arguments are in twips and are relative to the upper-left corner of the current window; one inch is 1440 twips. If the position is not specified, the dialog is centered both horizontally and vertically over the current window. The example in Listing 190 displays the input box two inches from the left edge of the window and four inches from the top. The size of the `InputBox` is defined automatically from the Message and buttons; `OOo` configures the layout of this box, as it does for the other basic input and output dialogs.

Listing 190. Demonstrate InputBox.

```

Sub ExampleInputBox
    Dim sReturn As String      'Return value
    Dim sMsg As String         'Holds the prompt
    Dim sTitle As String       'Window title
    Dim sDefault As String     'Default value
    Dim nXPos As Integer       'Twips from left edge
    Dim nYPos As Integer       'Twips from top edge
    nXPos = 1440 * 2           'Two inches from left edge of the window
    nYPos = 1440 * 4           'Four inches from top of the window
    sMsg = "Please enter some meaningful text"
    sTitle = "Meaningful Title"
    sDefault = "Hello"
    sReturn = InputBox(sMsg, sTitle, sDefault, nXPos, nYPos)
    If sReturn <> "" Then
        REM Print the entered string surrounded by double quotes
        Print "You entered """;sReturn;""""
    Else
        Print "You either entered an empty string or chose Cancel"
    End If
End Sub

```

Error: Reference source not found shows the dialog when it is first displayed. Pressing any key replaces the default text, because this text is highlighted when the dialog first opens. The macro in Listing 190 inspects the return value and checks for a zero-length string. A zero-length string could mean that the Cancel button was used to close the dialog, or it could mean that the user entered a zero-length string and then used the OK button to close the dialog. These two cases are not distinguishable from each other.

9.4. Error-related routines

The error-related routines (see Table 82) in Basic return information related to the last error. These routines are used to determine what happened and where. The error information is reset when the error handler is cleared, so save the error information if your macro program will require it later.

Table 82. Error functions in Basic.

Function	Description
CVErr	Convert an expression to an error object.
Erl	Line number of last error.
Err	Error number of last error.
Error Error(error_number)	Get error message either for the last error or for the specified error message.

The macro in Listing 191 checks for an error before the error handler is reset. Error information is then saved. Although the macro in Listing 191 does not save the error message, the Error function accepts an error number as an optional argument for which the error message is returned. Also see Figure 69.

Listing 191. Demonstrate error related statements.

```
Sub ExampleError
    On Error Goto BadError          'Set up the error handler
    Print 1/ CInt(0.2)              'Do a division by zero
BadError:                          'Error handler starts here
    Dim s As String                 'Accumulate the message
    Dim oldError As Integer         'Save the error number
    Dim lineNum As Integer         'Save the line number
    If Err <> 0 Then                 'If an error occurred
        oldError = Err             'Save the error number
        lineNum = Erl              'Save the error line number
        s = "Before clearing the error handler" & CHR$(10) & _
            "Error number " & Err & " Occurred at line number " & Erl & CHR$(10) & _
            "Err message: " & Error() & CHR$(10)
    End If
    On Error Goto 0                 'Reset the error handler

    REM There is now no information to print
    s = s & CHR$(10) & "After clearing handler:" & CHR$(10) & _
```

```

"Error number " & Err & " Occurred at line number " & Erl & CHR$(10)

REM Use the saved information
s = s & CHR$(10) & "Error info was saved so:" & CHR$(10) & _
  "Error number " & oldError & " Occurred at line number " & _
  lineNum & CHR$(10) & "Err message: " & Error(oldError)
MsgBox s, 0, "Error Handling"
End Sub

```



Figure 69. Error information must be saved if it is used after resetting the error handler.

9.5. Miscellaneous routines

The miscellaneous routines described in this section are general-purpose routines that aren't necessarily related to each other (see Table 83).

Table 83. Miscellaneous functions in Basic.

Function	Description
Beep	Make a system-dependent beep.
CBool(expression)	Convert an integer or string to Boolean.
Environ(string)	Return an environment variable.
GetSolarVersion	Internal running version.
CreateObject(obj_type)	Dynamic version of "Dim As New".
Erase(obj)	Free an objects memory.

The Beep statement generates a system-dependent beep. You can't change the pitch or the length of the generated beep. On some systems, this plays a configurable sound file through the built-in speaker, and on others it generates a system event that plays a system-defined sound through low-level internal hardware.

```

Beep          'Generate a noise

```

```
Wait(500)      'Wait 1/2 second
Beep           'Generate a noise
```

Use the CBool function to convert a string or a number to a Boolean value. Any numeric expression that evaluates to 0 returns False. Numeric expressions that are not 0 return True. String expressions that evaluate to “True” or “False” return True or False respectively; case does not matter. A string that does not evaluate exactly to True or False is evaluated as a number. If the string doesn’t contain a number or the text “true” or “false”, a run-time error occurs.

```
Print CBool(False)      'False
Print CBool(13)          'True
Print CBool("13")       'True
Print CBool("trUe")     'True
Print CBool("&h1")       'True
Print CBool("13xx")     'run-time error
Print CBool("Truee")    'run-time error
```

Use the Environ function to retrieve environment variables. If the environment variable does not exist, an empty string is returned. No method is provided to set or change environment variables.

```
Print Environ("PATH")
Print Environ("TEMP")
```

Use getSolarVersion to obtain the integer internal build number of OOo. You can write your macro to work around known issues, based on release documentation or discovered bugs for different versions of OOo.

```
Print GetSolarVersion
```

The CreateObject function allows objects to be created dynamically. If an object can be created using “Dim v As New”, it can be created with the CreateObject function. In other words, if an object can be created as a user-defined type, CreateObject can create one. The OOo underlying special data objects, which are covered in depth later, are called Universal Network Objects (UNO). These objects cannot be created using CreateObject. OOo does define structures that are not UNO objects that can be created using Dim As New and CreateObject (see Listing 192).

Listing 192. *Create an object using CreateObject or Dim As New.*

```
Dim oProp As New com.sun.star.beans.PropertyValue
Dim o As Object
o = CreateObject("com.sun.star.beans.PropertyValue")
```

Listing 192 demonstrates creating a variable type defined by OOo that is like a user-defined type. The actual type name of the object is “com.sun.star.beans.PropertyValue”. Many of the objects in OOo have similarly long and cumbersome names. While writing about or discussing variable types such as this, it’s common to abbreviate the type name as the last portion of the name. For example, set the Name property of the PropertyValue variable (see Listing 193). Objects of type PropertyValue have two properties: Name as a String and Value as a Variant.

Listing 193. *Dim a PropertyValue and use CreateObject to create a new one.*

```
Dim aProp As New com.sun.star.beans.PropertyValue
aProp.Name = "FirstName"      'Set the Name property
```

```

aProp.Value = "Paterton"      'Set the Value property
Erase aProp
Print IsNull(aProp)           'True

REM Create a new one!
Dim aPropr
aPropr = CreateObject("com.sun.star.beans.PropertyValue")
Erase aProp
Print IsNull(aProp)           'True
Print IsEmpty(aProp)          'False

Dim a
a = array("hello", 2)
Erase a
Print IsNull(a)               'False
Print IsEmpty(a)              'True

Dim b() As String
ReDim b(0 To 1) As String
b(0) = "Hello" : b(1) = "You"
'b() = "hello"                'Runtime error, variable not set (Expected)
'Print b()                    'Runtime error, variable not set (Expected)
'Erase b()                    'Syntax Error, not too surprised
Erase b                        'I did not expect this to work.
Print IsNull(b())              'False
Print IsEmpty(b())             'False
Print IsArray(b())             'False, This is probably bad.
'Print LBound(b())            'Error, variable not set.
b() = "hello"                  'Odd, now I can treat b() as a string variable
Print b()                      'hello

```

Listing 193 demonstrates the Erase statement, introduced with OOO version 2.0. Use the Erase statement to free memory that will no longer be used. Do not use Erase unless you are finished with the variable.

Use the CreateObject function to create an object dynamically — in other words, when you don't want to create the object when it's declared. You can use CreateObject to create only one object at a time. Use the Dim As New construction to create an array of a particular type (see Listing 194). You can even change the dimension of the array and preserve the data. It is more cumbersome to declare an array and then fill it with the appropriate values individually (see Listing 195).

Listing 194. Demonstrate ReDim with Preserve

```

Sub ExampleReDimPreserveProp
    REM this is easy to create this way
    Dim oProps(2) As New com.sun.star.beans.PropertyValue
    oProps(0).Name = "FirstName" : oProps(0).Value = "Joe"
    oProps(1).Name = "LastName"  : oProps(1).Value = "Blather"
    oProps(2).Name = "Age"       : oProps(1).Value = 53
    ReDim Preserve oProps(3) As New com.sun.star.beans.PropertyValue
    oProps(3).Name = "Weight"    : oProps(3).value = 97
    Print oProps(2).Name        'Age
End Sub

```

Listing 195. *You can add PropertyValue variables to a declared array.*

REM This is more cumbersome, but you can still do it...

```
Dim oProps(2)
oProps(0) = CreateObject("com.sun.star.beans.PropertyValue")
oProps(1) = CreateObject("com.sun.star.beans.PropertyValue")
oProps(2) = CreateObject("com.sun.star.beans.PropertyValue")
oProps(0).Name = "FirstName" : oProps(0).Value = "Joe"
oProps(1).Name = "LastName"  : oProps(1).Value = "Blather"
oProps(2).Name = "Age"       : oProps(1).Value = 53
```

Assigning one array to another assigns a reference so that both arrays reference the same array object. With variable types such as Integer and PropertyValue, assignment makes a copy. Failure to understand which types copy by value and which types copy by reference is a common source of errors. Structures and integral types (such as Integer and String) copy as a value, but arrays and UNO variables, as will be discussed later, copy as a reference. Copying by value is demonstrated in an obvious way in Listing 196.

Listing 196. *Properties copy by value.*

```
Sub ExampleCopyAsValue
    Dim aProp1
    Dim aProp2
    aProp1 = CreateObject("com.sun.star.beans.PropertyValue")
    aProp1.Name = "Age"      'Set Name Property on one
    aProp1.Value = 27        'Set Value Property on one
    aProp2 = aProp1         'Make a copy
    aProp2.Name = "Weight"  'Set Name Property on two
    aProp2.Value = 97       'Set Value Property on two
    Print aProp1.Name, aProp2.Name 'Age Weight
End Sub
```

TIP

Standard object variables copy by value, and UNO variables copy by reference.

When one integer variable is assigned to another, it is understood that the value was copied and nothing more. The two variables are still independent of each other. This is also true for structures. Text cursors, discussed later, contain a property called CharLocale, which specifies the country and language for the text selected by the text cursor. The common, incorrect method to set the locale is to access the variable directly. This sets the language and country on a copy of the CharLocale property rather than on the copy used by the text cursor. I see this type of error often.

```
oCursor.CharLocale.Language = "fr" 'set language to French on a copy
oCursor.CharLocale.Country = "CH"  'set country to Switzerland on a copy
```

One correct method to set the locale is to create a new Locale structure, modify the new structure, and copy the new structure to the text cursor.

```
Dim aLocale As New com.sun.star.lang.Locale
aLocale.Language = "fr"      'Set Locale to use the French language
aLocale.Country = "CH"       'Set Locale to use Switzerland as the country
oCursor.CharLocale = aLocale 'Assign the value back
```

You can also obtain a copy of the structure, modify the copied structure, and copy the modified structure to the text cursor.

```
Dim aLocale
aLocale = oCursor.CharLocale 'Or use a copy
aLocale.Language = "fr"      'Set Locale to use the French language
aLocale.Country = "CH"       'Set Locale to use Switzerland as the country
oCursor.CharLocale = aLocale 'Assign the value back
```

9.6. Partition

Partition is not documented and was likely added for VB compatibility. Partition return a Variant (String) indicating where a number occurs in a calculated series of ranges.

```
Partition(number, start_value, stop_value, interval)
```

Consider the following values:

```
start_value = 0
stop_value = 17
interval = 5
```

The following “partitions” are assumed:

- 1) “:-1” Everything before 0
- 2) “0:4” Five numbers from 0 to 4.
- 3) “5:9” Five numbers from 5 to 9.
- 4) “10:14” Five numbers from 10 to 14.
- 5) “15:17” Three numbers from 15 to 17.
- 6) “18: ” Everything after 17.

The example in Listing 197 tests numbers before and after the interval. As expected, values before the first interval run from “:-1”. Values that fall into an interval are nicely identified. The only tricky part is that the final interval claims to include values from “15:19” even though 18 and 19 are not in the interval.

Listing 197. Run partition through a series of values.

```
Sub ExamplePartition
    Dim i%
    Dim s$
    For i = -2 To 20
        s = s & "Partition(" & i & ", 0, 17, 5) = " & _
            Partition(i, 0, 17, 5) & CHR$(10)
    Next
    MsgBox s
End Sub
```

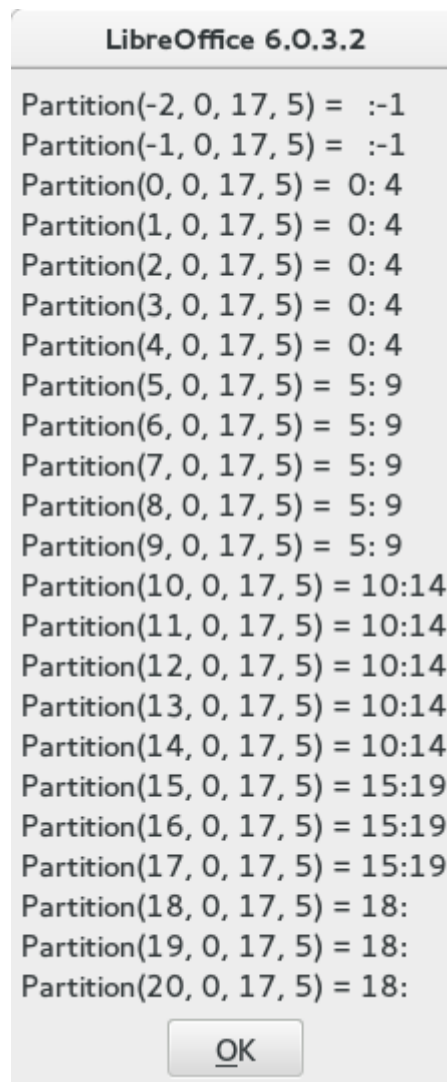



Figure 70. Partition shows the interval containing a number.

The values returned by Partition are carefully formatted. The lower value and upper value have the same number of characters, which means that they will sort properly if you choose to do so. This also helps if you want to parse the returned values.

9.7. Inspection and identification of variables

Basic contains numerous functions to inspect and identify variables (see Table 84). These routines are frequently used when you call a function and aren't certain of the return type. These routines are also useful for debugging. For example, you could use these functions to verify that a return type is valid.

Table 84. Variable inspection functions in Basic.

Function	Description
IsArray	Is the variable an array?
IsDate	Does the string contain a valid date?
IsEmpty	Is the variable an empty Variant variable?
IsMissing	Is the variable a missing argument?
IsNull	Is the variable an unassigned object?
IsNumeric	Does the string contain a valid number?
IsObject	Is the variable an object?
IsUnoStruct	Is the variable a UNO structure?
TypeLen	Space used by the variable type.
TypeName	Return the type name of the object as a String.
VarType	Return the variable type as an Integer.

Use `IsArray` to see if a variable is an array (see Listing 198). If `IsArray` returns `True`, it does not imply that the variable has data or even that it is dimensioned — merely that it exists and is defined as an array. The `UBound` and `LBound` functions, as already discussed, return the upper and lower bounds of the array.

Listing 198. Use `IsArray` to see if a variable is an array.

```
Dim n As Long      'This is NOT an array
Dim a() As String  'This is an array
Dim b(5)           'This is an array
Dim v As Variant   'This is not an array yet
Print IsArray(v)    'False
Print IsArray(n)    'False
Print IsArray(a)    'True
Print IsArray(b())  'True
ReDim v(3)          'It is an array now!
Print IsArray(v())  'True
```

Use the `IsDate` function to test if a string contains a valid date (see Listing 199). The arguments are converted to a string before they are used, so numeric arguments return `False`. The `IsDate` function tests more than just syntax; it checks to see if the string contains a valid date. The same check is not made on the time component of the string.

Listing 199. `IsDate` verifies that a string contains a valid date.

```
Print IsDate("December 1, 1582 2:13:42") 'True
Print IsDate("2:13:42")                  'True
Print IsDate("12/1/1582")                 'True
Print IsDate(Now)                         'True
Print IsDate("26:61:112")                  'True
Print IsDate(True)                        'False converts to string first
Print IsDate(32686.22332)                  'False converts to string first
Print IsDate("02/29/2003")                'False, only 28 days in February 2003
```

Like the `IsDate` function, the `IsNumeric` function looks at strings (see Listing 200). If the argument is not entirely a solitary valid number, except for leading or trailing spaces and enclosing quotation marks, it returns `False`.

Listing 200. *IsNumeric is very picky about what it accepts.*

```
Print IsNumeric(" 123") 'True
Print IsNumeric(" 12 3") 'False
Print IsNumeric(1.23) 'True
Print IsNumeric(1,23) 'True
Print IsNumeric("123abc") 'False
Print IsNumeric(True) 'False
Print IsNumeric(Now) 'False
```

Variant variables start with no value at all; they are initially empty. Object variables are initialized with the value `null`. Use the functions `IsEmpty` and `IsNull` to test these conditions. Use the `IsObject` function to determine if a variable is an object.

```
Dim v As Variant 'Starts as not initialized, Empty
Dim o As Object 'Initialized to null

Print IsObject(v) 'False No, a Variant is not an Object
Print IsObject(o) 'True Yes, this is an Object

Print IsEmpty(v) 'True Variants start as Empty, not initialized
Print IsNull(v) 'False To be null, a Variant must contain something

Print IsEmpty(o) 'False Variants start as Empty, not Objects
Print IsNull(o) 'True Objects start as null

v = o
Print IsObject(v) 'True The variant just became an Object.
Print IsEmpty(v) 'False Variant now contains a value (an Object)
Print IsNull(v) 'True Variant contains a null Object
```

Use the `IsMissing` function to determine if an optional argument is missing. Usually, some default value is used if an argument is missing.

```
Sub TestOptional
    Print "Arg is ";ExampleOptional() 'Arg is missing
    Print "Arg is ";ExampleOptional("hello") 'Arg is hello
End Sub

Function ExampleOptional(Optional x) As String
    ExampleOptional = IIF(IsMissing(x), "Missing", CStr(x))
End Function
```

Use the function `IsUnoStruct` to determine if a variable contains a structure defined by OpenOffice.org.

```
Dim v
Print IsUnoStruct(v) 'False
v = CreateUnoStruct("com.sun.star.beans.Property") 'Create a UNO
Print IsUnoStruct(v) 'True
```

Use the TypeName function for a string representation of the variable's type. The VarType function returns an integer corresponding to the variable type. Table 85 contains a list of the possible types. The first column, labeled BASIC, indicates if the type is common to BASIC and therefore likely to be seen. The other values represent types that are internal to OOo. Basic typically maps internal types to standard Basic types, so you aren't likely to see these other types. They are, however, contained in the source code and they are included for completeness.

Table 85. Variable types and names.

BASIC	VarType	TypeName	Len	Description
yes	0	Empty	0	Variant variable is not initialized
yes	1	Null	0	No valid data in an Object variable
yes	2	Integer	2	Integer variable
yes	3	Long	4	Long Integer variable
yes	4	Single	4	Single floating-point variable
yes	5	Double	8	Double floating-point variable
yes	6	Currency	8	Currency variable
yes	7	Date	8	Date variable
yes	8	String	strlen	String variable
yes	9	Object	0	Object variable
no	10	Error	2	Internal OOo type
yes	11	Boolean	1	Boolean variable
yes	12	Variant	0	Variant variables act like any type
no	13	DataObject	0	Internal OOo type
no	14	Unknown Type	0	Internal OOo type
no	15	Unknown Type	0	Internal OOo type
no	16	Char	1	Internal OOo type, a single text character
yes	17	Byte	1	Internal OOo type, but you can use CByte to create one
no	18	UShort	2	Internal OOo type, unsigned short integer (16 bits)
no	19	ULong	4	Internal OOo type, unsigned long (32 bits)
no	20	Long64	8	Internal OOo type, long (64 bits)
no	21	ULong64	8	Internal OOo type, unsigned long (64 bits)
no	22	Int	2	Internal OOo type, integer (16 bits)
no	23	UInt	2	Internal OOo type, unsigned integer (16 bits)
no	24	Void	0	Internal OOo type, no value
no	25	HResult	0	Internal OOo type
no	26	Pointer	0	Internal OOo type, pointer to something
no	27	DimArray	0	Internal OOo type
no	28	CArray	0	Internal OOo type
no	29	Userdef	0	Internal OOo type, user-defined
no	30	Lpstr	strlen	Internal OOo type, long pointer to a string

BASIC	VarType	TypeName	Len	Description
no	31	Lpwstr	strlen	Internal OOo type, long pointer to a “Wide” Unicode string
no	32	Unknown Type	0	Internal core string type
no	33	WString	strlen	Internal OOo type, “Wide” Unicode string
no	34	WChar	2	Internal OOo type, “Wide” Unicode character
no	35	Int64	8	Internal OOo type, integer (64 bits)
no	36	UInt64	8	Internal OOo type, unsigned integer (64 bits)
no	37	Decimal		OLE Automation type available in VB.

Use the TypeLen function to determine how many bytes a variable uses. The returned value is hard coded for every value except for the strings, which return the length of the string. Array variables always return a length of zero. The macro in Listing 201 generates all of the standard BASIC types, places them into an array, and builds a string containing the type, length, and type name, as shown in Figure 71.

Listing 201. *Show type information for standard types.*

```

Sub ExampleTypes
    Dim b As Boolean
    Dim c As Currency
    Dim t As Date
    Dim d As Double
    Dim i As Integer
    Dim l As Long
    Dim o As Object
    Dim f As Single
    Dim s As String
    Dim v As Variant
    Dim n As Variant
    Dim ta()
    Dim ss$
    n = null
    ta() = Array(v, n, i, l, f, d, c, t, s, o, b, CByte(3) _
        CreateUnoValue("unsigned long", 10)))
    For i = LBound(ta()) To UBound(ta())
        ss = ss & ADPTypeString(ta(i))
    Next
    MsgBox ss, 0, "Type, Length, and Name"
End Sub

Function ADPTypeString(v) As String
    Dim s As String           'Hold the return string
    Dim i As Integer          'Dummy integer value
    s = s & "Type = "         'Leading type string
    i = VarType(v)            'Get the type
    If i < 10 Then s = s & "0" 'Leading zero if required
    s = s & CStr(i)           'Add the type number
    If IsArray(v) Then

```

```

s = s & " ("
i = i AND NOT 8192
If i < 10 Then s = s & "0" 'Leading zero if required
s = s & CStr(i) & ")" 'Add the type number
Else
s = s & "    Len = " 'Leading length string
i = TypeName(v) 'What is the length
If i < 10 Then s = s & "0" 'Leading zero if required
s = s & CStr(i) 'Add in the length
End If
s = s & " Name = " 'Leading Name string
s = s & TypeName(v) & CHR$(10) 'Add in the name and a new-line character
ADPTypeString = s 'Return value for the function
End Function

```

Type, Length, and Name		
Type = 00	Len = 00	Name = Empty
Type = 01	Len = 00	Name = Null
Type = 02	Len = 02	Name = Integer
Type = 03	Len = 04	Name = Long
Type = 04	Len = 04	Name = Single
Type = 05	Len = 08	Name = Double
Type = 06	Len = 08	Name = Currency
Type = 07	Len = 08	Name = Date
Type = 08	Len = 00	Name = String
Type = 09	Len = 00	Name = Object
Type = 11	Len = 01	Name = Boolean
Type = 17	Len = 01	Name = Byte
Type = 09	Len = 00	Name = Object

OK

Figure 71. Variable types, lengths, and names.

The function ADPTypeString does the work of generating the display string. It gives special handling to arrays, because the returned type numbers for arrays are completely different than the type numbers for the standard variables. At least they seem that way until you start looking very closely at the numbers. If the thought of twiddling bits makes you tremble, skip the rest of this paragraph. The value returned by VarType for an array always has bit 14 set — a 1 followed by 13 zeros in binary. This is 8192 in decimal and 2000 in hexadecimal. The IsArray function is implemented by checking bit 14 of the VarType. If you clear bit 14, the number that remains tells you what numeric type is used for the array. The NOT operator clears every bit that is set and sets every bit that is clear, so NOT 8192 provides the number with every bit set except for bit 14. If you AND this with the type, it clears bit 14, leaving the rest of the bits intact.

```
i = i AND NOT 8192
```

The length of an array is always returned as zero, so I didn't include this by VarType in Listing 201. The code in Listing 202 is similar to Listing 201 but the types are arrays. Notice that the type name for array types contains parentheses “()” following the name (see Figure 72).

Listing 202. Show type information for arrays.

```
Sub ExampleTypesArray
    Dim b() As Boolean
    Dim c() As Currency
    Dim t() As Date
    Dim d() As Double
    Dim i() As Integer
    Dim l() As Long
    Dim o() As Object
    Dim f() As Single
    Dim s() As String
    Dim v() As Variant
    Dim ta(), j%
    Dim ss$
    ta() = Array(i, l, f, d, c, t, s, o, b, v)
    For j% = LBound(ta()) To UBound(ta())
        ss = ss & ADPTypeString(ta(j%))
    Next
    MsgBox ss, 0, "Type, Length, and Name"
End Sub
```

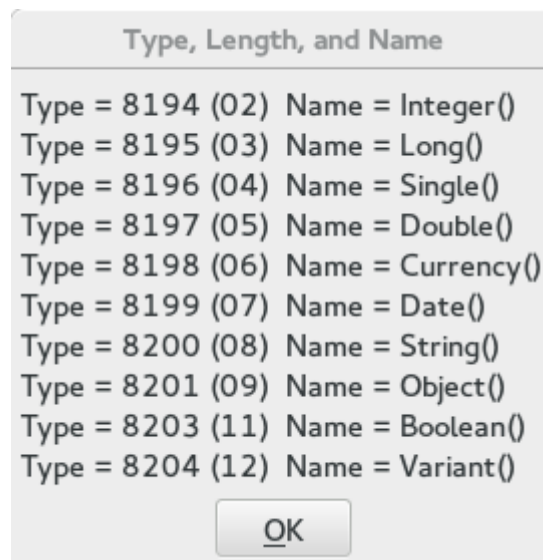


Figure 72. Types, lengths, and names for array variables.

9.8. Routines you should not use and other curiosities

I first heard my good friend Manfred say that “the book that is forbidden to read turns one reader into three.” Do not apply this logic to the routines outlined in this section. These routines are not officially documented; even if they work now, they may stop working in the near future.

Deprecated routines may still exist and they may work. They may also be removed at any time. Empty stubs exist, compile, and run, but do nothing. You may find older code that uses these legacy functions (see Table 86). I include CDec in the list because it is only listed on the Windows platform, which is just plain silly.

Table 86. *Deprecated and questionable routines.*

Routine	Comment
AboutStarBasic	Deprecated empty stub that used to be an information dialog.
SendKeys	Deprecated, generates an error.
DumpAllObjects(path, bool)	Internal debugging routine; no typical use known.
Load(Object)	Deprecated.
Unload(Object)	Deprecated.
LoadPicture(path)	Deprecated. Loads an image file.
SavePicture(obj, path)	Deprecated. Fails to save an image file.
CreatePropertySet(object)	Deprecated. Early UNO support function.
CDec(expression)	Generate a Decimal type; only implemented on Windows.
GetGUIVersion()	Deprecated, returns -1.

DumpAllObjects is an internal debugging routine that accepts two arguments. The first argument is a file name that will contain the text output. The second argument is a Boolean value that determines if each object should be fully loaded before it is dumped. For performance reasons, some properties are not created until they are accessed for the first time. This is true, even for BASIC objects that wrap UNO objects.

```
DumpAllObjects("C:\foo.txt", true) 'Fully load all objects before dumping
```

The LoadPicture and SavePicture routines refer to the old BASIC dialogs that are not implemented using UNO. The object returned by LoadPicture was used to set a picture on an image control. There is now an image control that uses the UNO API to set the picture.

```
Dim v
v = LoadPicture("C:\test1.jpg") 'This seems to load the picture
SavePicture(v, "C:\test2.jpg") 'This writes a zero length file
```

The CreatePropertySet function accepts an object as an argument and returns a property set. The property set is empty and not very useful. This function was created when the UNO functionality was first introduced, and will be removed from the source code in the future. If you find code that uses this function, I recommend that you change it.

```
v = CreatePropertySet(ThisComponent.Text)
```

9.9. Routines I do not understand

I find functions by reading the source code, some I understand and some I do not. This section contains the functions that I have not been able to satisfactorily document.

Table 87. Functions I do not fully understand.

Function	Description
EnableReschedule(bool)	Looks like it lets basic yield, or not yield, to another thread or process.
Me	Visual Basic thing, must use Compatibility Mode.
RTL	Visual Basic thing, must use Compatibility Mode.
GetDialogZoomFactorX()	The factor by which a dialog is scaled in the X direction. Used while determining how to scale an image for preview.
GetDialogZoomFactorY()	The factor by which a dialog is scaled in the Y direction.

EnableReschedule accepts a boolean argument. I have found neither usage nor documentation for this method, so, I have only speculation as to its purpose. I speculate that OOo sometimes reschedules things, perhaps events, and that sometimes rescheduling should not be allowed; say during a callback. Use EnableReschedule to enable, or disable, the internal reschedule flag. It is problematic that there is no method to determine the current state, so you cannot set it and then restore it. It appears, however, that the default state is enabled.

Me is available in the .Net world to reference the class or structure in which the code is executing. Me is only available in compatibility mode and must be used in the correct context or an error occurs.

GetDialogZoomFactorX and GetDialogZoomFactorY appear to be used to determine the scale factor to use so that a preview image will be properly displayed. In the very few examples that I have found, I always see code similar to the following:

```
widthToUse = GetDialogZoomFactorX(imageWidth) * imageWidth
heightToUse = GetDialogZoomFactorY(imageHeight) * imageHeight
```

9.10. New Compatibility Functions

TODO: ??? Come back to these and integrate them.

I found functions that I had not previously documented. All of these require that compatibility mode be set to true. In other words, they were added for Visual Basic compatibility. I did not look at this in depth.

Table 88. New VBA compatibility functions

Function	Comment	Parameters
DDB	Asset Depreciation	Cost as double, Salvage as double, Life as double, Period as double, Factor as variant)
FV	Future Value	Rate as double, NPer as double, Pmt as double, PV as variant, Due as Variant
IRR	Internal Rate of Return	ValueArray as array, Guess as variant
MIRR	Modified IRR	ValueArray as array, FinanceRate as double, ReinvestRate as double
Nper	Number of Periods	Rate as double, Pmt as double, PV as double, FV as variant, Due as variant

Function	Comment	Parameters
NPV	Net Present Value	Rate as double, ValueArray as array
Pmt	Loan Payment	Rate as double, NPer as double, PV as double, FV as variant, Due as Variant
Ppmt	Payment on Principal	Rate as double, Per as double, NPer as double, PV as double, FV as variant, Due as variant
PV	Present Value	Rate as double, NPer as double, Pmt as double, FV as variant, Due as variant
Rate	Periodic Interest Rate	NPer as double, Pmt as double, PV as double, FV as variant, Due as variant, Guess as variant
Round	Round	Expression as double, Numdecimalplaces as integer
SLN	Straight Line Deprecation	Cost as double, Double as double, Life as double
SYD	Asset Deprecation	Cost as double, Salvage as double, Life as double, Period as double

9.11. Conclusion

This chapter contains a lot of subtle information. If you are using Basic for the first time, consider reading the chapter again after you have more experience. The routines for inspecting variables are useful while evaluating returned objects. Knowledge of twips is required while trying to determine spacing and image sizes. Be certain to watch for deprecated routines so that can avoid using them.

10. Universal Network Objects

The internals of Libreoffice are based on Universal Network Objects (UNO). This chapter introduces the subroutines and functions supported by LO Basic that are related to UNO. This chapter covers methods that create and inspect objects that are vital to the internals of LO.

Up to this point, I have mostly dealt with simple single value things such as string and integer. An object, however, usually contains many values and it also has methods; for example, you can access a document as a variable and access information about the document (data or properties), and call methods that manipulate the document.

In this chapter, I begin to discuss things related to the actual implementation of LO — things that allow you to exploit the internal capabilities of LO. You'll also begin to see more details about how LO is implemented — which you need to build the really cool stuff.

Are you a programmer or a highly technical person? If not, skip this paragraph and go directly to the tip. Still reading? Great, so UNO is:

- The interface based component model for OOo.
- Enables interoperability between programming languages, object models and hardware architectures, either in process or over process boundaries, as well as in the intranet or the Internet.
- New languages are supported by adding a language binding. This is also stated as adding a “bridge” or “adapter”. This paradigm makes it easier to support multiple languages.
- UNO components may be implemented in, and accessed from, any programming language with a complete language binding.

TIP

You can write powerful macros without fully understand Universal Network Objects. Just think of a UNO as any object that is used internally by LO.

Stated more simply: The internal parts of LO are used as Universal Network Objects. By using UNOs, it is possible to access an instance of LO running on a different computer and operating system. A vague understanding of Universal Network Objects is important because most of the internals of LibreOffice are implemented using UNO.

Table 89 lists the Basic functions used to deal with UNO.

Table 89. Functions related to Universal Network Objects in Basic.

Function	Description
BasicLibraries	Access Basic libraries stored in a document.
CreateObject(obj_type)	Able to create any standard type, more flexible than CreateUnoStruct and CreateUnoService.
CreateUnoDialog()	Create an existing dialog.
CreateUnoListener()	Create a listener.
CreateUnoService()	Create a Universal Network Object Service.
CreateUnoStruct()	Create a Universal Network Object Structure.
CreateUnoValue()	Create a Universal Network Object value.
DialogLibraries	Access dialog libraries stored in a document.
EqualUNOObjects()	Determine if two UNO objects reference the same instance.
FindObject()	Find a named object; do not use.
FindPropertyObject()	Find object property based on name; do not use.
GetDefaultContext()	Get a copy of the default context.
GetProcessServiceManager()	Get service manager.
GlobalScope	Application-level libraries.
HasUnoInterfaces()	Does an object support specified interfaces?
IsUnoStruct()	Is this variable a Universal Network Object?
StarDesktop	Special variable representing the desktop object.
ThisComponent	Special variable representing the current document.

10.1. Base types and structures

Most of the internal data used by OOO is based on standard types such as strings and numbers. These types are combined into structures (called structs), which act like user-defined data types. The structures are combined to form more complex UNO objects. A struct supports properties but not methods.

A structure provides a method of placing more than one value into a single variable. You access the values in a structure based on a name that is decided when the struct type is designed by a programmer. A commonly used struct is the PropertyValue, whose primary purpose is to hold a string name and a variant value. The following listing demonstrates creating a PropertyValue structure and then setting the name and the value. The contained properties are accessed by placing a period between the variable and the property name.

Listing 203. Use Dim As New to create a UNO structure.

```
Dim aProp As New com.sun.star.beans.PropertyValue
aProp.Name = "FirstName"      'Set the Name property
aProp.Value = "Paterton"      'Set the Value property
```

TIP

OOo objects have long names such as `com.sun.star.beans.PropertyValue`; in the text I usually abbreviate the name and simply write `PropertyValue`, but your macros must use the full name.

You must create (or obtain) a UNO structure before you can use it. The most common method to create a UNO structure is to use `Dim As New` (see Listing 203). You can also use `Dim` to create an array of structures.

Listing 204. *Use `Dim As New` to create an array of UNO structures.*

```
Dim aProp(4) As New com.sun.star.beans.PropertyValue
aProp(0).Name = "FirstName" 'Set the Name property
aProp(0).Value = "Clyde" 'Set the Value property
```

Use the `CreateUnoStruct` function to create a UNO structure when it is needed rather than declare it ahead of time. Dynamically creating a UNO structure allows the name of the structure to be provided at run time rather than compile time. Providing a name at run time is shown in Listing 205 and Listing 208. Providing a name at compile time is shown in Listing 203.

Listing 205. *Use `CreateUnoStruct` to create a UNO structure.*

```
Dim aProp
aProp = CreateUnoStruct("com.sun.star.beans.PropertyValue")
aProp.Name = "FirstName" 'Set the Name property
aProp.Value = "Andrew" 'Set the Value property
```

The `With` statement simplifies the process of setting the properties of a structure.

Listing 206. *Use `With` to simplify setting properties on structures.*

```
Dim aProp(4) As New com.sun.star.beans.PropertyValue
With aProp(0)
    .Name = "FirstName" 'Set the Name property
    .Value = "Paterton" 'Set the Value property
End With
```

The function `CreateUnoStruct` used to be the only method to create a UNO structure. It is used less since the introduction of the “`Dim As New`” syntax. The `CreateObject` function is a more general function than `CreateUnoStruct`. It is able to create instances of all types supported by the Basic internal factory mechanism. This includes user-defined types.

Listing 207. *Create a user-defined type with `CreateObject` or `Dim As`.*

```
Type PersonType
    FirstName As String
    LastName As String
End Type

Sub ExampleCreateNewType
    Dim Person As PersonType
    Person.FirstName = "Andrew"
    Person.LastName = "Pitonyak"
    PrintPerson(Person)
    Dim Me As Object
    Me = CreateObject("PersonType")
    Me.FirstName = "Andy"
```

```

    Me.LastName = "Pitonyak"
    PrintPerson(Me)
End Sub

Sub PrintPerson(x)
    Print "Person = " & x.FirstName & " " & x.LastName
End Sub

```

TIP For a user-defined type, “Dim As New” and “Dim As” both work. For a UNO struct, however, you must use “Dim As New”.

The CreateObject function accepts the same arguments as CreateUnoStruct, but it works for all supported types, whereas CreateUnoStruct works only for UNO structures. Therefore, there is no reason to use CreateUnoStruct rather than CreateObject.

***Listing 208.** Use CreateObject to create a UNO structure.*

```

Dim aProp
aProp = CreateObject("com.sun.star.beans.PropertyValue")
aProp.Name = "FirstName"      'Set the Name property
aProp.Value = "Paterton"      'Set the Value property

```

TIP CreateObject offers more flexibility than CreateUnoStruct to dynamically create objects based on a name.

I wrote a small test program that created 20000 structs. CreateUnoStruct and CreateObject took about the same amount of time. Using Dim As New, however, used 500 clock ticks less, which is useful if you want to make a macro run as fast as possible.

The TypeName function indicates that a UNO structure is an object. Use the IsUnoStruct function to determine if a variable is a UNO structure.

***Listing 209.** Use IsUnoStruct to check if an object is an UNO Struct.*

```

Dim aProp As New com.sun.star.beans.PropertyValue
Print TypeName(aProp)      'Object
Print IsUnoStruct(aProp)   'True

```

10.2. UNO interface

An interface defines how something interacts with its environment. A UNO interface resembles a group of subroutine and function declarations; arguments and return types are specified along with functionality.

You can use the interface to retrieve data from an object, set data in an object, or tell an object to do something. The interface indicates how an object can be used, but it says nothing about the implementation. For example, if an interface contains the method GetHeight that returns an integer, it’s natural to assume that the object contains an integer property named Height. It’s possible, however, that the height is derived or calculated from other properties. The interface does not specify how the height is obtained, just that it is available. A UNO structure, however, contains properties that are accessed directly; the internal representation is not hidden.

TIP

UNO interface names start with the capital letter X.

When you want to know what methods are supported by an object, check the interfaces.

UNO interface names start with the capital letter X, which makes it easy to recognize an interface; for example, the interface `com.sun.star.text.XTextRange` specifies a section of text with both a start and end position. Objects that support the `XTextRange` interface are also used to identify an object's position in text document. The start and end positions may be the same. The `XTextRange` interface defines the methods in Table 90.

Table 90. *Methods defined by the `com.sun.star.text.XTextRange` interface.*

Method	Description
<code>getText()</code>	Return the <code>com.sun.star.text.XText</code> interface that contains this <code>XTextRange</code> .
<code>getStart()</code>	Return a <code>com.sun.star.text.XTextRange</code> that references only the start position.
<code>getEnd()</code>	Return a <code>com.sun.star.text.XTextRange</code> that references only the end position.
<code>getString()</code>	Return a string that includes the text inside this text range.
<code>setString(str)</code>	Set the string for this text range, replacing existing text and clearing all styles.

TIPA UNO interface may be derived from another. Every UNO interface is required to be derived from `com.sun.star.uno.XInterface`.

A new UNO interface may be derived from another. This is not something that you do, but rather it's something that is done by the designer of the interface. The derived interface supports all of the methods defined in the interface from which it is derived. For example, the `com.sun.star.text.XTextCursor` extends the `XTextRange` interface to allow it to change the range, which makes sense if you think about what you do with a cursor. Any object that implements the `XTextCursor` interface, supports the methods in Table 90 and the new methods introduced by the `XTextCursor` interface.

The main points regarding interfaces (so far) are:

- 1) An interface defines methods. In other words, an interface defines what an object can do; including getting and setting internal properties.
- 2) An interface may be derived from another interface.
- 3) The last portion of an interface name begins with an X.

In UNO, objects are accessed by their interface. Many of the programming languages, such as Java and C++, force you to perform a little UNO magic and extract the correct interface before you can call the methods defined in the interface. Basic hides these details for you so that you can directly call methods and access properties directly.

TIPBasic hides many of the complicated details, so, for most things, it is easier to write a Basic program than to write Java script.

10.3. UNO service

A service abstractly defines an object by combining interfaces and properties to encapsulate some useful functionality. A UNO interface defines how an object interacts with the outside world; a UNO structure defines a collection of data; and a UNO service combines them together. Like a UNO interface, a UNO service does not specify the implementation. It only specifies how to interact with the object.

Almost every UNO object is defined by a service, so UNO objects are called services. Strictly speaking, however, “a service” is the object definition. The UNO object is the actual object created as defined by the service. A service may include multiple services and interfaces. An interface usually defines a single aspect of a service and therefore is usually smaller in scope.

Many services have a name similar to an interface name; for example, one of the interfaces exported by the TextCursor service is the XTextCursor interface. An interface or property may be declared as optional for a service. The XWordCursor interface is marked as optional for the TextCursor service. The implication is that not all text cursors support the word cursor interface. In practice, you learn which text cursors support which interfaces and then simply use them.

There are two typical methods for obtaining a service.

- Get a reference to an existing object; for example, retrieving the first text table in the current document.
- Ask a service factory to create an instance of an object; for example, if I want to insert a new text table into a document, I ask the document to give me a new empty table, which I then configure and insert into the document.

A service factory returns objects based on the name. The process service manager is the main object factory for OpenOffice.org. The factory is given the service name, and the factory decides what to return. A factory may return a brand new instance of an object, or, it may return an existing instance. Use `GetProcessServiceManager()` to obtain a reference to the process service manager. Use `CreateInstance` to create a service from the process service manager as shown in Listing 210.

Listing 210. *Use process service manager to create a service.*

```
Sub ManagerCreatesAService
    Dim vFileAccess
    Dim s As String
    Dim vManager
    vManager = GetProcessServiceManager()
    vFileAccess = vManager.CreateInstance("com.sun.star.ucb.SimpleFileAccess")
    s = vFileAccess.GetContentType("http://www.pitonyak.org/AndrewMacro.odt")
    Print s
End Sub
```

The code in Listing 210 obtains the process service manager, creates an instance of the SimpleFileAccess service, and then uses the created service. The `CreateUnoService` function is a

shortcut for creating a UNO service (see Listing 211). The purpose of Listing 211 is to demonstrate the `CreateUnoService` function, showing that it's simpler than creating a service manager. Listing 211 also demonstrates some useful functionality, using a dialog to choose a file.

Listing 211. Select a file on disk.

```
Function ChooseAFileName() As String
    Dim vFileDialog          'FilePicker service instance
    Dim vFileAccess          'SimpleFileAccess service instance
    Dim iAccept as Integer   'Response to the FilePicker
    Dim sInitPath as String  'Hold the initial path
    'Note: The following services MUST be called in the following order
    'or Basic will not remove the FileDialog Service
    vFileDialog = CreateUnoService("com.sun.star.ui.dialogs.FilePicker")
    vFileAccess = CreateUnoService("com.sun.star.ucb.SimpleFileAccess")
    'Set the initial path here!
    sInitPath = ConvertToUrl(CurDir)
    If vFileAccess.Exists(sInitPath) Then
        vFileDialog.SetDisplayDirectory(sInitPath)
    End If

    iAccept = vFileDialog.Execute()      'Run the file chooser dialog
    If iAccept = 1 Then                  'What was the return value?
        ChooseAFileName = vFileDialog.Files(0) 'Set file name if it was not canceled
    End If
    vFileDialog.Dispose()                'Dispose of the dialog
End Function
```

A directory can be chosen similarly.

Listing 212. Select a directory.

```
REM sInPath specifies the initial directory. If the initial directory
REM is not specified, then the user's default work directory is used.
REM The selected directory is returned as a URL.
Function ChooseADirectory(Optional sInPath$) As String
    Dim oDialog As Object
    Dim oSFA As Object
    Dim s As String
    Dim oPathSettings
    Dim oContext

    oDialog = CreateUnoService("com.sun.star.ui.dialogs.FolderPicker")
    'oDialog = CreateUnoService("com.sun.star.ui.dialogs.OfficeFolderPicker")
    oSFA = createUnoService("com.sun.star.ucb.SimpleFileAccess")

    If IsMissing(sInPath) Then
        'oPathSettings = CreateUnoService("com.sun.star.util.PathSettings")
        oContext = getProcessServiceManager().DefaultContext
        oPathSettings = oContext.GetValueByName("/singletons/com.sun.star.util.thePathSettings")
        oDialog.setDisplayDirectory(oPathSettings.Work)
    ElseIf oSFA.Exists(sInPath) Then
        oDialog.setDisplayDirectory(sInPath)
    Else
        s = "Directory '" & sInPath & "' Does not exist"
```

```

    If MsgBox(s, 33, "Error") = 2 Then Exit Function
End If

If oDialog.Execute() = 1 Then
    ChooseADirectory() = oDialog.getDirectory()
End If
End Function

```

TIP

The file picker and folder picker dialogs can use either the operating system supplied default dialogs, or OOO specific dialogs. The operating system specific dialogs provide less functionality; for example, you cannot set the initial displayed directory. Use **Tools > Options > LibreOffice > General** and place a check next to "Use LibreOffice dialogs".

The code in Listing 211 creates two UNO services by using the function `CreateUnoService`. There are times, however, when the service manager is required. For example, the service manager has methods to create a service with arguments, `CreateInstanceWithArguments`, and to obtain a list of all supported services, `getAvailableServiceNames()`. The code in Listing 213 obtains a list of the supported service names; there are 999 services on my computer using LO version 6; up from 562 with OOO version 1 and down a few from version 3.

Listing 213. *Service manager supports services.*

```

Sub HowManyServicesSupported
    Dim oDoc          ' Document created to hold the service names.
    Dim oText          ' Document text object.
    Dim oSD            ' SortDescriptor created for the document.
    Dim oCursor        ' Text cursor used for sorting.
    Dim i%             ' Index Variable.
    Dim sServices

    sServices = GetProcessServiceManager().getAvailableServiceNames()
    Print "Service manager supports ";UBound(sServices);" services"
    ' Create a Writer document.

    oDoc = StarDesktop.loadComponentFromURL( "private:factory/swriter", "_blank", 0, Array() )
    oText = oDoc.getText()

    ' Print the service names into a Writer document.
    For i = LBound( sServices ) To UBound( sServices )
        oText.insertString( oText.getEnd(), sServices(i), False )
        ' Do not insert a blank line at the end.
        oText.insertControlCharacter( oText.getEnd(), _
            com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, False )
    Next

    oCursor = oDoc.Text.CreateTextCursorByRange(oDoc.Text)
    oSD = oCursor.createSortDescriptor()
    oCursor.sort(oSD)

    oText.insertString( oText.getStart(), _
        "Service manager supports "&UBound(sServices)&" services", False )
End Sub

```

It is useful to look at the list of supported services. This provides insight into available functionality that you can explore. A great example of this is shown Listing 214. An object inserted into a document must be created by that document. Using the methods of Listing 213, you can check to see what types of objects the document can create.

Listing 214. What objects can a document create.

```
REM Print the object / service types that a document can create.
REM If the document (oDoc) is missing, the current document is used.
REM If nameFilter is included, then service names that contain
REM the string are printed; based on a case insensitive compare.
REM The service names are stored in a newly created Write document.
Sub TypesDocCanCreate(Optional oDoc, Optional nameFilter$)
    Dim allNames    ' List of all the names.
    Dim oWriteDoc    ' Newly created Write document to contain the names.
    Dim oText        ' Document text object.
    Dim s : s = "private:factory/swriter"

    ' Find out what this document can create.
    If IsMissing(oDoc) Then
        allNames = ThisComponent.getAvailableServiceNames()
    Else
        allNames = oDoc.getAvailableServiceNames()
    End If

    ' Create a new Write document to contain the list.
    oWriteDoc = StarDesktop.loadComponentFromURL(s, "_blank", 0, Array())
    oText = oWriteDoc.getText()
    If IsMissing(nameFilter) Then
        oText.insertString ( oText.End, Join(allNames, CHR$(13)), False )
    Else
        Dim i%
        For i = LBound(allNames) To UBound(allNames)
            If (InStr(allNames(i), nameFilter) > 0) Then
                ' Insert the text.
                oText.insertString ( oText.End, allNames(i), False )

                ' Insert a new paragraph.
                oText.InsertControlCharacter(oText.getEnd(), _
                    com.sun.star.text.ControlCharacter.APPEND_PARAGRAPH, False)
            End If
        Next
    End If
End Sub
```

10.3.1. Setting a Read-Only Value

Some values can be set only once, so changing the value requires creating a new object. As an example, consdier:

- A Calc sheet with combobox controls (com.sun.star.awt.UnoControlComboBoxModel).
- The controls are anchored to cells.

- Each control bound to a cell.
- Write a macro that finds each combobox and bind it to the cell in which it is anchored.

The obvious way to change the bound cell is to access and change the bound cell doing something like the following code that fails because ValueBinding is read-only:

```
' BoundCell is a struct, so copy the value out, change it and write it back.
oBoundCell = oControlDataModel.ValueBinding.BoundCell
oBoundCell.Column = 6
oBoundCell.Row = 6
oBoundCell.Sheet = 0
oControlDataModel.ValueBinding.BoundCell = oBoundCell
```

If you create a new value binding object and try to set the BoundCell property, this still fails. The value binding object must be initialized with the bound cell value, and it cannot be changed again:

```
Dim arg(0) as new com.sun.star.beans.NamedValue 'not a PropertyValue, a Named Value.
Dim oCellAddress As new com.sun.star.table.CellAddress
oCellAddress.Column = 6
oCellAddress.Row=13
oCellAddress.Sheet=0
arg(0).Name = "BoundCell"
arg(0).Value = oCellAddress
oService = oDoc.CreateInstanceWithArguments("com.sun.star.table.CellValueBinding", arg)
oControlDataModel.ValueBinding = oService
```

Using CreateInstanceWithArguments allows the service to be created and values be set when the object is initialized. You can accomplish the same thing in two steps (instead of one) by creating the instance without arguments and calling the initialize method after creating the object.

10.4. Context

LibreOffice has “things” that are available by name. For example, the type description manager and the service manager. A context is a collection of name-value pairs that you can use to get these “things” by name. LO maintains a default context that is available with the function GetDefaultContext. Although a context is required when a service is created, Basic automatically uses the default context when calling CreateUnoService; yet another reason why Basic is easier to use than other languages.

Listing 215. View context element names.

```
MsgBox Join(GetDefaultContext().getElementNames(), CHR$(10))
```

The example in Listing 165 uses the default context.

10.5. Inspecting Universal Network Objects

While writing a Basic macro, I don’t always understand the values returned from internal LO functions — for example, inspecting the value returned by GetDefaultContext. I write test code to examine the return values so that I can make appropriate decisions. I then frequently add more

test code to further inspect the returned object. The initial inspection uses routines that you probably know (see Table 91).

Table 91. *Initial inspection routines.*

Routine	Comment
IsMissing(obj)	Use this for optional arguments to see if they are missing.
IsNull(obj)	You cannot inspect a null object, but you know that it is null.
IsEmpty(obj)	You cannot inspect an empty object, but you know that it is empty.
IsArray(obj)	Use array-inspection methods to learn more about the array.
TypeName(obj)	Determine if this is a simple type such as a String or an Integer. If this is an Object, it is probably a UNO structure or a UNO service.
IsUnoStruct(obj)	Determine if this is a UNO structure.

The code in Listing 216 demonstrates an initial use of the functions in Table 91.

Listing 216. *Inspect the TextTables object on the current document.*

```
Dim v
v = ThisComponent.getTextTables()
Print IsObject(v)      'True
Print IsNull(v)        'False
Print IsEmpty(v)       'False
Print IsArray(v)       'False
Print IsUnoStruct(v)   'False
Print TypeName(v)      'Object
MsgBox v.dbg_methods  'This property is discussed later
```

If the returned object has a type name of Object and it isn't a UNO structure, it is probably a UNO service. Use the HasUnoInterfaces function to determine if a UNO object supports a set of interfaces. The first argument is the object to test. The arguments following the first are a list of interface names. If all of the listed interfaces are supported, the function returns True, otherwise it returns False. More than one interface may be checked at the same time.

```
HasUnoInterfaces(obj, interface1)
HasUnoInterfaces(obj, interface1[, interface2[, interface3[, ...]])
```

To distinguish between a UNO structure, an arbitrary object, and a UNO service, first check to see if the variable is an object. This is easily done using the TypeName function. If the TypeName contains the word Object, then you know it's an object of some sort. The next step is to see if the object is a UNO structure, by using the function IsUnoStruct. Finally, if the object supports an interface (any interface will do), you know it's a UNO service. Every interface is derived from com.sun.star.uno.XInterface, so it suffices to see if the object supports the XInterface interface. The code in Listing 217 uses the Basic variable ThisComponent, which represents the current document.

Listing 217. *Use HasUnoInterfaces and IsUnoStruct to determine the UNO type.*

```
Dim aProp As New com.sun.star.beans.PropertyValue
Print IsUnoStruct(aProp)                                     'True
```

```
Print HasUnoInterfaces(aProp, "com.sun.star.uno.XInterface") 'False
Print IsUnoStruct(ThisComponent) 'False
Print HasUnoInterfaces(ThisComponent, "com.sun.star.uno.XInterface") 'True
```

TIP

If the first argument to the function `HasUnoInterfaces` is not an object, a run-time error occurs. The object may be null, but it must be an object. If the argument is a variant, it must contain an object; it cannot be empty.

Most UNO services also support the `com.sun.star.lang.XServiceInfo` interface, which allows you to ask the object what services it supports (see Table 92).

Table 92. *XServiceInfo methods.*

Method	Description
<code>getImplementationName()</code>	Returns a string that uniquely identifies the implementation of the service. For example, <code>SwXTextDocument</code> is the name of a Writer text document.
<code>getSupportedServiceNames()</code>	Return a string array of services that the object supports.
<code>supportsService(serviceName)</code>	Return True if the object supports the service name.

LO supports different types of documents, such as Writer (word processing) and Calc (spreadsheet). Each document type supports at least one service that is supported only by that document type (see Table 93). You can determine the document type by checking if it supports one of these services. You can also use the method `getImplementationName()`, as shown in Table 92.

The code in Listing 218 demonstrates how to verify that a variable references a Writer document. If the argument `vDoc` does not support the `XServiceInfo` interface, however, a run-time error occurs because the `SupportsService` method is not implemented. Use appropriate error handling if this is required, as shown in Listing 219; this is used in Listing 227. If the argument is missing, the current document is used. If an error occurs or a recognized service is not supported, the text “unknown” is returned.

Table 93. *Unique service names based on document type.*

Document Type	Service
Drawing	<code>com.sun.star.drawing.DrawDocument</code>
Writer	<code>com.sun.star.text.TextDocument</code>
Writer HTML	<code>com.sun.star.text.WebDocument</code>
Calc	<code>com.sun.star.sheet.SpreadsheetDocument</code>
Math	<code>com.sun.star.formula.FormulaProperties</code>
Presentation	<code>com.sun.star.presentation.PresentationDocument</code>
Base	<code>com.sun.star.sdb.OfficeDatabaseDocument</code>
Base Table	<code>com.sun.star.sdb.DataSourceBrowser</code>
Basic IDE	<code>com.sun.star.script.BasicIDE</code>

Listing 218. *Verify that the document is a Writer document.*

```
Sub DoSomethingToWriteDocument(vDoc)
    If NOT vDoc.supportsService("com.sun.star.text.TextDocument") Then
        MsgBox "A Writer document is required", 48, "Error"
        Exit Sub
    End If
    REM rest of the subroutine starts here
End Sub
```

Listing 219. *Determine a document's type.*

```
Function getDocType(Optional vDoc) As String
    On Error GoTo Oops
    If IsMissing(vDoc) Then vDoc = ThisComponent
    If vDoc.SupportsService("com.sun.star.sheet.SpreadsheetDocument") Then
        getDocType = "calc"
    ElseIf vDoc.SupportsService("com.sun.star.text.TextDocument") Then
        getDocType = "writer"
    ElseIf vDoc.SupportsService("com.sun.star.drawing.DrawingDocument") Then
        getDocType = "draw"
    ElseIf vDoc.SupportsService(_
        "com.sun.star.presentation.PresentationDocuments") Then
        getDocType = "presentation"
    ElseIf vDoc.SupportsService("com.sun.star.formula.FormulaProperties") Then
        getDocType = "math"
    ElseIf vDoc.SupportsService("com.sun.star.sdb.OfficeDatabaseDocument") Then
        getDocType = "base"
    Else
        getDocType = "unknown"
    End If
Oops:
    If Err <> 0 Then getDocType = "unknown"
    On Error GoTo 0 'Turn off error handling AFTER checking for an error
End Function
```

Some document types cannot be easily distinguished; for example, an XForms document and a Master document both support the same services as a regular text document. All of the Write based documents support the service `com.sun.star.text.GenericTextDocument`, and all of the office type documents support the service `com.sun.star.document.OfficeDocument`.

When I want to know about an interface or service, the first place that I look is online for the official documentation. Unfortunately, only a partial picture is easily produced (for the TextCursor service, see <http://www.openoffice.org/api/docs/common/ref/com/sun/star/text/TextCursor.html>). There are some issues:

- 1) The documentation does not present a unified picture, you must follow many links to determine what the object can do. All supported and optional interfaces are listed, but you must inspect each interface to know what each one does.
- 2) Each interface has its own page. If an interface is derived from another, you must look at another page to see the definition of the parent interface.

- 3) You don't know if the optional items are supported for a specific object instance without inspecting the object.

TIP Use Google to search for “site:http://www.openoffice.org/api/docs TextCursor” to quickly find the documentation for the TextCursor service.

TIP <http://www.openoffice.org/api/docs/common/ref/com/sun/star/text/module-ix.html> lists all of the services and interfaces at the com.sun.star.text level.

Most UNO services contain the properties `dbg_properties`, `dbg_methods`, and `dbg_supportedInterfaces`. Each of these properties is a string that contains a list of the supported properties, methods, or supported interfaces. Each string starts with text similar to “Properties of Object “ThisComponent”:”. The individual items are separated by the delimiter as shown in Table 94. Sometimes there are extra spaces following the delimiter and sometimes there are extra new-line characters — `CHR$(10)`.

Table 94. UNO “dbg_” properties.

Property	Delimiter	Description
<code>dbg_properties</code>	“,”	All properties supported by the object.
<code>dbg_methods</code>	“,”	All methods supported by the object.
<code>dbg_supportedInterfaces</code>	<code>Chr\$(10)</code>	All interfaces supported by the object.

The code in Listing 220 provides an easy method to see what an object supports. Sometimes, however, too many items are displayed and portions of the dialog do not fit on the screen. To avoid this problem, the code in Listing 221 splits the list into smaller, easily managed chunks. Figure 73 is only one of the many dialogs that are displayed by the macro in Listing 221. Be warned, the dialogs will not fit on the screen, so, just press enter to close each dialog.

Listing 220. The “dbg_” properties return useful information.

```
MsgBox vObj.dbg_properties
MsgBox vObj.dbg_methods
MsgBox vObj.dbg_supportedInterfaces
```

Listing 221. Display information from one of the debug properties.

```
Sub ExampleDisplayDbgInfoStr()
    Dim vObj
    vObj = ThisComponent
    DisplayDbgInfoStr(vObj.dbg_properties, ";", 40, "Properties")
    DisplayDbgInfoStr(vObj.dbg_methods, ";", 40, "Methods")
    DisplayDbgInfoStr(vObj.dbg_supportedInterfaces, CHR$(10), 40, "Interfaces")
End Sub

Sub DisplayDbgInfoStr(sInfo$, sSep$, nChunks, sHeading$)
    Dim aInfo() 'Array to hold each string
    Dim i As Integer 'Index variable
    Dim j As Integer 'Junk integer variable for temporary values
```



```

Dim s As String                                'holds the portion that is not completed
s = sInfo$
j = InStr(s, ":")                              'Initial colon
If j > 0 Then Mid(s, 1, j, "") 'Remove portion up to the initial colon
Do
    aInfo() = Split(s, sSep$, nChunks)          'Split string based on delimiter.
    s = aInfo(UBound(aInfo()))                  'Grab the last piece. Contains
    If InStr(s, sSep$) < 1 Then                  'the rest if there were too many.
        s = ""                                  'If there were not, then clear s.
    Else                                         'If there was a delimiter then
        ReDim Preserve aInfo(nChunks - 2)      'change array dimension to
    End If                                     'remove the extra last piece.
    For i = LBound(aInfo()) To UBound(aInfo()) 'Look at each piece to remove
        aInfo(i) = Trim(aInfo(i))              'leading and trailing spaces.
        j = InStr(aInfo(i), CHR$(10))          'Some have an extra
        If j > 0 Then Mid(aInfo(i), j, 1, "") 'new line that should be removed.
    Next
    MsgBox Join(aInfo(), CHR$(10)), 0, sHeading$
Loop Until Len(s) = 0
End Sub

```

When a type is included in one of the “dbg_” properties, it is preceded by the text “Sbx”, as shown in Figure 73. These names starting with Sbx correspond to the internal names used by Basic.

TIP	A complete object browser is presented later that demonstrates all of these concepts (see section 18.5 The object inspector example).

TIP	A popular object browser, called Xray described at the following link: http://wiki.openoffice.org/wiki/Extensions_development_basic

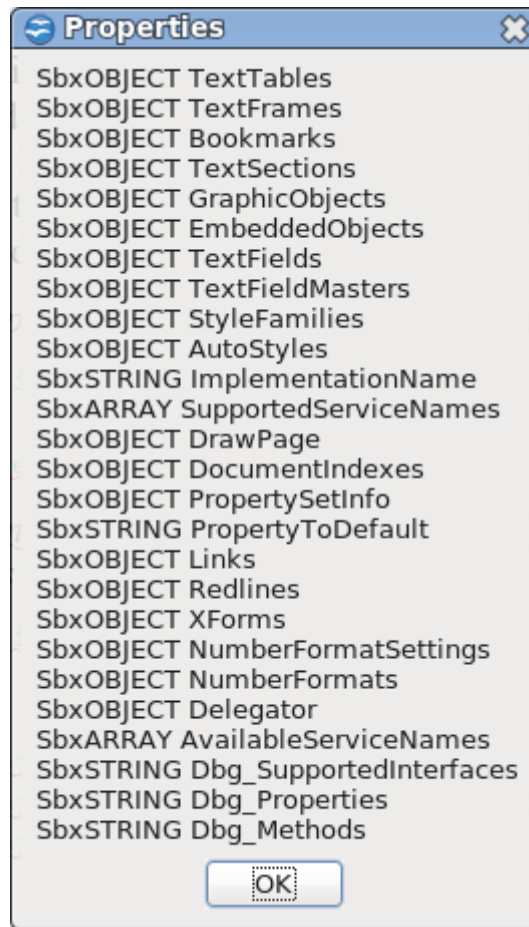


Figure 73. A few properties in a Writer document; one dialog of many displayed.

10.6. Using the type description manager

The TypeDescriptionManager manages type descriptions and acts as a central access point for every type description. This object is available from the default context as a singleton. Creating the object directly does not provide a usable object

```
CreateUnoService("com.sun.star.reflection.TypeDescriptionManager")
```

I created a usable object using the accepted method (shown shortly), and even when using that type directly, it too failed.

```
CreateUnoService("com.sun.star.comp.stoc.TypeDescriptionManager")
```

Get the type description manager from the default context; see section 10.4 Context, which also shows how to enumerate all of the singleton objects available.

```
Function GetTypeDescriptionManager()
    Dim sTDMName$ ' Name of the type description manager.
    sTDMName = "/singletons/com.sun.star.reflection.theTypeDescriptionManager"
    GetTypeDescriptionManager() = GetDefaultContext().getValueByName(sTDMName)
End Function
```

The following method enumerates all “things” in the com.sun.star.awt module.

Listing 222. Enumerate types in a module.

```
Sub EnumerateTypesTest
    Dim oTDM          ' Type Description Manager.
    Dim oTDE          ' Type Description Enumerations.
    Dim oTD           ' One Type Description.
    Dim typeArray     ' Types for which descriptions are returned.
    Dim s$            ' Utility string variable.

    REM All supported types.
    typeArray = Array(com.sun.star.uno.TypeClass.VOID, _
        com.sun.star.uno.TypeClass.CHAR, _
        com.sun.star.uno.TypeClass.BOOLEAN, _
        com.sun.star.uno.TypeClass.BYTE, _
        com.sun.star.uno.TypeClass.SHORT, _
        com.sun.star.uno.TypeClass.UNSIGNED_SHORT, _
        com.sun.star.uno.TypeClass.LONG, _
        com.sun.star.uno.TypeClass.UNSIGNED_LONG, _
        com.sun.star.uno.TypeClass.HYPER, _
        com.sun.star.uno.TypeClass.UNSIGNED_HYPER, _
        com.sun.star.uno.TypeClass.FLOAT, _
        com.sun.star.uno.TypeClass.DOUBLE, _
        com.sun.star.uno.TypeClass.STRING, _
        com.sun.star.uno.TypeClass.TYPE, _
        com.sun.star.uno.TypeClass.ANY, _
        com.sun.star.uno.TypeClass.ENUM, _
        com.sun.star.uno.TypeClass.TYPDEF, _
        com.sun.star.uno.TypeClass.STRUCT, _
        com.sun.star.uno.TypeClass.UNION, _
        com.sun.star.uno.TypeClass.EXCEPTION, _
        com.sun.star.uno.TypeClass.SEQUENCE, _
        com.sun.star.uno.TypeClass.ARRAY, _
        com.sun.star.uno.TypeClass.INTERFACE, _
        com.sun.star.uno.TypeClass.SERVICE, _
        com.sun.star.uno.TypeClass.MODULE, _
        com.sun.star.uno.TypeClass.INTERFACE_METHOD, _
        com.sun.star.uno.TypeClass.INTERFACE_ATTRIBUTE, _
        com.sun.star.uno.TypeClass.UNKNOWN, _
        com.sun.star.uno.TypeClass.PROPERTY, _
        com.sun.star.uno.TypeClass.CONSTANT, _
        com.sun.star.uno.TypeClass.CONSTANTS, _
        com.sun.star.uno.TypeClass.SINGLETON)

    oTDM = GetTypeDescriptionManager()

    Dim sBaseName$ : sBaseName = "com.sun.star.awt"

    ' Change com.sun.star.reflection.TypeDescriptionSearchDepth.ONE
    ' to com.sun.star.reflection.TypeDescriptionSearchDepth.INFINITE
    ' to traverse more than a single level.
    oTDE = oTDM.createTypeDescriptionEnumeration(sBaseName, _
        typeArray, _
        com.sun.star.reflection.TypeDescriptionSearchDepth.ONE)
```

```

While oTDE.hasMoreElements()
    oTD = oTDE.nextTypeDescription()
    s$ = s & oTD.Name & CHR$(10)
Wend
MsgBox s
End Sub

```

To get the information on a specific fully qualified type, use the following macro (adapted from an example by Bernard Marcelly):

Listing 223. Get object description from a fully qualified name.

```

Function GetOOoConst(constString)
    Dim sTDMName$
    Dim oTDM

    sTDMName = "/singletons/com.sun.star.reflection.theTypeDescriptionManager"
    oTDM = GetDefaultContext().getValueByName(sTDMName)

    If oTDM.hasByHierarchicalName(constString) Then
        GetOOoConst = oTDM.getByHierarchicalName(constString)
    Else
        MsgBox "Unrecognized name : " & constString, 16, "OOo API constant or enum"
    End If
End Function

```

The method is usable to obtain constant and enumeration values from a text string

```
Print GetOOoConst("com.sun.star.awt.FontSlant.ITALIC")
```

This can also return an object that describes the type. This can be used to enumerate the values and the strings.

Listing 224. Inspect enumerations.

```

Sub EnumerateEnumerations(sName$)
    Dim oTD          ' One Type Description.
    Dim oTDE          ' Element enumeration
    Dim s$            ' Utility string variable.
    Dim sNames
    Dim lValues
    Dim i As Long
    Dim iCount As Integer

    oTD = GetOOoConst(sName)
    If IsNull(oTD) OR IsEmpty(oTD) Then
        Exit Sub
    End If

    If HasUnoInterfaces(oTD, "com.sun.star.reflection.XEnumTypeDescription") Then

        'MsgBox Join( oTD.getEnumNames(), CHR$(10))
        sNames = oTD.getEnumNames()
        lValues = otd.getEnumValues()
        For i = LBound(sNames) To UBound(sNames)

```

```

        iCount = iCount + 1
    If (iCount > 40) Then
        MsgBox(s)
        s = ""
    End If
    s = s & lValues(i) & CHR$(9) & sNames(i) & CHR$(10)
Next
ElseIf HasUnoInterfaces(oTD, "com.sun.star.reflection.XConstantsTypeDescription") Then
    lValues = oTD.getConstants()
    For i = LBound(lValues) To UBound(lValues)
        iCount = iCount + 1
        If (iCount > 40) Then
            MsgBox(s)
            s = ""
        End If
        s = s & lValues(i).getConstantValue() & CHR$(9) & lValues(i).getName() & CHR$(10)
    Next
Else
    'Inspect oTD
    MsgBox "Unsupported type " & sName
    Exit Sub
End If
MsgBox s
End Sub

```

This can be used to see enumerations.

```
EnumerateEnumerations("com.sun.star.awt.FontSlant")
```

This can be used to see constant groups.

```
EnumerateEnumerations("com.sun.star.awt.FontWeight")
```

10.7. Use Object or Variant

It is almost always safe to use Object variables to hold UNO services. It is so safe, apparently, that even the macro recorder uses Object variables to hold UNO services. Unfortunately, it is not always safe. The OOo Developer's Guide specifically states that Variant variables should be used instead of Object variables. Always use the type Variant to declare variables for UNO services, not the type Object. The Basic type Object is tailored for pure Basic objects — in other words, objects that can be created with the Dim As New syntax. The Variant variables are best for UNO services to avoid problems that can result from the Basic specific behavior of the type Object. I asked Andreas Bregas (one of the primary developers for the Basic infrastructure) about this, and he said that in most cases, both work. The OOo Developer's Guide prefers Variant because there are some odd situations in which the usage of type Object leads to an error due to the old Basic Object type semantics. But if a Basic program uses type Object, it will almost always run correctly and there should be no problems. It is sufficiently rare, however, that he could not remember an example where it was a problem. On the other hand, with OOo version 1, I experienced a problem that corrected itself, when I changed the variable type from Variant to Object.

10.8. Comparing UNO variables

Use the `EqualUnoObjects` function to determine if two UNO objects are the same object. UNO structures are copied by value but UNO services are copied by reference. This means that `EqualUnoObjects` should always return `False` for two variables that contain a structure, and it might return `True` for UNO services.

```
Dim vObj
vObj = ThisComponent
Print EqualUnoObjects(vObj, ThisComponent)           'True

Dim aProp As New com.sun.star.beans.PropertyValue
vObj = aProp
Print EqualUnoObjects(vObj, aProp)                   'False
```

After using LO for many years, I finally found a use for `EqualUnoObjects`. Be warned that this is an advance topic and makes use of constructs not widely known. The problem as posed is: “When a button is pressed in a Calc document, I want to know which cell contains the button.” Here are some bits of information to help understand the problem:

- Each document has at least one “draw page” that contains graphical things such as images embedded in the document.
- You can embed controls such as a button in a document. When the button is pressed, and a macro is called, it is trivial to get a reference to the control “data model”.
- Each control has a shape embedded in the draw page.
- Each control also has a “data model” that contains properties and other similar things.
- From a control data model I cannot easily obtain the shape or the cell containing the control.
- From a shape in a Calc document, I can easily find the cell – if the control is anchored to a cell.

My solution was to find the shape associated with a control. It is easy to find the control name, but, a control name need not be unique. I opted to look at every shape on the draw page and then check to see if the control associated with that shape is the same object as the control in question.

Listing 225. Find the control shape for a specific control.

```
Function FindCellWithControl(oDrawPage, oControl)
    Dim oShape
    Dim oAnchor
    oShape = FindShapeForControl(oDrawPage, oControl)
    If Not IsEmpty(oShape) Then
        If oShape.getAnchor().supportsService("com.sun.star.sheet.SheetCell") Then
            FindCellWithControl = oShape.getAnchor()
        End If
    End If
End Function
```

```

Function FindShapeForControl(oDrawPage, oControl)
    Dim i
    Dim oShape
    For i = 0 To oDrawPage.getCount()
        oShape = oDrawPage.getByIndex(i)
        If oShape.supportsService("com.sun.star.drawing.ControlShape") Then
            If EqualUNOObjects(oControl, oShape.Control) Then
                FindShapeForControl = oShape
                Exit Function
            End If
        End If
    Next
End Function

```

This is part of an event handler called by a button that finds the cell containing the button.

```

Sub ButtonHandler(oEvent)
    Dim oControlModel
    Dim oParent
    Dim oCell

    'Print oEvent.Source.Model.getName()
    oControlModel = oEvent.Source.Model

    oParent = oControlModel.getParent()
    Do While NOT oParent.supportsService("com.sun.star.sheet.SpreadsheetDocument")
        oParent = oParent.getParent()
    Loop

    oCell = FindCellWithControl(oParent.getCurrentController().getActiveSheet().getDrawPage(),
oControlModel)
    If NOT IsEmpty(oCell) Then
        Print "Control is in cell " & oCell.AbsoluteName
    Else
        Print "Unable to find cell with control"
    End If
End Sub

```

10.9. Built-in global UNO variables

LO Basic contains built-in global variables that provide quick access to frequently used components in LO. The most commonly used variable is `ThisComponent`, which refers to the currently active document. To demonstrate using the variable `ThisComponent`, the macro in Listing 226 displays all of the styles in the current document. Figure 74 is only one of the many dialogs that are displayed while the macro runs. The fact that a style exists in a document does not imply that the style is actually used in the document.

Listing 226. Display all styles known by this document.

```

Sub DisplayAllStyles
    Dim vFamilies As Variant 'All the style types
    Dim vFamNames As Variant 'Array with names of the style types
    Dim vStyles As Variant   'One style type such as Number or Page Styles

```

```

Dim vStlNames As Variant 'Array with names of specific styles
Dim s As String          'Message to display
Dim n As Integer         'Iterate through the style types
Dim i As Integer         'Iterate through the styles

vFamilies = ThisComponent.StyleFamilies 'Get the styles
vFamNames = vFamilies.getElementNames() 'What type of styles?
For n = LBound(vFamNames) To UBound(vFamNames) 'Look at all style types
    s = ""
    vStyles = vFamilies.getByNames(vFamNames(n)) 'Get styles of a type
    vStlNames = vStyles.getElementNames() 'names of styles for type
    For i = LBound(vStlNames) To UBound(vStlNames) 'for all styles of a type
        s = s & i & " : " & vStlNames(i) & Chr$(13) 'Build a display string
        If ((i + 1) Mod 35 = 0) Then 'Display 35 at a time
            MsgBox s, 0, vFamNames(n) 'Display them
            s = "" 'Clear string, start over
        End If
    Next i 'Next style
    If Len(s) > 0 Then MsgBox s, 0, vFamNames(n) 'Leftover styles for type
Next n 'Next style type
End Sub

```

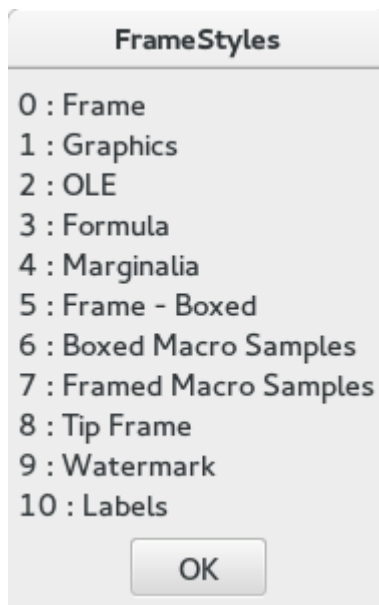


Figure 74. Frame styles known by the current document.

LO is based on the same code as StarOffice, which had a desktop. All of the individual windows were contained in this desktop. The desktop paradigm is now gone, but for legacy reasons a desktop object still acts as the global application that ties all of the documents together. Although I frequently see code that creates a desktop service by using the function `CreateUnoService`, this is not required in Basic. Basic provides the variable `StarDesktop`, which accesses the primary desktop service in LO. The macro in Listing 227 demonstrates the use of the `StarDesktop` by traversing all of the currently open documents. The `getDocType` method is defined in Listing 219.

Listing 227. Inspect each open component.

```
Sub IterateThroughAllDocs
    On Error Resume Next      'Ignore the non-document components
    Dim vComponents As Object 'All of the components
    Dim vDocs As Object       'Enumeration of the documents
    Dim vDoc As Object        'A single document
    Dim s As String
    GlobalScope.BasicLibraries.LoadLibrary("Tools") 'Contains FileNameOutOfPath
    vComponents = StarDesktop.getComponents()        'Get all the components
    vDocs = vComponents.createEnumeration()          'Enumerate them
    Do While vDocs.hasMoreElements()                'While there are more
        vDoc = vDocs.nextElement()                  'Get the next component
        s = s & getDocType(vDoc) & "  "              'Add the document type
        s = s & FileNameOutOfPath(vDoc.getURL())     'Add the file name
        s = s & CHR$(10)                             'Add a new line
    Loop
    MsgBox s, 0, "Currently Open Components"
End Sub
```

TIP The primary visible windows in LO are called “components.” Every open document is a component, as is the Basic IDE and the Help window. In LO, the word “component” almost always means an open document.

While iterating through the open documents (components), you may find some unexpected documents. These are component windows such as the Basic IDE and the Help window. The macro in Listing 227 uses the function `FileNameOutOfPath`. This is another macro and is not a function that is built into Basic. This function is stored in the Strings module of the application-level Tools library. If a library is not currently loaded, you cannot call the methods that it contains.

The `GlobalScope` variable references the application-level libraries and is used to load the Tools library. Loading a library loads all of the modules in the specified library. OOo contains libraries and modules that are not built into Basic. Use the `LoadLibrary` method before you use the routines in the libraries.

```
GlobalScope.BasicLibraries.LoadLibrary("Tools")
```

To access the Basic libraries in the current document, either use the `BasicLibraries` global variable or access the `BasicLibraries` property in the current document.

```
Print EqualUnoObjects(vObj.BasicLibraries, BasicLibraries) 'True
```

Use the `DialogLibraries` variable to access the dialog libraries in the current document. Unlike `BasicLibraries`, an individual document does not contain a property called `DialogLibraries` to directly obtain the dialog libraries for a specific document. You can easily obtain the dialog and Basic libraries for a specific document through a less direct route. Each document has a `LibraryContainer` property.

```
ThisComponent.LibraryContainer.getByName("OOME_40").getModuleContainer()
ThisComponent.LibraryContainer.getByName("OOME_40").getDialogContainer()
```

The `getByName()` method on the `LibraryContainer` returns the named library. The `getModuleContainer()` method returns the Basic container for the specified library, and the `getDialogContainer()` method returns the Dialog container for the specified library. The code in Listing 228, however, uses the variables `DialogLibraries` and `BasicLibraries` to list the number of dialogs and modules in each library in the current document. Figure 75 shows the results.

Listing 228. *View the libraries and dialogs stored in the current document.*

```
Sub ExamineDocumentLibraries
    Dim vLibs          'Hold the library names
    Dim vMod           'Hold the modules/dialogs object
    Dim nNumMods%      'Number of modules or dialogs in a library
    Dim i%            'Scrap index variable
    Dim s$            'Scrap string variable
    s = "*** Dialog Libs In Document" & CHR$(10)      'Initialize s
    vLibs = DialogLibraries.getElementNames()          'Library names
    For i = LBound(vLibs) To UBound(vLibs)             'Look at each name
        vMod = DialogLibraries.getByName(vLibs(i))     'Get the dialog library
        nNumMods = UBound(vMod.getElementNames()) + 1  'How many dialogs
        s = s & vLibs(i) & " has " & nNumMods & " dialogs" 'Build string
        s = s & CHR$(10)
    Next i
    s = s & CHR$(10)
    s = s & "*** Basic Libs In Document" & CHR$(10)   'Ready for code libs
    vLibs = BasicLibraries.getElementNames()           'Library names
    For i = LBound(vLibs) To UBound(vLibs)             'Look at each name
        vMod = BasicLibraries.getByName(vLibs(i))     'Get the code library
        nNumMods = UBound(vMod.getElementNames()) + 1  'Number of modules
        s = s & vLibs(i) & " has " & nNumMods & " modules" 'Build the string
        s = s & CHR$(10)
    Next i
    MsgBox s, 0, "Libraries"
End Sub
```

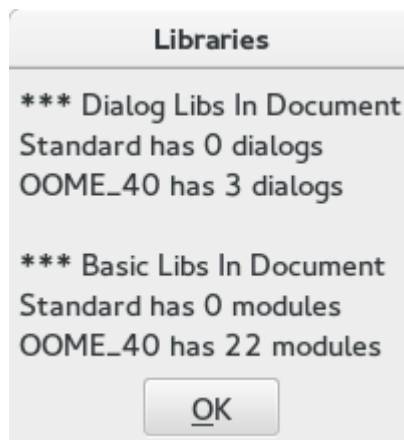


Figure 75. *Dialog and Basic libraries in the current document.*

To view the libraries in the application-level library container, modify the code in Listing 228. Add `GlobalScope` before each occurrence of `BasicLibraries` and `DialogLibraries`.

10.10. Creating UNO values for OOO internals

Basic does an excellent job of converting between the native Basic types and the types required internally by OOO. However, if you call a method on a Universal Network Object and Basic doesn't know what the type should be, it may not properly convert the type. For example, the `setProperty` method in the `XPropertySet` interface accepts two arguments — a string name of the property to set and the value to set. The type of the value to set depends on which property is set. Use the function `CreateUnoValue` (shown in Listing 229) to create a reference to a Universal Network Object that contains the appropriate type if there is a problem creating the correct type. I have never seen this happen, so don't fret that much about your arguments; you can almost always trust Basic to do the correct thing.

Listing 229. Use `CreateUnoValue` to create a reference to an internal UNO value.

```
Dim v
v = CreateUnoValue("unsigned long", 10)
v = CreateUnoValue("string", "hello")
v = CreateUnoValue("byte", 10)           'A byte is from 0 through 255
v = CreateUnoValue("[]byte", Array(3, 2, 1)) 'You can even create arrays
'v = CreateUnoValue("Byte", 10)          'No such element exception
'v = CreateUnoValue("byte", 1000)        'Out of range exception
'v = CreateUnoValue("uint64", 10)        'No such element exception
```

The first argument to `CreateUnoValue` is the data type that should be created. The supported data types are `void`, `char`, `boolean`, `byte`, `short`, `unsigned short`, `long`, `unsigned long`, `hyper`, `unsigned hyper`, `float`, `double`, `string`, `type`, and `any`. The names are case sensitive and may be preceded by square brackets to indicate an array. The value returned by `CreateUnoValue` is not usable by Basic; it is only useful to the internals of OOO. In other words, do not create a “byte” type and then expect to use it as a number.

Listing 230. Test types supported by `CreateUnoValue`.

```
Sub TestCreateUnoValues
    Dim typeNames
    Dim v
    Dim i%

    typeNames = Array("void", "boolean", "char", "byte", "string", _
        "short", "unsigned short", "long", "unsigned long", _
        "hyper", "unsigned hyper", "float", "double", "any")

    ' I took the list of names from
    ' cppu/source/typelib/typelib.cxx
    For i = LBound(typeNames) To UBound(typeNames)
        v = CreateUnoValue(typeNames(i), 65)
        ' You cannot directly use the value because it is an UNO type,
        ' which is only good for passing back into UNO.
        ' Things like "Print v" or "CStr(v)" will fail.
    Next
End Sub
```

10.11. Finding objects and properties

Basic has functions to find variables and their properties based on their names. This functionality was initially described to me as poorly documented, possibly deprecated, and buggy — and I cannot disagree. Use `FindObject` to obtain a reference to a variable with a given name, and `FindPropertyObject` to obtain a reference to a named property of an object. The macro in Listing 231 demonstrates some of the idiosyncrasies of the `FindObject` and `FindPropertyObject` functions.

TIP Do not use the functions `FindObject` and `FindPropertyObject` — they are poorly documented, buggy, and likely to be deprecated. I only mention the methods to be complete.

Listing 231. Test the FindObject method.

```
Sub TestFindObject
    Dim oTst
    Dim oDoc 'Initial value is EMPTY, whis ic not null.
    Dim oLib 'Gimmicks library.
    Dim oMod 'UserFields module in the Gimmicks library.
    Dim s$

    oTst = FindObject("oDoc")

    s = "oTst = FindObject(""oDoc""), " & CHR$(10) & _
        "IsNull(oTst) = " & IsNull(oTst) & _
        " and (oTst IS oDoc) = " & (oTst IS oDoc)

    Rem Test against the current document
    oDoc = ThisComponent
    oTst = FindObject("oDoc")

    s = s & CHR$(10) & CHR$(10) & _
        "oDoc = ThisComponent" & CHR$(10) & _
        "oTst = FindObject(""oDoc"*)" & CHR$(10) & _
        "oDoc IS ThisComponent = " & (oDoc IS ThisComponent) & CHR$(10) & _
        "oTst IS oDoc = " & (oTst IS oDoc) & CHR$(10) & _
        "oTst IS ThisComponent = " & (oTst IS ThisComponent) & _
        CHR$(10) & CHR$(10) & _
        "oTst = FindObject(""ThisComponent"*)" & CHR$(10)

    REM Do it again, but do search for ThisComponent
    oTst = FindObject("ThisComponent")

    s = s & "oTst IS oDoc" & (oTst IS oDoc) & CHR$(10) & _
        "oTst IS ThisComponent = " & (oTst IS ThisComponent) & CHR$(10) & _
        "oDoc IS ThisComponent = " & (oDoc IS ThisComponent) & CHR$(10)

    REM ThisComponent has a DocumentInfo property
    oTst = FindPropertyObject(ThisComponent, "DocumentInfo")

    s = s & "IsNull(FindPropertyObject(ThisComponent, " & _
        ""DocumentInfo"*)" & CHR$(10) & _
        "returns " & IsNull(oTst)
```

```

MsgBox s

REM Load the Gimmicks library
GlobalScope.BasicLibraries.LoadLibrary("Gimmicks")

REM Find the library even though it is not a variable
oLib = FindObject("Gimmicks")

REM userfields is a module in the Gimmicks library
oMod = FindPropertyObject(oLib, "Userfields")

s = "And now for something different." & CHR$(10) & _
    "Find the loaded Gimmicks library" & CHR$(10) & _
    "oMod Is Gimmicks.Userfields = " & (oMod Is Gimmicks.Userfields)

MsgBox s

'The StartChangesUserfields function is in the module Userfields
'Call the routine
Print "Call 'StartChangesUserfields' directly"
oLib.Userfields.StartChangesUserfields
Print "Call 'StartChangesUserfields' using oMod"
oMod.StartChangesUserfields
End Sub

```

If these functions serve your purposes, use them. I tried to use these to write an object inspector, but the FindPropertyObject function did not work as I expected.

10.12. UNO listeners

LO uses listeners to inform other objects what it is doing. Consider a hospital worker as an analogy: Before I (the hospital worker) do anything interesting to treat patient Paolo, I am required first to call his parents. For some things the parents may say “no, you cannot do that”, and for others the parents are only informed (status updates). This is very similar to UNO listeners. The code that you write is called the listener, and the thing to which you listen is called the broadcaster. In the previous example, the hospital worker is the broadcaster and Paolo’s parents are the listeners.

You can listen to the LO objects that act as a broadcaster. The routines required to listen to a specific OOO interface are specific to that interface. In other words, the set of subroutines and functions that can act as a listener to printing events is different than the set of subroutines and functions that can listen to keystroke events.

TIP If you do not implement a required method specified for the listener interface, the listener may crash LO.

10.12.1. Your first listener

This listener is presented with details that are explained in the next section, so, follow the easy process and then find out why. A print listener is demonstrated in section 13.15.3 A Calc example with a Print listener.

All listeners must implement the methods in the `com.sun.star.lang.XEventListener` interface, which defines a one subroutine; `disposing(EventObject)`. The disposing method is called when the broadcaster is about to become unavailable. In other words, the broadcaster will no longer exist and you should not use it. For example, a print job is finished so it is no longer required or a document is closing. Listing 232 demonstrates the most basic listener possible, `XEventListener`; Notice the string prefix “first_listen_” before the required name “disposing”; I chose “first_listen_” because it is descriptive.

Listing 232. *Simple listener that does nothing.*

```
Sub first_listen_disposing( vArgument )
    MsgBox "Disposing In First Listener"
End Sub
```

Listing 232 is all that is required for this simple listener. Use the `CreateUNOListener` to create a UNO listener that is associated with the macro in Listing 232. The first argument is the string prefix and the second argument is the name of the service to create.

The routine in Listing 233 creates the UNO listener and associates it with the routine in Listing 232. When the disposing method is called on `vListener`, it calls the routine in Listing 232.

TIP

The code in Listing 233 creates a listener and then calls the objects disposing method. The listener is implemented in Listing 232, so, this ultimately calls `first_listen_disposing` in Listing 232.

Listing 233. *Create a listener and then call the disposing method.*

```
Sub MyFirstListener
    Dim vListener
    Dim vEventObj As New com.sun.star.lang.EventObject
    Dim sPrefix$
    Dim sService$
    sPrefix = "first_listen_"
    sService = "com.sun.star.lang.XEventListener"
    vListener = CreateUnoListener(sPrefix, sService)
    vListener.disposing(vEventObj)
End Sub
```

10.12.2. A complete listener: selection change listener

The two most difficult parts of creating a listener are determining which broadcaster and which listener interface to use. The rest of the steps are easier.

- 1) Determine the broadcaster to use.
- 2) Determine the listener interface to implement.

- 3) Create global variables for the listener and maybe the broadcaster. Test your understanding and ask yourself why the variable must be global before reading the tip below.
- 4) Determine the prefix that you will use; make it descriptive.
- 5) Write all of the subroutines and functions that implement the listener.
- 6) Create the UNO listener and save it in the global variable.
- 7) Register the listener with the broadcaster.
- 8) When finished, remove the listener from the broadcaster.

TIP

Why store the listener in a global variable? There is usually a subroutine to create and register the listener with the broadcaster and then the macro ends. The listener must exist when the macro stops running. The only way to do this in Basic is to use a Global variable. Another macro, run later, removes the listener from the broadcaster. The latter macro must be able to access the listener.

Knowing which broadcaster to use is not always an easy task. It requires understanding how LO is designed; this comes with experience. There is usually an obvious place to start looking — the current document, available from the variable `ThisComponent`, contains the document data. For example, the document accepts an `XPrintJobListener` to monitor how printing is progressing. Most tasks related to user interactions — such as selecting text, moving the mouse, and entering keystrokes — all go through the document’s controller.

After selecting the broadcaster check the broadcaster for methods with names like “add listener”. I was asked how to prevent a dialog from closing when the user clicks on the little X icon in the upper-right corner of a dialog. I reasoned that the dialog is probably the correct broadcaster to use, so I created a dialog and inspected the “`dbg_methods`” property to see which listeners it supports (each supported listener has “set” and “remove” methods). Another option is to determine the event type that is produced and then search for broadcasters supporting this event type. I usually do this by searching the OOO Developer’s Guide and the Internet; I discuss effective methods for finding information in Chapter 19, Sources of Information.

To listen for a selection change event, implement an `XSelectionChangeListener` and register it with the current controller. The `XSelectionChangeListener` interface defines only one subroutine that must be implemented — `selectionChanged(ObjectEvent)`. All listeners are derived from the `XEventListener` so the `disposing(ObjectEvent)` method must also be implemented. The fastest way to determine which subroutines and functions must be implemented is to create an instance of the listener by using the function `CreateUNOListener` and then inspect the `dbg_methods` property (see Listing 234 and Figure 76).

Listing 234. *See the methods supported by a specific listener.*

```
Sub InspectListenerMethods
    Dim sPrefix$
    Dim sService$
    Dim vService
    sPrefix = "sel_change_"
```

```

sService = "com.sun.star.view.XSelectionChangeListener"
vService = CreateUnoListener(sPrefix, sService)
DisplayDbgInfoStr(vService.dbg_methods, ";", 35, "Methods")
End Sub

```

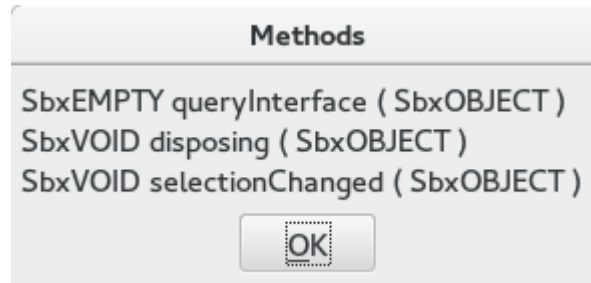


Figure 76. Ignore the queryInterface method.

The queryInterface method is from the interface com.sun.star.uno.XInterface, which is inherited from the XEventListener. You can ignore this method. However, you must implement the rest of the routines. Choose a descriptive prefix so that the code can contain more than one listener. See Listing 235.

Listing 235. *Disposing and selection changed.*

```

'All listeners must support this event
Sub sel_change_disposing(vEvent)
    msgbox "disposing the selection change listener"
End Sub

Sub sel_change_selectionChanged(vEvent)
    Dim vCurrentSelection As Object
    vCurrentSelection = vEvent.source
    Print "Number of selected areas = " & _
        vCurrentSelection.getSelection().getCount()
End Sub

```

The routine in Listing 236 creates a listener by calling the function CreateUnoListener. When I tested this listener, I ran the macro and then the macro stopped running. The listener is still active and called by the broadcaster even though the macro is no longer running. In other words, the routine in Listing 236 registers a listener, and then it is finished. The broadcaster calls the listener when things occur. The listener must be stored in a global variable so that it's available later — for example, when it's time to remove the listener from the object or when one of the methods is called.

Listing 236. *Start listening for selection change events.*

```

Global vSelChangeListener 'Must be global
Global vSelChangeBroadCast

'Run this macro to start event intercepting
Sub StartListeningToSelChangeEvent
    Dim sPrefix$
    Dim sService$

```



```

sPrefix = "sel_change_"
sService = "com.sun.star.view.XSelectionChangeListener"

REM to register for a selection change, you must register with the
REM current controller
vSelChangeBroadCast = ThisComponent.getCurrentController

'Create a listener to intercept the selection change events
vSelChangeListener = CreateUnoListener(sPrefix, sService)

'Register the listener to the document controller
vSelChangeBroadCast.addSelectionChangeListener(vSelChangeListener)
End Sub

```

The behavior of the code in Listing 235 is annoying. Every time a new selection is made, the code is called and it opens a dialog on the screen. This interferes with selecting text. To remove the listener, use the code in Listing 237.

Listing 237. Stop listening for selection change events.

```

Sub StopListeningToSelChangeEvent
    ' removes the listener
    vSelChangeBroadCast.removeSelectionChangeListener(vSelChangeListener)
End Sub

```

To test the listener:

- 1) Start listening.
- 2) Make very simple selection changes such as: Click someplace. Shift+Click to select more. Ctrl+Click to make a second selection.
- 3) Stop listening.

Save all open documents while testing listeners, a simple error is likely to cause OOo to crash.

10.13. Creating a UNO dialog

Use CreateUnoDialog to create an existing dialog. This section introduces the function CreateUnoDialog, it does not discuss how to create dialogs. CreateUnoDialog accepts a single argument, the full path to the dialog definition.

```
CreateUnoDialog(GlobalScope.BasicLibraries.LibName.DialogName)
```

The variable GlobalScope provides access to the application-level libraries. The Tools library contains the dialog named DlgOverwriteAll; you must load the library containing the dialog before you can use the dialog. You can manually load the library, or do it directly from the macro. Use **Tools > Macros > Organize Dialogs** to open the Basic Macros Organizer dialog, which can be used To manually load libraries. Use the LoadLibrary command to load a library using a macro.

```
GlobalScope.BasicLibraries.LoadLibrary("Tools")
CreateUnoDialog(GlobalScope.BasicLibraries.Tools.DlgOverwriteAll)
```

If the dialog is defined inside a document, use the BasicLibraries variable.

```
CreateUnoDialog(BasicLibraries.LibName.DialogName)
```

TIP

An event means that something happened. When certain things happen, an event object is created that describes something about what happened. The event object is sent to a listener, or what ever routine is configured to be called when the event occurs. You can write an “event” routine that is called when something happens; for example, when a button is pressed.

When you create a dialog, you usually write event handlers as subroutines in Basic; for example, a button in a dialog does nothing unless events are mapped to subroutines or functions that you write. You can add a close button to a dialog and map the button's “Execute action” event to a macro that closes the dialog. If a macro called from an event handler accesses a running dialog, then the dialog must be stored in a variable that the macro can access. I generally choose to create a private variable that holds this type of data. See Listing 238.

Listing 238. *Display information about an object using a dialog.*

```
Private MySampleDialog
Sub DisplayObjectInformation(Optional vOptionalObj)
    Dim vControl 'Access the text control in the dialog.
    Dim s$       'Temporary string variable.
    Dim vObj      'Object about which to display information.

    REM If vOptionalObj object is not provided, use the current document.
    If IsMissing(vOptionalObj) Then
        vObj = ThisComponent
    Else
        vObj = vOptionalObj
    End If

    REM Create the dialog and set the title
    MySampleDialog = CreateUnoDialog(DialogLibraries.OOME_40.MyFirstDialog)
    MySampleDialog.setTitle("Variable Type " & TypeName(vObj))

    REM Get the text field from the dialog
    REM I added this text manually
    vControl = MySampleDialog.getControl("TextField1")

    If InStr(TypeName(vObj), "Object") < 1 Then
        REM If this is NOT an object, simply display simple information
        vControl.setText(Dlg_GetObjTypeInfo(vObj))

    ElseIf NOT HasUnoInterfaces(vObj, "com.sun.star.uno.XInterface") Then
        REM It is an object but it is not a UNO object.
        REM Cannot call HasUnoInterfaces if it is not an UNO object.
        vControl.setText(Dlg_GetObjTypeInfo(vObj))

    Else
        REM This is a UNO object so attempt to access the "dbg_" properties
        MySampleDialog.setTitle("Variable Type " & vObj.getImplementationName())
        s = "***** Methods *****" & CHR$(10) & _
            Dlg_DisplayDbgInfoStr(vObj.dbg_methods, ";") & CHR$(10) & _
            "***** Properties *****" & CHR$(10) & _
```

```

        Dlg_DisplayDbgInfoStr(vObj.dbg_properties, ";" & CHR$(10) & _
        "***** Services *****" & CHR$(10) & _
        Dlg_DisplayDbgInfoStr(vObj.dbg_supportedInterfaces, CHR$(10))
        vControl.setText(s)
    End If

    REM tell the dialog to start itself
    MySampleDialog.execute()
End Sub

```

TIP

Store the dialog in a private variable declared as type Object. Use a private variable so that it doesn't needlessly affect other modules. On at least one occasion, a dialog that I created worked when it was held in a variable declared as an Object, but failed when held in a variable declared as a Variant; so much for my recommendation to usually a variant rather than an Object.

The macro in Listing 238 displays a dialog that displays information about the object that was passed in as an argument. I use the HasUnoInterfaces function to see if this is a UNO service. I first check to see if the object's TypeName contains the text "Object"; this tells me if the object is really an object.

```

If InStr(TypeName(vObj), "Object") < 1 Then

```

If the object is not an object that supports a UNO interface, the object is inspected and information about it displayed. Listing 239 shows the text string to do this. Figure 77 shows the results.

Listing 239. *Return type information as a string.*

```

Function Dlg_GetObjTypeInfo(vObj) As String
    Dim s As String
    s = "TypeName = " & TypeName(vObj) & CHR$(10) & _
        "VarType = " & VarType(vObj) & CHR$(10)
    If IsNull(vObj) Then
        s = s & "IsNull = True"
    ElseIf IsEmpty(vObj) Then
        s = s & "IsEmpty = True"
    Else
        If IsObject(vObj) Then s = s & "IsObject = True" & CHR$(10)
        If IsUnoStruct(vObj) Then s = s & "IsUnoStruct = True" & CHR$(10)
        If IsDate(vObj) Then s = s & "IsDate = True" & CHR$(10)
        If IsNumeric(vObj) Then s = s & "IsNumeric = True" & CHR$(10)
        If IsArray(vObj) Then
            On Local Error Goto DebugBoundsError:
            Dim i%, sTemp$
            s = s & "IsArray = True" & CHR$(10) & "range = ("
            Do While (i% >= 0)
                i% = i% + 1
                sTemp$ = LBound(vObj, i%) & " To " & UBound(vObj, i%)
                If i% > 1 Then s = s & ", "
                s = s & sTemp$
            Loop
            DebugBoundsError:
            On Local Error Goto 0

```

```

        s = s & " " & CHR$(10)
    End If
End If
Dlg_GetObjTypeInfo = s
End Function

```

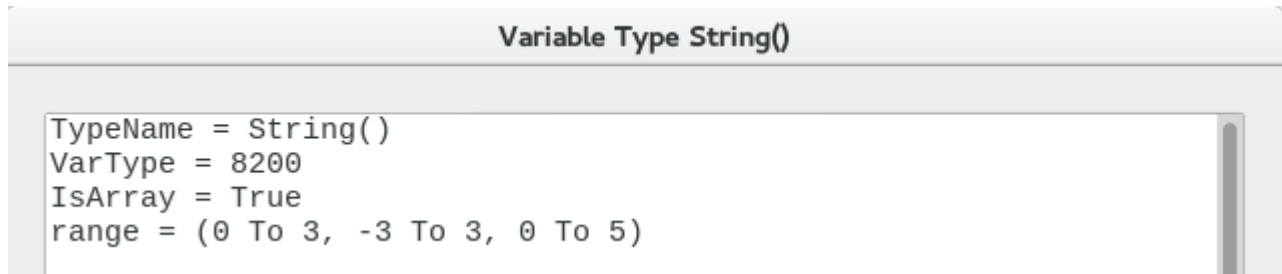


Figure 77. The variable is an array of strings

If the first argument supports a UNO interface, the “dbg_” properties are used to display all of the methods, properties, and services supported by this object (see Listing 240 and Figure 78). This is very similar to the code in Listing 221, except that it returns a string rather than displaying a series of simple dialogs. This is an excellent way to quickly see what methods, properties, and interfaces an object supports. Notice also that the variable type is displayed in the upper-left corner of the dialog.

Listing 240. Convert the debug string into one string with new line characters.

```

Function Dlg_DisplayDbgInfoStr(sInfo$, sSep$) As String
    Dim aInfo()           'Array to hold each string
    Dim i As Integer      'Index variable
    Dim j As Integer      'Junk integer variable for temporary values
    Dim s As String       'holds the portion that is not completed
    s = sInfo$
    j = InStr(s, ":")      'Initial colon
    If j > 0 Then Mid(s, 1, j, "") 'Remove portion up to the initial colon
    aInfo() = Split(s, sSep$) 'Split string based on delimiter
    For i = LBound(aInfo()) To Ubound(aInfo()) 'Look at each piece to remove
        aInfo(i) = Trim(aInfo(i)) 'leading and trailing spaces
        j = InStr(aInfo(i), CHR$(10)) 'Some have an extra
        If j > 0 Then Mid(aInfo(i), j, 1, "") 'new line that should be removed
    Next
    Dlg_DisplayDbgInfoStr = Join(aInfo(), CHR$(10))
End Function

```

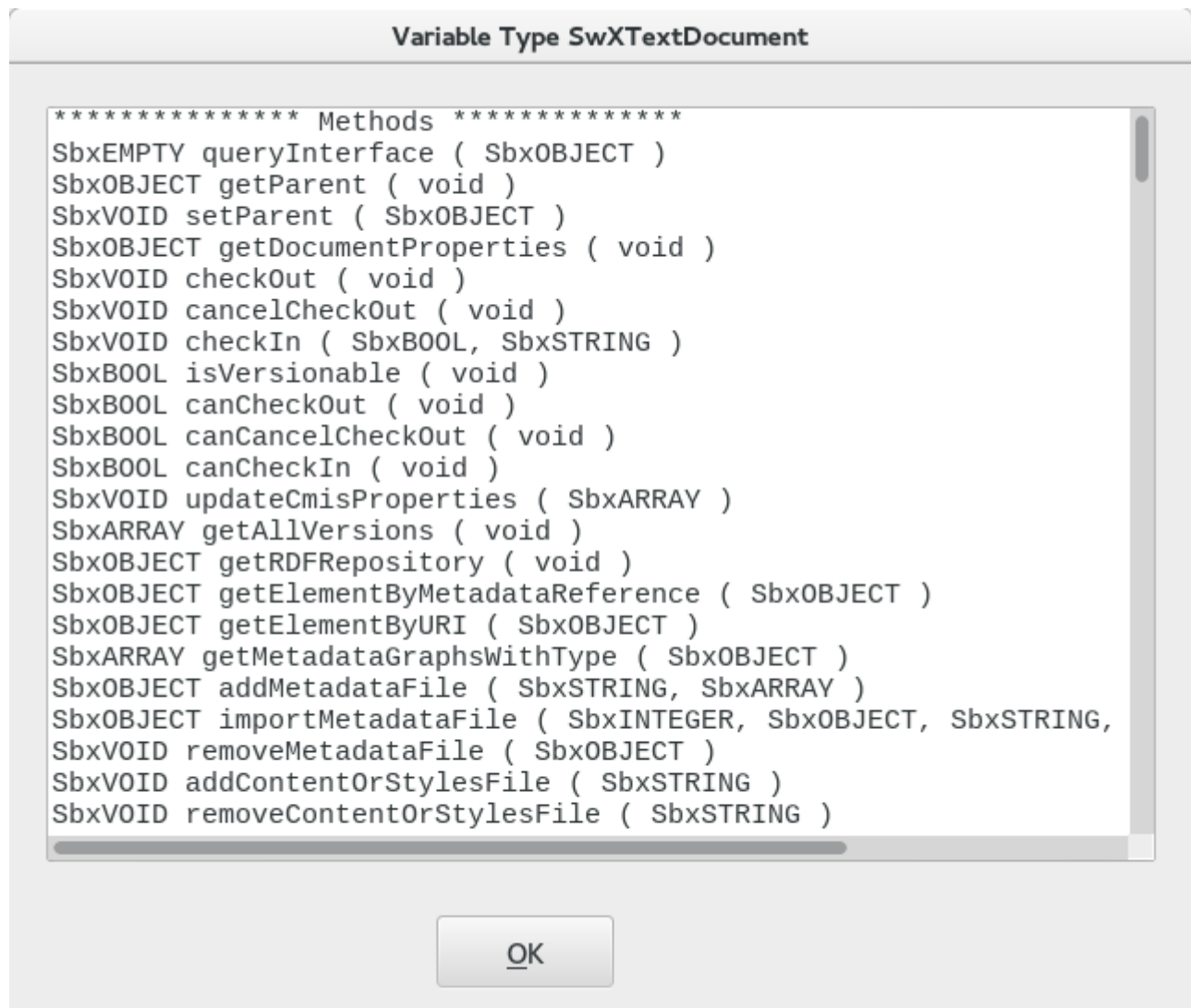


Figure 78. The variable is a Writer document.

10.14. Conclusion

The subroutines and functions supplied with Basic support a wide range of operations that are required to access the internals of OOo. Use the methods presented in this chapter to inspect and use the UNO services. This chapter examined the fundamentals for creating UNO listeners. UNO listeners provide basic system functionality, and new UNO listeners extend this functionality to give the programmer a great deal of control over system applications and behavior. Any UNO listener listens to one or more broadcasters, providing the capability to respond to system events that are communicated across applications.

11. The Dispatcher

This chapter builds on previous coverage of the LibreOffice paradigm and then introduces the dispatcher. The dispatcher provides a simple mechanism for invoking internal functionality with limited knowledge of how the internals work, but it is considered the least-favored way of controlling LO.

11.1. The environment

LibreOffice separates a component's (document's) functionality into the model, the controller, and the frame.

FRAME

Combines the pieces; it contains a model's controller and knows all about the display window. The frame does not, however, have any display window functionality; the frame just knows a display windows exist.

Send a dispatch to the frame, because it controls things and knows how to route a dispatch.

Model

Consists of the document data and the methods to change the data.

You can update data directly using the model.

Controller

Aware of the current view and the data; it manipulates the document while receiving input from the user interface.

The controller knows things like what is selected and is able to select things.

TIP

Component almost always means an open document, but other windows are also components; for example, the Basic IDE and the help window.

11.1.1. Two different methods to control LO

The most versatile method to manipulate an LO document is to get the internal UNO objects and manipulate them. Using the model provides significant control, but you must understand much about the different services and interfaces.

Another method, which requires very little understanding of the LO internals, is to use a UNO dispatcher. The dispatcher accepts a command such as “.uno:Undo” and takes care of the rest of the details. The frame provides the required dispatchers to perform the work. For simplicity, think of issuing a dispatch as very similar to selecting an action from the menu.

Although direct control using UNO services provides the most control and versatility, some operations are much easier to perform using the dispatcher; sometimes a dispatcher is the only way to accomplish a task. For example, the dispatcher is the best solution for handling the clipboard. Even the macro recorder performs almost all tasks using a dispatcher.

Three things are required to accomplish tasks using a dispatcher: (1) the command to dispatch, (2) arguments that control the dispatch, and (3) an object capable of performing the dispatch (a dispatch provider, which is usually a frame). Each document has a controller that, loosely speaking, acts as the interface between the outside world and the document. For example, use the current controller to select text, find the current cursor location, or determine which sheet is active in a spreadsheet. The current controller can also return the frame of the document, which supports the dispatch command.

Listing 241. *The DispatchHelper service simplifies executing dispatches.*

```
oDispatchHelper = createUnoService("com.sun.star.frame.DispatchHelper")
```

The dispatch helper implements the function `executeDispatch`, which implements most of the functionality required to perform a dispatch. Table 95 contains a list of the arguments supported by the `executeDispatch` method.

While executing a dispatch, as discussed here, you cannot dispatch a command to a frame based on its name — either leave the target frame empty or enter “_self”. It is also, therefore, not important to use anything other than zero, or omit the argument completely, for the search flag. Attempting to specify another frame results in a run-time error (it seems silly to me that the arguments exist if they are not usable¹).

Table 95. *Arguments to executeDispatch.*

Argument	Description
XDispatchProvider	Dispatch provider that performs the dispatch.
URL String	The command to dispatch, as a string.
Target Frame String	Identifies the frame that will receive the dispatch. Use an empty string or “_self” to specify the current frame (any other value is invalid).
long	Optional search flags that indicate how to find the target frame. Use zero or blank (see Listing 242), because it is not supported.
PropertyValue()	Optional arguments that are dependent on the implementation.

Listing 242. *Execute the “undo” dispatch.*

```
Sub NewUndo
    Dim oDispatchHelper as object
    Dim oProvider As Object
    oProvider = ThisComponent.CurrentController.Frame
    oDispatchHelper = createUnoService("com.sun.star.frame.DispatchHelper")
    oDispatchHelper.executeDispatch(oProvider, ".uno:Undo", "", , Array())
End Sub
```

1 I have not verified that a named frame does not work recently, but it did fail when I tested around version 2.0.

The dispatch commands have both a name and a number, called a “slot.” Although a dispatch can be made based on either, the original OpenOffice.org developers told me that the slot number may change in the future so it is safer to use the name. If you must use a slot, the Tools library contains a subroutine called DispatchSlot, which performs a dispatch based on a slot number alone.

Listing 243. Dispatch to a numeric slot.

```
'Include this library to use the DispatchSlot command.
GlobalScope.BasicLibraries.LoadLibrary("Tools")
DispatchSlot(5301) 'Load the About dialog, same as ".uno:About"
```

TIP

You cannot call a routine in a library unless the library is loaded. You can load a library manually from the Macro dialog and you can load it using the LoadLibrary command as shown in Listing 243. The Tools library included with LO contains the DispatchSlot subroutine.

Some dispatch commands require arguments. The example in Listing 244 perform a dispatch with arguments. The GoToCell command needs to know which cell to go to. This macro moves the current cursor to the cell B3 in a spreadsheet document.

Listing 244. Dispatch commands can use arguments.

```
Dim args2(0) as new com.sun.star.beans.PropertyValue
args2(0).Name = "ToPoint"
args2(0).Value = "$B$3" ' position to B3
Dim oDispHelper as object
Dim oProvider As Object
oProvider = ThisComponent.CurrentController.Frame
oDispHelper = createUnoService("com.sun.star.frame.DispatchHelper")
oDispHelper.executeDispatch(oProvider, ".uno:GoToCell", "", 0, args2())
```

11.1.2. Finding dispatch commands

A complete list of dispatch commands is tricky to find. Previously, I assembled a complete list of all supported dispatch commands, but, the list changes and it is not clear to me that it is of much use. As such, I decided to tell you how to find the list on your own. If there is a demand for it, I may be convinced to compile a new list. Thanks to Ariel Constenla Haile for this information.

Use the WIKI for OpenOffice.org

http://wiki.services.openoffice.org/wiki/Framework/Article/OpenOffice.org_3.x_Commands

contains a list of commands and slots. Clearly some commands are missing, such as “.uno:ObjectTitleDescription”. Even worse, this is for an older version of the software. Some of the names differ between AOO and LO, for example, view200 in OOO and Zoom200Percent in LO.

Read the source

I downloaded the source code and searched for instances of “.uno:” to find the UNO commands. This is a bit arduous, so I did not pursue it, but, it does work.

Probe the interface

The following macro uses the `UICommandDescription` service to enumerate the supported modules. A new Calc document is created and a sheet is created for each module. The commands supported by the module are placed on the sheet. Note that the macro takes a while to run because there is a lot of data to process.

Listing 245. Get commands from the ModuleManager.

```
Sub Print_All_Commands
    ' Create a new Calc document to hold the dispatch commands
    Dim oDoc As Object
    oDoc = StarDesktop.loadComponentFromURL("private:factory/scalc", "_default", 0, Array())

    Dim oSheets As Object : oSheets = oDoc.getSheets()

    'The UICommandDescription service provides access to the user interface commands that are
    'part of OpenOffice.org modules, like Writer or Calc.
    Dim oUICommandDescription As Object
    oUICommandDescription = CreateUnoService("com.sun.star.frame.UICommandDescription")

    'Identify office modules and provide read access to the configuration of office modules.
    Dim oModuleIdentifier As Object
    oModuleIdentifier = CreateUnoService("com.sun.star.frame.ModuleManager")

    Dim oModuleUICommandDescription As Object, aModules$(), aCommand
    Dim aCommands$()
    Dim n&, i&

    ' Get a list of module names such as "com.sun.star.presentation.PresentationDocument"
    ' Create a sheet for each module.
    aModules = oModuleIdentifier.getElementNames()
    For n = 0 To UBound(aModules)
        oModuleUICommandDescription = oUICommandDescription.getByName(aModules(n))

        ' Get the commands supported by this module.
        ReDim aCommands$()
        aCommands = oModuleUICommandDescription.getElementNames()

        If n <= UBound(oSheets.getElementNames()) Then
            oSheets.getByIndex(n).setName(aModules(n))
        Else
            oSheets.insertNewByName(aModules(n), n)
        End If

        oSheets.getCellByPosition(0, 0, n).getText().setString("Command")
        oSheets.getCellByPosition(1, 0, n).getText().setString("Label")
        oSheets.getCellByPosition(2, 0, n).getText().setString("Name")
        oSheets.getCellByPosition(3, 0, n).getText().setString("Popup")
        oSheets.getCellByPosition(4, 0, n).getText().setString("Property")

        For i = 0 To UBound(aCommands)
            aCommand = oModuleUICommandDescription.getByName(aCommands(i))
```

```

Dim sLabel$, sName$, bPopup as Boolean, nProperty&, k%
For k = 0 To UBound(aCommand)
    If aCommand(k).Name = "Label" Then
        sLabel = aCommand(k).Value
    ElseIf aCommand(k).Name = "Name" Then
        sName = aCommand(k).Value
    ElseIf aCommand(k).Name = "Popup" Then
        bPopup = aCommand(k).Value
    ElseIf aCommand(k).Name = "Property" Then
        nProperty = aCommand(k).Value
    End If
Next

oSheets.getCellByPosition(0, i+1, n).getText().setString(aCommands(i))
oSheets.getCellByPosition(1, i+1, n).getText().setString(sLabel)
oSheets.getCellByPosition(2, i+1, n).getText().setString(sName)
If bPopup Then
    oSheets.getCellByPosition(3, i+1, n).getText().setString("True")
Else
    oSheets.getCellByPosition(3, i+1, n).getText().setString("False")
End If
oSheets.getCellByPosition(4, i+1, n).getText().setString(CStr(nProperty))
Next

Dim oColumns as Object
oColumns = oSheets.getByIndex(n).getColumns()
Dim j%
For j = 0 To 4
    oColumns.getByIndex(j).setProperty("OptimalWidth", True)
Next
Next
End Sub

```

XDispatchInformationProvider constructs a list of dispatch commands as returned by the current controller and displays the list in a Calc document. When I wrote the macro below some years back, less than 32,000 rows were generated. In LO 6, this generates over 300,000 rows with some duplicates, so, be warned that this will run for a long time. I added a check to get out at 300,000 rows.

Listing 246. Get commands from the current controller.

```

Sub XDispatchInformationProvider
    Dim oDoc As Object
    oDoc = StarDesktop.loadComponentFromURL("private:factory/scalc", "_default", 0, Array())

    Dim oSheet As Object : oSheet = oDoc.getSheets().getByIndex(0)
    oSheet.getCellByPosition(0, 0).getText().setString("Group")
    oSheet.getCellByPosition(1, 0).getText().setString("Command")

    Dim oController : oController = ThisComponent.getCurrentController()
    If IsNull(oController) Or _
        NOT HasUnoInterfaces( oController, "com.sun.star.frame.XDispatchInformationProvider") Then

```

```

        'TODO: some warning
    Exit Sub
End If

Dim iSupportedCmdGroups%()
iSupportedCmdGroups = oController.getSupportedCommandGroups()

Dim i As Long, j As Long
Dim k As Long
For i = 0 To UBound(iSupportedCmdGroups)
    Dim aDispatchInfo()
    aDispatchInfo = oController.getConfigurableDispatchInformation(iSupportedCmdGroups(i))

    For j = 0 To UBound(aDispatchInfo)
        Dim aDispatchInformation
        aDispatchInformation = aDispatchInfo(j)
        k = k + 1
        oSheet.getCellByPosition(0, k).getText().setString(iSupportedCmdGroups(i))
        oSheet.getCellByPosition(1, k).getText().setString(aDispatchInformation.Command)
        ' I could trouble shoot this to see why this seems to not finish, but,
        ' I do not have the time right now.
        If k > 300000 Then
            Exit Sub
        End If
    Next
Next
End Sub

```

I exchanged email with a developer who expressed doubts as to the completeness and accuracy of the returned list; because the Bibliographic commands provided by the UICommandDescription included commands such as “.uno:ArrowShapes”, which is nonsensical in this context.

Read source code

SFX2 based components share a common base class for their Controller object implementation, SfxBaseController. For modules based on SFX2, it is possible to obtain the dispatches as well as the arguments by parsing the SDI files inside every sfx2 based module (look at the folder <module>/sdi/); or even parse the header files generated in the build environment. Consider files with names such as sfxslots.hxx, svxslots.hxx, scslots.hxx, sdslots.hxx, swslots.hxx, basslots.hxx, and smslots.hxx.

For new modules it may be easier to just browse the source code; for example, in chart2:

- chart2/source/controller/main/ChartController.cxx
- chart2/source/controller/main/DrawCommandDispatch.cxx
- chart2/source/controller/main/ShapeController.cxx

TIP Dispatch command names are case sensitive.

11.2. Writing a macro using the dispatcher

When you can't find a method to accomplish a task using the UNO API, the next choice is usually to use the dispatcher. Macros created by the macro recorder using dispatches; use **Tools | Macros | Record Macro** to start the macro recorder.

TIP Some component types do not support the macro recorder; for example, Draw.

The macro recorder has many limitations; for example, the macro recorder usually does not track what happens when a dialog is opened. I discovered this limitation when I used the macro recorder to create a macro to import text files. The “Text (encoded)” import filter opens a dialog and asks questions about the imported file. The values entered in the dialog are not captured by the macro recorder.

If you desire to write a macro using the API, start by reading through the supported commands as generated by one of the macros shown above.

The “SendOutlineToStarImpress” command creates an Impress presentation with an outline created from the headings in the current document. No arguments are required.

Listing 247. *Create an Impress document with an outline from this document.*

```
Sub CreateOutlineInImpress
    Dim oDispHelper as object
    Dim oProvider As Object
    oProvider = ThisComponent.CurrentController.Frame
    oDispHelper = createUnoService("com.sun.star.frame.DispatchHelper")
    oDispHelper.executeDispatch(oProvider, ".uno:SendOutlineToStarImpress", _
        "", , Array())
End Sub
```

11.3. Dispatch failure – an advanced clipboard example

The request was easy, copy an entire Writer document to the clipboard. The macro recorder very quickly provided a solution. Unfortunately, the macro fails in LO when called from a button added to the document; but it works from the IDE and from AOO when called from a button. The solution is to set the focus on the document, before executing the dispatches.

Listing 248. *Select all and copy using dispatches.*

```
Sub CopyToClipboard_Dispatch
    dim document as object
    dim dispatcher as object
    document = ThisComponent.CurrentController.Frame

    ' This next line was NOT added by the macro recorder.
    ' Without the next line, this fails in LO when called from a button.
    document.ContainerWindow.setFocus

    dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")
    dispatcher.executeDispatch(document, ".uno:SelectAll", "", 0, Array())
```

```
dispatcher.executeDispatch(document, ".uno:Copy", "", 0, Array())
End Sub
```

The API is more stable than the dispatches and works where a dispatch may fail. Another example where the API may be required is when the macro is invoked from LibreOffice started in headless mode; a topic not otherwise discussed in this document.

Listing 249. Select all and copy using the API.

```
Sub CopyToClipboard_API()
    Dim o          ' Transferable content.
    Dim oClip      ' Clipboard service.
    Dim oContents
    Dim sClipName As String

    ThisComponent.CurrentController.Select(ThisComponent.Text)
    o = ThisComponent.CurrentController.getTransferable()

    sClipName = "com.sun.star.datatransfer.clipboard.SystemClipboard"
    oClip = createUnoService(sClipName)
    oContents = oClip.setContents(o, null)
End Sub
```

11.4. Conclusion

The dispatch commands are powerful and require little knowledge of the internal workings of LO. Although some functionality, such as the Undo command, can only be used in the dispatcher, macros that will be used for a long time are better off using the internal objects directly.

12. StarDesktop

The desktop acts as the main application that controls LibreOffice. This chapter introduces general techniques — such as accessing indexed objects, enumerating open documents, and loading new documents — while discussing and demonstrating the base functionality of the Desktop object. This chapter also covers the Desktop object and ThisComponent.

The Desktop object is a `com.sun.star.frame.Desktop` service, which supplies four primary functions:

1. StarDesktop acts as a frame. The desktop is the parent frame and controls the frames for all of the documents.
2. StarDesktop acts as a desktop. The desktop is the main application with the ability to close the documents — when the application is shut down, for example.
3. StarDesktop acts as a document loader. The role of the main application also allows the desktop to load existing documents and create new documents.
4. StarDesktop acts as an event broadcaster. The desktop notifies existing documents (or any other listener) of things that are happening — when the application is about to shut down, for example.

The Desktop, acting as the primary application object, is created when LO is started. Use the globally available variable `StarDesktop` to access the LO Desktop object.

12.1. The Frame service

The desktop is a `com.sun.star.frame.Frame` service (remember that an object can, and usually does, implement more than one service). For a document, the primary purpose of a frame is to act as a liaison between the document and the visible window. For the desktop, however, the primary purpose is to act as the root frame, which contains all other frames. A frame can hold a component and zero or more subframes — for simplicity, consider a component to be a document. In the case of the desktop, all the other frames are subframes of the root frame, and the component (document) is a set of data. Simply stated, each document contains a frame that it uses to interact with the visible window, but the desktop is a frame so that it can contain, control, and access all of the document's frames.

The service `com.sun.star.frame.Frame` — Frame for short — provides numerous interesting capabilities that are not generally useful as part of the Desktop object. For example, the `Title` and the `StatusIndicator` defined as part of the Frame service are of no use for the Desktop object because the Desktop object does not have a displayed window. These properties are meaningful only in a frame that contains a display window. The desktop is a frame so that it can act as the root frame for all the other frames.

TIP

Although it is possible to use the desktop as a Frame service to enumerate the contained frames, the XDesktop interface is generally more useful to enumerate the documents rather than the frames. I usually access frames to obtain window titles.

Use the `getActiveFrame()` method of the desktop to obtain the active frame (see Listing 250). The active frame is the frame that contains the current focus. If the currently focused window is not an LO window, `getActiveFrame` returns the last LO frame that had focus.

Listing 250. *Print the title of the current frame.*

```
Print StarDesktop.getActiveFrame().Title
```

Use the `getFrames()` method to enumerate or search all of the frames contained in the desktop. The `getFrames()` method returns an object that implements the `com.sun.star.frame.XFrames` interface. A frame can contain other frames; the XFrames interface provides access to the contained frames.

TIP

Use the full interface name to find the Web address for the API information on the XFrames interface. It is important that you learn to find the Web pages on the API site from the full service or interface name.

12.1.1. The XIndexAccess interface

The XFrames interface is derived from the `com.sun.star.container.XIndexAccess` interface. As its name implies, this interface allows access to the contained frames using a numeric index. Numerous other interfaces also derive from the XIndexAccess interface, allowing a simple numeric index to access the contained elements. See Listing 251 and Figure 79. The sheets object in a Calc document allows you to access each sheet based on the index, or the name.

TIP

Learn how to use the XIndexAccess interface because OOO uses this service in numerous other places.

Listing 251. *Display frame titles of open components.*

```
Sub DisplayFrameTitles
    Dim vFrames As Variant           'All of the frames
    Dim vFrame As Variant           'A single frame
    Dim i As Integer                'Index to enumerate the frames
    Dim s As String                 'Contains the string to print

    vFrames = StarDesktop.getFrames() 'Get all of the frames

    REM getCount() returns the number of contained frames
    REM If there are four frames, then i has the values 1, 2, 3, and 4
    REM the getByIndex(i) method, however, is zero based. This means
    REM that it requires the values 0, 1, 2, and 3
    For i = 1 To vFrames.getCount()
        vFrame = vFrames.getByIndex(i-1)
        s = s & CStr(i-1) & " : " & vFrame.Title & CHR$(10)
    Next
    MsgBox s, 0, "Frame Titles"
End Sub
```

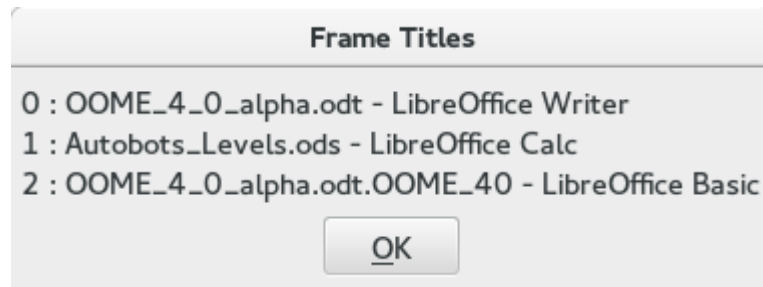


Figure 79. Titles of top-level frames.

12.1.2. Find frames with `FrameSearchFlag` constants

Use the `com.sun.star.frame.FrameSearchFlag` constants to search frames (see Table 96). `FrameSearchFlag` constants are used to create an enumerated list of frames from the `queryFrames()` method defined in the `XFrames` interface. The `FrameSearchFlag` constants are also used to find a frame when loading a document and specifying which frames will receive a dispatch. In other words, you will see the `FrameSearchFlag` constants again.

Table 96. `com.sun.star.frame.FrameSearchFlag` constants.

#	Name	Description
0	AUTO	Deprecated. Use 6 = SELF+CHILDREN.
1	PARENT	Include the parent frame.
2	SELF	Include this frame.
4	CHILDREN	Include the child frames of this frame.
8	CREATE	Create a frame if the requested frame is not found.
16	SIBLINGS	Include the child frames of the parent of this frame.
32	TASKS	Include all frames in all tasks in the current frames hierarchy.
23	ALL	Include all frames except TASKS frames. $23 = 1+2+4+16 = \text{PARENT} + \text{SELF} + \text{CHILDREN} + \text{SIBLINGS}$.
55	GLOBAL	Include every frame. $55 = 1+2+4+16+32 = \text{PARENT} + \text{SELF} + \text{CHILDREN} + \text{SIBLINGS} + \text{TASKS}$.

The values in Table 96 are enumerated constants. OOo has enumerated values for many different purposes. You can access all enumerated values in Basic in a similar fashion. Each constant has an assigned name, as seen in the Name column in Table 96. You can use this name by preceding it with `com.sun.star.frame.FrameSearchFlag`. For example, use `com.sun.star.frame.FrameSearchFlag.TASKS` to use the TASKS constant (see Listing 252). The name of a constant provides meaningful information while reading the Basic code. Using the constant values directly, although allowed, obfuscates your code — in other words, causes it to be less readable.

TIP Constant names are case sensitive the first time that they are used. If the two lines in Listing 252 are switched, then a run-time error occurs because the version with the wrong case is found first.

Listing 252. *Constant names are not case sensitive the second time they are used.*

```
Print com.sun.star.frame.FrameSearchFlag.TASKS '32
Print COM.SUN.STAR.frame.FrameSearchFLAG.taskS '32 - Works because it is already known.
```

The values in Table 96 are flags that may be combined. For example, the values for ALL and GLOBAL exist only as a convenience; they are a combination of the other flags as shown in Table 96. You can create your own values if a suitable combination is not provided. For example, to search ALL and to create a frame if it is not found, use the value $31 = 23 + 8 = \text{ALL} + \text{CREATE}$.

TIP	Frame search flags are bit-flags that can be combined by using the OR operator. If you do not know what a bit-flag is, just smile and nod.

The code in Listing 253 uses the FrameSearchFlag constants to search the frames that are children of the Desktop object. The output from Listing 253 duplicates the output from Listing 251 by obtaining a list of all child frames from the Desktop object. Note that the return type from the queryFrames() method is an array. I know this because I looked at the API Web site. Although you can inspect the returned object to see what it is, it isn't possible to determine the values of the arguments to the queryFrames() method by inspection alone.

Listing 253. *Query frames to get frame titles.*

```
Sub QueryFrames
    Dim vFrames As Variant           'All of the frames
    Dim vFrame As Variant           'A single frame
    Dim i As Integer                 'Index to enumerate the frames
    Dim s As String                  'Contains the string to print
    REM Call the queryFrames() method on the XFrames interface.
    REM This takes one argument, a FrameSearchFlag.
    REM This searches the children of the desktop.
    vFrames = StarDesktop.getFrames().queryFrames(_
        com.sun.star.frame.FrameSearchFlag.CHILDREN)

    For i = LBound(vFrames) To UBound(vFrames) ' The return value is an array.
        s = s & vFrames(i).Title & CHR$(10)    ' Append the title and a new line.
    Next
    MsgBox s, 0, "Frame Titles"                ' Display the titles.
End Sub
```

12.2. The XEventBroadcaster interface

When certain important events occur, the event is broadcast to all the objects that register as listeners for a specific event — for example, to be notified before a document is closed. The com.sun.star.document.XEventBroadcaster interface allows the desktop to act as a broadcaster.

The XEventBroadcaster interface defines the methods addEventListener() and removeEventListener() to add and remove event listeners. These two methods are not usually used directly because methods that are specific to the listener are typically used. For example, the

Controller object contains the methods `addKeyHandler()` and `removeKeyHandler()` to add and remove listeners for key-press events.

12.3. The XDesktop interface

The `com.sun.star.frame.XDesktop` interface defines the primary functionality of the Desktop service. The desktop contains the top-level components that can be viewed in a frame. In other words, it contains and controls the life cycle of the OpenOffice.org documents, help window, Basic IDE, and the other component types.

The desktop is a Frame service so that it can act as the root frame containing all the other frames. The desktop, therefore, has access to and controls all the other frames. This control includes the ability to load documents and the ability to close all frames and exit LO.

12.3.1. Closing the desktop and contained components

To close the desktop and all the contained frames, call the `terminate()` method. This method is not guaranteed to close the desktop; it is merely a suggestion (or request) that you would like LO to close. Before closing, the desktop asks every object that asked to be notified before closing if it is OK to close. If any listener says no, then LO will not terminate. Every open document registers as a terminate listener and asks you if you want to save a document if it has not yet been saved; in case you wondered how that works.

All document types support the `com.sun.star.util.XCloseable` interface. The `XCloseable` interface defines the method “`close(bForce As Boolean)`”. If `bForce` is `False`, the object may refuse to close. If `bForce` is `True`, the object is not able to refuse.

According to Mathias Bauer, one of the original lead developers, the Desktop object does not support the `XCloseable` interface for legacy reasons. The `terminate()` method was used before it was determined to be inadequate for closing documents or windows. If the `terminate()` method was not already implemented, then the desktop would also support the `XCloseable` interface.

The code in Listing 254 demonstrates the safe way to close a document using any version of OOo. If you know that your code will run on OOo 1.1 or later, you can simply use the `close()` method.

Listing 254. *The safe way to close a document using any version of OOo.*

```
If HasUnoInterfaces(oDoc, "com.sun.star.util.XCloseable") Then
    oDoc.close(true)
Else
    oDoc.dispose()
End If
```

The code in Listing 254 assumes that the variable `oDoc` references a document. Some component types do not support the `XCloseable` interface. One example is the Basic IDE. The desktop contains methods to enumerate the currently open components and to access the current component. These methods are discussed shortly.

The `dispose()` method automatically discards a document even if it is modified, while the `close()` method does not. Use `setModified(False)` to mark a modified document as not modified, thereby preventing the `close()` method from complaining that you are closing a modified document.

12.3.2. Enumerating components using `XEnumerationAccess`

Usually, a component refers to a document, but it may refer to other things, such as the Basic IDE or the included help pages. Use the `getComponents()` method defined in the `XDesktop` interface to return an enumeration of the components controlled by the desktop. The returned object supports the interface `com.sun.star.container.XEnumerationAccess`.

TIP LO has many interfaces for returning a list of objects — some methods return an array of objects. In the desktop, `XIndexAccess` is used to enumerate frames and the `XEnumerationAccess` interface is used to enumerate components.

A component that is also a document supports the `XModel` interface. The model represents the underlying document data, so if a component doesn't support the `XModel` interface, it isn't an OOo document. In other words, supporting the `XModel` interface implies that a component has data; if it doesn't have data, then it isn't a document. Use the `HasUnoInterfaces` function to check each component to see if it is an OOo document. To find a specific document, search all of the components and check the URL or some other distinguishing characteristic.

It is possible that a document does not have a URL, in this case, the returned URL is an empty string. The subroutine `FileNameOutOfPath` fails with an empty string; so `EnumerateComponentNames` checks for this case.

Listing 255. Demonstrate how to enumerate components.

```
Sub EnumerateComponentNames
    Dim vComps          'Enumeration access object
    Dim vEnumerate      'Enumeration object
    Dim vComp           'Single component
    Dim s As String     'Display string
    Dim sURL As String  'Document URL

    REM Load the Tools library because Strings module contains
    REM the function FileNameOutOfPath()
    GlobalScope.BasicLibraries.LoadLibrary("Tools")

    vComps=StarDesktop.getComponents() 'com.sun.star.container.XEnumerationAccess
    If NOT vComps.hasElements() Then    'I do not have to do this, but
        Print "There are no components" 'this demonstrates that I can
        Exit Sub
    End If

    vEnumerate = vComps.createEnumeration() 'com.sun.star.container.XEnumeration
    Do While vEnumerate.hasMoreElements() 'Are there any elements to retrieve?
        vComp = vEnumerate.nextElement() 'Get next element

        REM Only try to get the URL from document components
```

```

REM This will skip the IDE, for example
If HasUnoInterfaces(vComp, "com.sun.star.frame.XModel") Then
    sURL = vComp.getURL()
    If sURL = "" Then
        s = s & "<Component With No URL>" & CHR$(10)
    Else
        s = s & FileNameOutOfPath(sURL) & CHR$(10)
    End If
End If
Loop
MsgBox s, 0, "Document Names"
End Sub

```

Figure 80, produced by Listing 6, lists the file names of all currently open documents.

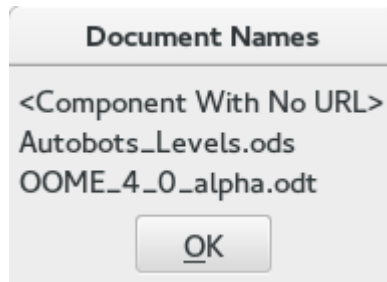


Figure 80. File names of currently open documents.

12.3.3. Current component

Use `getCurrentComponent()` to return the currently focused component, which is not necessarily a document. To obtain the most current document, use the globally defined variable `ThisComponent`.

Listing 256. *ThisComponent references the current document.*

```

Print ThisComponent.getURL()           'Works from Basic IDE
Print StarDesktop.getCurrentComponent().getURL() 'Fails from Basic IDE

```

TIP `StarDesktop.getCurrentComponent()` returns the currently focused component. If the Basic IDE is open, the Basic IDE component is returned. Use the globally defined variable `ThisComponent` to return the most current document.

12.3.4. Current component (again)

Although `StarDesktop.getCurrentComponent()` can be used in many situations, it is not a reliable method to obtain the document that most recently had focus. The method `getCurrentComponent()` returns either the component that currently has focus, or the component that had the focus before control was switched to a different application. This behavior causes problems when attempting to debug Basic programs, because the Basic IDE is the current component. It is therefore preferable to use `ThisComponent` rather than

StarDesktop.CurrentComponent. Forcing a new document's window to become focused does not change the value of ThisComponent. The value of ThisComponent is set when the macro begins running and then it is not changed.

Listing 257. Use one of these two methods to cause oDoc2 to become the focused document.

```
oDoc2.CurrentController.Frame.ContainerWindow.toFront()  
oDoc2.CurrentController.Frame.Activate()
```

TIP

When a macro is called because an event occurred, ThisComponent refers to the document that produced the event unless the macro is contained in another document. Therefore, it is better to place global macros that will be called from other documents inside a global library rather than inside a document.

Macros may be called when certain events occur (perhaps as a registered event handler). When a macro is called because an event occurred, ThisComponent refers to the document that produced the event even if it isn't the currently focused document. The variable ThisComponent is available only from BASIC.

TIP

It is possible to run a macro when no document is open; for obvious reasons, the macro must reside in the application library rather than in a document.

12.3.5. Current frame

Although the Frame service allows the desktop to enumerate the contained frames, it is the XDesktop interface that defines the method getCurrentFrame(). The getCurrentFrame() method returns the frame of the currently focused component.

TIP

The component contained in the current frame may, or may not, be a document.

The code in Listing 258 shows one of many things you can do with the current frame. The macro obtains the current frame, the frame returns the current window, and then the macro reduces the size of the current window.

Listing 258. Shrink the size of the current window by 25%

```
REM Shrink the current window to 75% of its current size.
```

```
Sub ShrinkWindowBy75
```

```
    Dim vFrame      'Current frame
```

```
    Dim vWindow     'container window
```

```
    REM And this is a struct that holds a rectangle object
```

```
    REM I could have also used a Variant or an Object to
```

```
    REM hold the return type, but I know what it is, so I
```

```
    REM defined the correct type to hold it.
```

```
    Dim vRect As New com.sun.star.awt.Rectangle
```

```
    vFrame = StarDesktop.getCurrentFrame()
```

```
    vWindow = vFrame.getContainerWindow()
```

```
    REM As a struct, the object is copied by value, not by reference.
```

```
    REM This means that I can modify vRect and it will not affect
```

```

REM the position and size of the window.
REM In my tests, vRect.X and vRect.Y are zero, which is wrong.
REM vRect.Width and vRect.Height are indeed correct.
vRect = vWindow.getPosSize()

REM When setting the position and size, the last argument specifies
REM which of the arguments to use.
'com.sun.star.awt.PosSize.X          Set only the X-position
'com.sun.star.awt.PosSize.Y          Set only the Y-position
'com.sun.star.awt.PosSize.WIDTH      Set only the Width
'com.sun.star.awt.PosSize.HEIGHT     Set only the Height
'com.sun.star.awt.PosSize.POS        Set only the Position
'com.sun.star.awt.PosSize.SIZE       Set only the Size
'com.sun.star.awt.PosSize.POSSIZE    Set both the Position and Size
vWindow.setPosSize( vRect.X, vRect.Y, 3*vRect.Width/4, 3*vRect.Height/4, _
    com.sun.star.awt.PosSize.SIZE )

End Sub

```

12.4. Load a document

The desktop implements the `com.sun.star.frame.XComponentLoader` interface. This is a simple interface to load components from a URL. Use the method `LoadComponentFromUrl()` defined in the `XComponentLoader` interface to load an existing document or create a new one.

```

com.sun.star.lang.XComponent loadComponentFromUrl(
    String aURL,
    String aTargetFrameName,
    Long nSearchFlags,
    sequence< com::sun::star::beans::PropertyValue > aArgs)

```

The first argument to the method `LoadComponentFromUrl()` specifies the URL of the document to load. To load an existing document, provide the URL of the document. Use the function `ConvertToURL` to convert a file name expressed in an operating-system-specific format to a URL.

```
Print ConvertToURL("c:\temp\file.txt") 'file:///c:/temp/file.txt
```

The URL may indicate a local file, or even a file on the Internet (see Listing 259). I recommend obtaining a high-speed Internet connection before running this macro because it loads a 500-page document. Also, do not run this macro unless you have OOo 3.4 or later, because when you close `AndrewMacro`, it will cause OOo to crash.

Listing 259. Load document over HTTP.

```

Sub LoadMacroDocFromHttp
    Dim noArgs()          'An empty array for the arguments.
    Dim vComp             'The loaded component
    Dim sURL As String    'URL of the document to load
    sURL = "http://www.pitonyak.org/AndrewMacro.odt"
    vComp = StarDesktop.LoadComponentFromUrl(sURL, "_blank", 0, noArgs())
End Sub

```

LO uses special URLs to indicate that a new document, rather than an existing document, should be created.

Table 97. URLs for creating new documents.

URL	Document Type
"private:factory/scalc"	Calc document
"private:factory/swriter"	Writer document
"private:factory/swriter/web"	Writer HTML Web document
"private:factory/swriter/GlobalDocument"	Master document
"private:factory/sdraw"	Draw document
"private:factory/smath"	Math formula document
"private:factory/simpress"	Impress presentation document
"private:factory/schart"	Chart
".component:Bibliography/View1"	Bibliography — Edit the bibliography entries
".component:DB/QueryDesign"	Database components
".component:DB/TableDesign"	
".component:DB/RelationDesign"	
".component:DB/DataSourceBrowser"	
".component:DB/FormGridView"	

The macro in Listing 260 opens five new documents — each in a new window.

Listing 260. Create new documents.

```
Sub LoadEmptyDocuments
    Dim noArgs()           'An empty array for the arguments
    Dim vComp              'The loaded component
    Dim sURLs()            'URLs of the new document types to load
    Dim sURL As String     'URL of the document to load
    Dim i As Integer
    sURLs = Array("scalc", "swriter", "sdraw", "smath", "simpress")
    For i = LBound(sURLs) To UBound(sURLs)
        sURL = "private:factory/" & sURLs(i)
        vComp = StarDesktop.LoadComponentFromUrl(sURL, "_blank", 0, noArgs())
    Next
End Sub
```

TIP You can also load a document from a frame.

When a component (document) is loaded, it is placed into a frame; the second argument to `LoadComponentFromUrl()` specifies the name of the frame. If a frame with the specified name already exists, the newly loaded document uses the existing frame. The code in Listing 260 demonstrates the special frame name “_blank” — the special frame names are discussed shortly. Frame names that are not “special,” specify the name of an existing frame or a new frame. If you specify a frame name, you must tell OOo how to find the frame. The values in Table 96 enumerate valid values for the frame search flags. It used to be possible to specify the name of the frame that will contain a document (but this has not worked since OOo 1.1.0). Here are a few items to note:

1. When a document is loaded, the frame name becomes something like “Untitled 1 – OpenOffice.org Writer” rather than the specified name.
2. If you specify the frame name, the document will fail to load if a frame with that name does not exist. If the frame does exist, it will load it into a new frame.

If you experiment with this, be certain to save your documents before running the macro, at least one experiment caused the new document to load into the current frame.

Listing 261. *Open a document into an existing frame.*

```
Sub UseAnExistingFrame
    Dim noArgs()           'An empty array for the arguments
    Dim vDoc               'The loaded component
    Dim sURL As String     'URL of the document to load
    Dim nSearch As Long    'Search flags
    Dim sFName As String   'Frame Name
    Dim vFrame             'Document Frame
    Dim s As String        'Display string

    REM Search globally for this
    nSearch = com.sun.star.frame.FrameSearchFlag.GLOBAL + _
              com.sun.star.frame.FrameSearchFlag.CREATE

    REM I can even open a real file for this, but I don't know what files
    REM you have on your computer so I create a new Writer document instead
    sURL = "file:///andrew0/home/andy/doc1.odt"
    sURL = "private:factory/swriter"

    REM Create a frame with the name MyFrame rather than _default
    sFName = "MyFrame"
    vFrame = ThisComponent.CurrentController.Frame
    vDoc = vFrame.LoadComponentFromUrl(sURL, sFName, nSearch, noArgs())
    If IsNull(vDoc) Then
        Print "Failed to create a document"
        Exit Sub
    End If

    REM The name of the frame is MyFrame. Note that the name has nothing
    REM to do with the title!
    sFName = vDoc.CurrentController.Frame.Name
    s = "Created document to frame " & sFName & CHR$(10)
    MsgBox s

    REM This time, do not allow creation; only allow an existing frame
    nSearch = com.sun.star.frame.FrameSearchFlag.Global
    sURL = "file:///andrew0/home/andy/doc2.odt"
    sURL = "private:factory/scalc"
    'sFName = "doc1 - OpenOffice.org Writer"
    vDoc = vFrame.LoadComponentFromUrl(sURL, sFName, nSearch, noArgs())
    If IsNull(vDoc) Then
        Print "Failed to create a document"
        Exit Sub
    End If
End Sub
```



```

End If
s = s & "Created document to frame " & sFName
MsgBox s
End Sub

```

LO uses special frame names to specify special behavior (see Table 98). Use these special frame names to cause the specified behavior.

Table 98. Special frame names.

Frame Name	Description
"_blank"	Creates a new frame.
"_default"	Detects an already loaded document or creates a new frame if it is not found.
"_self"	Use or return this frame.
""	Use or return this frame.
"_parent"	Use or return the direct parent of this frame.
"_top"	Use or return the highest level parent frame.
"_beamer"	Use or return a special subframe.

Although you cannot currently cause LO to find a frame and use it, you can search for the frame with the desired name and then use that frame to load a document into that frame; by using the frame name “_self”.

Listing 262. ReuseAFrame

```

Sub UseFrameSelf
    Dim noArgs()           'An empty array for the arguments
    Dim vDoc               'The loaded component
    Dim sURL As String     'URL of the document to load
    Dim nSearch As Long   'Search flags
    Dim sFName As String   'Frame Name
    Dim vFrame             'Document Frame
    Dim s As String        'Display string

    REM Search globally for this
    nSearch = com.sun.star.frame.FrameSearchFlag.GLOBAL + _
              com.sun.star.frame.FrameSearchFlag.CREATE

    REM I can even open a real file for this, but I don't know what files
    REM you have on your computer so I create a new Writer document instead
    sURL = "file:///andrew0/home/andy/doc1.odt"
    sURL = "private:factory/swriter"

    REM Create a frame with the name MyFrame rather than _default
    sFName = "MyFrame"
    vFrame = ThisComponent.CurrentController.Frame
    vDoc = vFrame.LoadComponentFromUrl(sURL, sFName, nSearch, noArgs())
    If IsNull(vDoc) Then
        Print "Failed to create a document"
    Exit Sub

```

```

End If

REM The name of the frame is MyFrame. Note that the name has nothing
REM to do with the title!
sFName = vDoc.CurrentController.Frame.Name
s = "Created document to frame " & sFName & CHR$(10)
MsgBox s

REM This time, do not allow creation; only allow an existing frame
sURL = "file:///andrew0/home/andy/doc2.odt"
sURL = "private:factory/scalc"
sFName = "_self"
' Get a document's frame from the document's current controller.
vDoc = vDoc.CurrentController.Frame.LoadComponentFromUrl(sURL, sFName, nSearch, noArgs())
If IsNull(vDoc) Then
    Print "Failed to create a document"
    Exit Sub
End If
s = s & "Created document to frame " & sFName
MsgBox s
End Sub

```

12.4.1. Named arguments

The final argument to the `LoadComponentFromUrl()` method is an array of structures of type `com.sun.star.beans.PropertyValue`. Each property value consists of a name and a value. The properties are used to pass named arguments that direct LO when it loads the document. Table 99 contains a brief description of the supported named arguments.

TIP Table 99 contains only named arguments that are not deprecated. The API Web site on the `com.sun.star.document.MediaDescriptor` service contains a complete list of named arguments, deprecated and otherwise. Some properties are not shown, such as `Aborted`, because they are set while loading; for example, `Aborted`, which is set if an incorrect password is used.

See:

https://api.libreoffice.org/docs/idl/ref/servicecom_1_1sun_1_1star_1_1document_1_1MediaDescriptor.html
https://wiki.openoffice.org/wiki/Documentation/DevGuide/OfficeDev/Handling_Documents#MediaDescriptor

Table 99. Valid named arguments for loading and storing documents.

Argument	Description
AsTemplate	A value of True creates a new untitled document, even if the document is not a template. The default loads the template document for editing.
Author	Sets the current author if the component can track the author of the current version when the document is saved.
CharacterSet	Identifies the character set for single-byte characters.
Comment	Similar to the Author argument, but sets the document description for version control.
ComponentData	Allows component-specific properties.

Argument	Description
DocumentBaseURL	The base URL of the document to be used to resolve relative links.
DocumentTitle	Sets the Document title.
EncryptionData	Encryption information for encryption/decryption of documents; sadly, I have no examples using this.
FilterData	Additional properties for a filter if it is required.
FilterName	Name of the filter for loading or storing the component when not using LO types.
FilterOptions	Additional properties for a filter if it is required. These values must be strings and must be supported by the filter. Use FilterData for non-string options.
Frame	Specify the frame containing the document.
Hidden	A value of False loads the document so that it is hidden. Do not do this if you intend to make the document visible after it is loaded.
HierarchicalDocumentName	The hierarchical path to the embedded document from topmost container; for example, Base forms are stored in the Base document.
InputStream	You can specify an existing input stream for loading a document — for example, if you have the document in memory and don't want to write it to disk first.
InteractionHandler	Passes an interaction handler to handle errors and obtains a password if required.
JumpMark	Jumps to a marked position after loading the component. A similar thing can be accomplished by appending a '#' and the jump mark name in the URL used to open the document; the syntax with a '#' is not specified in most URL schemas.
MacroExecutionMode	The numeric value specifies if macros are executed when the document is loaded (see Table 100).
MediaType	Specifies the MIME type of the data that will be loaded.
OpenNewView	A value of True forces the component to create a new window even if the document is already loaded. Some components support multiple views for the same data. If the opened component does not support multiple views, a new window is opened. It is not a view, but merely the document loaded one more time.
OutputStream	If used when storing a document (not for reading), the stream must be used to write the data. If a stream is not provided, the loader will create a stream by itself using the other properties.
Overwrite	A value of True overwrites any existing file with the same name while saving.
Password	Password for loading or storing a document. If a document that requires a password is loaded and no password is specified, the document is not loaded.
PostData	Posts data to an HTTP address and then loads the response as a document. PostData is usually used to obtain a result from a Web form on a Web site.
Preview	A value of True specifies that the document is loaded for preview. OOo may make some optimizations when it opens a document in preview mode.
ReadOnly	Opens the document as read-only. Read-only documents are not

Argument	Description
	modifiable from the user interface, but you can modify them by using the API (in other words, from a macro).
Referer	A URL indicating the referrer who opened this document. Without a referrer, a document that requires security checks is denied. (Yes, the argument name is “Referer,” not “Referrer.”)
RepairPackage	Documents are stored in a compressed zipped format. A value of True attempts to recover information from a damaged ZIP file.
StartPresentation	If the document contains a presentation, it is started immediately.
StatusIndicator	Value is an object to use as a progress indicator as the document is loaded or saved.
SuggestedSaveAsDir	Specify the URL that is used next time SaveAs dialog is opened.
SuggestedSaveAsName	Specify the suggested file name that is used next time SaveAs dialog is opened.
TemplateName	Name of the template instead of the URL. Must also use the TemplateRegionName.
TemplateRegionName	Path to the template instead of the URL. Must also use the TemplateName.
Unpacked	LO stores documents in a zipped format. A value of True stores the file in a folder if it is supported for the component type. This has not worked in a long time, including my last test using LO 6.0.4.2.
UpdateDocMode	The numeric value specifies how the document is updated. See the API site for the constant com.sun.star.document.UpdateDocMode.
URL	Fully qualified URL of the document to load, including the JumpDescriptor if required.
Version	If versioning is supported for the component, this argument indicates the version to load or save. The main document is loaded or saved if no version is specified.
ViewControllerName	While loading a document into a frame, this specifies the name of the view controller to create. That is, the property is passed to the document's createViewController method. If the loaded document does not support the XModel2 interface, the property is ignored.
ViewData	The value for this argument is component specific and usually supplied by the frame controller.
ViewId	Some components support different views of the same data. The value specifies the view to use after loading. The default is zero and is treated as the default view.

12.4.2. Loading a template

While loading a document using `LoadComponentFromUrl()`, the API treats all documents the same. You can open a template for editing (changing a template and then saving it as a new template) or you can use a template as the starting point for a new document. The named argument `AsTemplate` directs LO to open the specified document as a template rather than opening a document for editing. The macro in Listing 263 opens a document as a template.

Listing 263. Load a document as a template.

```
Sub UseTemplate
    REM This is an array from 0 To 0 of type PropertyValue
    Dim args(0) As New com.sun.star.beans.PropertyValue
    Dim sURL As String 'URL of the document to load

    sURL = "file:///home/andy/doc1.ott"
    args(0).Name = "AsTemplate"
    args(0).Value = True
    StarDesktop.LoadComponentFromUrl(sURL, "_blank", 0, args())
End Sub
```

The method `LoadComponentFromUrl()` assumes that a URL is used to specify the document location; this is not operating-system specific. Although I ran the macro in Listing 263 on my Linux computer, it will work on a Windows computer as well. On a Windows computer, the URL usually includes the drive letter.

```
sURL = "file:///c:/home/andy/doc1.ott"
```

TIP Use the functions `ConvertFromURL` and `ConvertToURL` to convert between the operating-system-specific notation and URL notation.

12.4.3. Enabling macros while loading a document

When a document that contains a macro is opened from the user interface (UI), a security dialog opens, asking if macros should be run. When a document is loaded from a macro using the API, macros are disabled in the document. Some macros run based on events that occur in a document. If macros are disabled in the document, you can still manually run the contained macros, but the macros that are started from an event in the document will never run.

The named argument `MacroExecutionMode` directs OOO how to treat macros when a document is loaded. Table 100 enumerates the valid values for the `MacroExecutionMode` argument.

Table 100. *com.sun.star.document.MacroExecMode constants.*

#	Name	Description
0	NEVER_EXECUTE	A macro should not be executed at all.
1	FROM_LIST	Execute macros from secure list quietly.
2	ALWAYS_EXECUTE	Execute any macro, macros signed with trusted certificates and macros from secure list are executed quietly.
3	USE_CONFIG	Use configuration to retrieve macro settings. In case a user confirmation is required a dialog is output.
4	ALWAYS_EXECUTE_NO_WARN	Macros are always executed without confirmation.
5	USE_CONFIG_REJECT_CONFIRMATION	Use configuration to retrieve macro settings. Treat cases when user confirmation required as rejected.
6	USE_CONFIG_APPROVE_CONFIRMATION	Use configuration to retrieve macro settings. Treat cases when user confirmation required as approved.
7	FROM_LIST_NO_WARN	Execute only macros from secure list. Macros that are not from the list are not executed.

#	Name	Description
8	FROM_LIST_AND_SIGNED_WARN	Execute only macros from secure list or macros that are signed by trusted certificates.
9	FROM_LIST_AND_SIGNED_NO_WARN	Execute only macros from secure list or macros that are signed by trusted certificates. No warning/conformation should be shown.

The macro in Listing 264 loads a document as a template and allows macros to run when the document is loaded. This macro also demonstrates that multiple named arguments can be used simultaneously.

Listing 264. *Load a document as a template and enable contained macros.*

```
Sub UseTemplateRunMacro
    REM This is an array from 0 To 1 of type PropertyValue
    Dim args(1) As New com.sun.star.beans.PropertyValue
    Dim sURL As String 'URL of the document to load

    Print com.sun.star.document.MacroExecMode.USE_CONFIG
    sURL = "file:///home/andy/doc1.odt"
    args(0).Name = "AsTemplate"
    args(0).Value = True
    args(1).Name = "MacroExecutionMode"
    args(1).Value = com.sun.star.document.MacroExecMode.ALWAYS_EXECUTE_NO_WARN
    StarDesktop.LoadComponentFromUrl(sURL, "_blank", 0, args())
End Sub
```

12.4.4. Importing and exporting

OpenOffice.org has a type-detection mechanism to determine a document's type when it is loaded. This mechanism has been reliable for the limited number of document types that I deal with daily. Sometimes, however, you must specify the filter name when importing a document. To export a document, you must always specify the export filter type. The code in Listing 265 opens a Microsoft Word document. In my testing the documents opened correctly even when I did not specify the filter name; because it was able to guess the filter.

Listing 265. *Specify the filter name while loading a document.*

```
Sub LoadDocFile
    Dim noArgs(0) As New com.sun.star.beans.PropertyValue
    Dim sURL As String
    noArgs(0).Name = "FilterName"
    noArgs(0).Value = "Microsoft Word 97/2000/XP"
    sURL = "file:///home/andy/one.doc"
    StarDesktop.LoadComponentFromUrl(sURL, "_blank", 0, noArgs())
End Sub
```

12.4.5. Filter names

New filters are frequently added, so it seems silly to list the supported filters, when a macro can retrieve the latest list. The following macro creates a new Calc document and then displays the list of supported filters. A recorded macro is used to format the data.

The UIName is the general name; or localized name. The Name column contains the internal name to use when specifying a filter name for import or export. The Import and Export columns identify if the filter can be used for import or export. The Options column, indicates if options can be passed to the filter. These options direct the filter.

The filter names are also stored in the file TypeDetection.xcu stored in the configuration folders.

Listing 266. List supported filters in a Calc Sheet.

```
sub FiltersToCalc
    Dim oFF          ' FilterFactory service
    Dim oFilterNames ' Array of filter names.
    Dim oDoc          ' Document created to hold the filter names.
    Dim oSheet        ' Sheet that will hold the filter information.
    Dim oCell         ' Cell to update
    Dim i%            ' Index Variable.
    Dim fProps()      ' Properties for each filter.
    Dim fp%           ' Filter property index
    Dim fpCol%        ' Column for the specified filter data.
    Dim s$            ' Work string that holds some property values.
    Dim iUserData%    ' For enumerating array property values.
    Dim oProp         ' A single property.
    Dim oArrayData    ' Array of user properties.
    Dim sPropNames(7) As String
    Dim flags As Long
    sPropNames(0) = "Name"
    sPropNames(1) = "Import"
    sPropNames(2) = "Export"
    sPropNames(3) = "Options"
    sPropNames(4) = "Visible"
    sPropNames(5) = "Flags"
    sPropNames(6) = "DocumentService"
    sPropNames(7) = "UIName"

    oFF = createUnoService( "com.sun.star.document.FilterFactory" )
    oFilterNames = oFF.getElementNames()

    ' Create a Calc doc and save the filter names to it.
    oDoc = StarDesktop.loadComponentFromURL( "private:factory/scalc", "_blank", 0, Array() )
    oSheet = oDoc.getSheets().getByIndex(0)

    ' Print the filter names into a document.
    For i = LBound( oFilterNames ) To UBound( oFilterNames )
        fProps() = oFF.getByName(oFilterNames(i))
        For fp = LBound(fProps) To UBound(fProps)
            oCell = oSheet.getCellByPosition(fp, i+1)
            oProp = fProps(fp)
```

```

fpCol = FindInArrayAdd(oProp.Name, sPropNames)
If fpCol > UBound(sPropNames) Then
    ReDim Preserve sPropNames(fpCol) As String
    sPropNames(fpCol) = oProp.Name
End If

oCell = oSheet.getCellByPosition(fpCol, i+1)
If Not IsArray(oProp.Value) Then
    oCell.setString(oProp.Value)
    If oProp.Name = "Flags" Then
        flags = oProp.Value
        oCell.setString(CStr(flags) & " = " & flags_int2str(flags))
        If flags and 1 Then
            oCell = oSheet.getCellByPosition(FindInArrayAdd("Import", sPropNames), i+1)
            oCell.setString("X")
        Endif
        If flags and 2 Then
            oCell = oSheet.getCellByPosition(FindInArrayAdd("Export", sPropNames), i+1)
            oCell.setString("X")
        Endif
        If flags and 128 Then
            oCell = oSheet.getCellByPosition(FindInArrayAdd("Options", sPropNames), i+1)
            oCell.setString("X")
        Endif
        If ((flags and 4096) or (flags and 8192)) Then
            oCell = oSheet.getCellByPosition(FindInArrayAdd("Visible", sPropNames), i+1)
            oCell.setString("X")
        Endif
    End If
ElseIf LBound(oProp.Value) <= UBound(oProp.Value) Then
    s = ""
    oArrayData = oProp.Value
    For iUserData = LBound(oArrayData) To UBound(oArrayData)
        If VarType(oArrayData(iUserData)) = 8 Then
            s = s & "(String:" & oArrayData(iUserData) & ")"
        Else
            s = s & "(" & oArrayData(iUserData).Name & ":" & oArrayData(iUserData).Value & ")"
        End If
    Next
    oCell.setString(s)
End If
Next
Next
For fp = LBound(sPropNames) To UBound(sPropNames)
    oCell = oSheet.getCellByPosition(fp, 0)
    oCell.setString(sPropNames(fp))
Next
FinalFormat(oDoc)
End Sub

function flags_int2str(flags as long) as string
    Dim cflags as long
    Dim strval as string

```



```

Dim curFlag As Long

cflags = flags
strval = ""

if cflags and &H00000001 then
    strval = strval + " Import"
    cflags = cflags-1
endif
if cflags and &H00000002 then
    strval = strval + " Export"
    cflags = cflags-2
endif
if cflags and &H00000004 then
    strval = strval + " Template"
    cflags = cflags-4
endif
if cflags and &H00000008 then
    strval = strval + " Internal"
    cflags = cflags-8
endif
if cflags and &H00000010 then
    strval = strval + " TemplatePath"
    cflags = cflags-16
endif
if cflags and &H00000020 then
    strval = strval + " Own"
    cflags = cflags-32
endif
if cflags and &H00000040 then
    strval = strval + " Alien"
    cflags = cflags-64
endif
if cflags and &H00000080 then
    strval = strval + " UseOptions"
    cflags = cflags-128
endif
if cflags and &H00000100 then
    strval = strval + " Default"
    cflags = cflags-256
endif
if cflags and &H00000200 then
    strval = strval + " Executable"
    cflags = cflags-512
endif
if cflags and &H00000400 then
    strval = strval + " SupportSelection"
    cflags = cflags-1024
endif
if cflags and &H00000800 then
    strval = strval + " MapToAppPlug"
    cflags = cflags-2048
endif

```

```

if cflags and &H00001000 then
    strval = strval + " NotInFileDialog"
    cflags = cflags-4096
endif
if cflags and &H00002000 then
    strval = strval + " NotInChooser"
    cflags = cflags-8192
endif
if cflags and &H00004000 then
    strval = strval + " Acynchronas"
    cflags = cflags-16384
endif
if cflags and &H00008000 then
    strval = strval + " Creator"
    cflags = cflags-32768
endif
if cflags and &H00010000 then
    strval = strval + " Readonly"
    cflags = cflags-65536
endif
if cflags and &H00020000 then
    strval = strval + " NotInstalled"
    cflags = cflags-131072
endif
if cflags and &H00040000 then
    strval = strval + " ConsultService"
    cflags = cflags-262144
endif
if cflags and &H00080000 then
    strval = strval + " 3rdPartyFilter"
    cflags = cflags-524288
endif
if cflags and &H00100000 then
    strval = strval + " Packed"
    cflags = cflags-1048576
endif
if cflags and &H00200000 then
    strval = strval + " SilentExport"
    cflags = cflags-2097152
endif
if cflags and &H00400000 then
    strval = strval + " BrowserPreferred"
    cflags = cflags-4194304
endif
if cflags and &H00800000 then
    strval = strval + " [H00800000]"
    cflags = cflags-8388608
endif

if cflags and &H01000000 then
    strval = strval + " [H01000000]"
    cflags = cflags-16777216
endif
endif

```

```

    if cflags and &H02000000 then
        strval = strval + " [H02000000]"
        cflags = cflags-33554432
    endif

    if cflags and &H10000000 then
        strval = strval + " Preferred"
        cflags = cflags-268435456
    endif

    if cflags <> 0 then
        strval = strval + " !!! ATTENTION: unsupported flag ["+cflags+"] detected !!!"
    endif

    flags_int2str = strval
end function

Function FindInArrayAdd(sName$, nameArray) As Integer
    Dim i As Integer
    For i = LBound(nameArray()) To UBound(nameArray())
        'Print nameArray(i) & " and " & sName
        If nameArray(i) = sName Then
            FindInArrayAdd = i
            Exit Function
        End If
    Next
    i = UBound(nameArray()) + 1
    FindInArrayAdd = i
End Function

sub FinalFormat(oDoc)
rem -----
rem define variables
Dim document as object
Dim dispatcher as object
rem -----
rem get access to the document
document = oDoc.CurrentController.Frame
dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

rem -----
Dim args1(0) as new com.sun.star.beans.PropertyValue
args1(0).Name = "ToPoint"
args1(0).Value = "$A$1"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args1())

rem -----
Dim args2(1) as new com.sun.star.beans.PropertyValue
args2(0).Name = "By"
args2(0).Value = 1
args2(1).Name = "Sel"

```

```

args2(1).Value = true

dispatcher.executeDispatch(document, ".uno:GoRightToEndOfData", "", 0, args2())

rem -----
Dim args3(2) as new com.sun.star.beans.PropertyValue
args3(0).Name = "FontHeight.Height"
args3(0).Value = 12
args3(1).Name = "FontHeight.Prop"
args3(1).Value = 100
args3(2).Name = "FontHeight.Diff"
args3(2).Value = 0

dispatcher.executeDispatch(document, ".uno:FontHeight", "", 0, args3())

rem -----
Dim args4(0) as new com.sun.star.beans.PropertyValue
args4(0).Name = "Bold"
args4(0).Value = true

dispatcher.executeDispatch(document, ".uno:Bold", "", 0, args4())

rem -----
Dim args5(0) as new com.sun.star.beans.PropertyValue
args5(0).Name = "HorizontalAlignment"
args5(0).Value = com.sun.star.table.CellHoriJustify.CENTER

dispatcher.executeDispatch(document, ".uno:HorizontalAlignment", "", 0, args5())

rem -----
Dim args6(1) as new com.sun.star.beans.PropertyValue
args6(0).Name = "By"
args6(0).Value = 1
args6(1).Name = "Sel"
args6(1).Value = true

dispatcher.executeDispatch(document, ".uno:GoDownToEndOfData", "", 0, args6())

rem -----
Dim args7(0) as new com.sun.star.beans.PropertyValue
args7(0).Name = "aExtraWidth"
args7(0).Value = 254

dispatcher.executeDispatch(document, ".uno:SetOptimalColumnWidth", "", 0, args7())

rem -----
Dim args8(8) as new com.sun.star.beans.PropertyValue
args8(0).Name = "ByRows"
args8(0).Value = true
args8(1).Name = "HasHeader"
args8(1).Value = true
args8(2).Name = "CaseSensitive"
args8(2).Value = false

```

```

args8(3).Name = "IncludeAttribs"
args8(3).Value = true
args8(4).Name = "UserDefIndex"
args8(4).Value = 0
args8(5).Name = "Col1"
args8(5).Value = 7
args8(6).Name = "Ascending1"
args8(6).Value = true
args8(7).Name = "Col2"
args8(7).Value = 1
args8(8).Name = "Ascending2"
args8(8).Value = true

dispatcher.executeDispatch(document, ".uno:DataSort", "", 0, args8())
dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args1())

end sub

```

12.4.6. Loading and saving documents

The desktop or a frame is used to load a document. To store a document, use the method `storeToURL()`, which is implemented by the Document object. The first argument to this method is the URL to which the document will be stored. The second argument is an array of named arguments (see Table 99). To export a document to a different type, the `FilterName` must specify the document export type. The code in Listing 267 exports a Writer document to a PDF file.

Listing 267. *Export the current document, assuming it is a Writer document, as PDF.*

```

Dim args(0) as new com.sun.star.beans.PropertyValue
args(0).Name = "FilterName"
args(0).Value = "writer_pdf_Export"
ThisComponent.storeToURL("file:///test.pdf", args())

```

TIP Filter names are case sensitive.

Some Calc import and export filters support options; for example, DIF, dBase, and Lotus filters all use the same filter option, a numeric index that identifies which character set to use for single-byte characters.

Listing 268. *The DIF, dBase, and Lotus filters all use the same filter option.*

```

Dim args(1) as new com.sun.star.beans.PropertyValue
args(0).Name = "FilterName"
args(0).Value = "dBase"
args(1).Name = "FilterOptions" 'Identify character set for single-byte chars
args(1).Value = 0             'System character set

```

More detailed information on CSV files is found in `AndrewBase.odt`.

The discussion of the CSV filter assumes familiarity with the CSV file format. If you aren't familiar with this format, feel free to skip this section. The Text CSV filter, unlike the dBase filter, uses a

complicated string containing five tokens. Each token can contain multiple values separated by a slash mark (/). The tokens are then separated by a comma (,).

Listing 269. *The CSV filter options are complicated.*

```
Dim args(1) as new com.sun.star.beans.PropertyValue
args(0).Name = "FilterName"
args(0).Value = "scal: Text - txt - csv (StarCalc)"
args(1).Name = "FilterOptions"
args(1).Value = "44,34,0,1,1/5/2/1/3/1/4/1"
```

The first token contains the field separators as ASCII values. For example, to specify a comma as the delimiter, use the ASCII value 44 (see Listing 269). To specify that some fields are separated by a space (ASCII 32) and some are separated by a tab (ASCII 9), Listing 269 would use “32,9,34,0,1,1/5/2/1/3/1/4/1” for the filter options.

In a CSV file, the text portions are usually surrounded by either double quotation marks (") or single quotation marks ('). Listing 269 specifies that text portions should be surrounded by double quotation marks, using an ASCII value of 34. If you want to use multiple text delimiters, separate them with a slash.

The third token identifies the character set to use; this is the same value used to identify the character set in the DIF, dBase, and Lotus filters. The code in Listing 269 specifies a zero for the character set.

The fourth token identifies the first line to import — typically line one. Listing 269 specifies that the first line of text should be input.

The last token identifies the format of each column in the CSV file. The columns can be formatted as either delimited text (see Listing 270) or as fixed-width text (see Listing 271).

Listing 270. *Delimited-text-format string for the CSV filter.*

```
<field_num>/<format>/<field_num>/<format>/<field_num>/<format>/...
```

The <field_num> is an integer that identifies a field, where 1 is the leftmost field. For example, given the fields “one”, “two”, and “three”, a field_num of 2 refers to “two”. The <format> is an integer that identifies the format of the field (see Table 101). The code in Listing 269 specifies that fields one through four use a format of 1, which is the standard format.

Table 101. *CSV field format values.*

Format	Description
1	Standard
2	Text
3	MM/DD/YY
4	DD/MM/YY
5	YY/MM/DD
9	Do not import; ignore this field.
10	Import a number formatted in the US-English locale regardless of the current locale.

Listing 271. Fixed-width-columns format string for the CSV filter.

```
FIX/<start>/<format>/<start>/<format>/<start>/<format>/...
```

Tell the filter that the file uses a fixed-width format by using the text “FIX” as the first portion of the last token. Fixed-width CSV files identify fields by specifying where each field starts (see Listing 271). The <start> value specifies the first character in a field. A start of 0 refers to the leftmost character of the text. The <format> is an integer that identifies the format of the text (see Table 101).

12.4.7. Error handling while loading a document

Exceptions are not thrown while loading a document; instead, errors are thrown using an interaction handler that is passed as a named argument. Unfortunately, you cannot implement an interaction handler in Basic. The Developer’s Guide has examples of error handlers using other languages. In other words, in BASIC, you cannot trap errors while loading a document; the document simply fails to load and returns null.

The Graphical User Interface (GUI) provides an interaction handler that interacts with the user. The GUI’s interaction handler displays error messages when errors occur and prompts the user for a password if it is required. If no handler is included as a named argument, a default handler is used. The default handler simply ignores most errors, providing little feedback to the user.

12.5. Conclusion

The desktop acts as the main application that controls LO, so when you need to obtain something that is globally related to the documents or frames, look at the desktop. The globally available variables `StarDesktop` and `ThisComponent` provide easy access to the Desktop object and the current document. This chapter introduced techniques for accessing containers with multiple objects. Familiarize yourself with the basic capabilities for opening documents, for importing and exporting file types, and the capabilities and limitations of LO, in order to expand your ability to use and create a wide variety of file types within the LO environment.

13. Generic Document Methods

LO supports six primary document types: text, spreadsheet, drawing, math, database, and presentation. The different document types share significant functionality and interfaces, such as accessing the document model, printing, and saving. This chapter introduces common tasks across all of the document types.

Every document contains data that can be manipulated and printed. For example, the primary data in a Writer document is text, but it can also contain tables and graphics. The data model consists of this underlying data that can be manipulated independently of how it is displayed. To manipulate the data by means of a macro, you manipulate the data model directly. Although you can use the dispatcher to indirectly manipulate the data, the changes are still made directly to the model. A typical usage of the dispatcher is to paste the clipboard contents into the document (see Chapter 11 The Dispatcher).

The data model contains a Controller object, which manipulates the visual presentation of the data. The controller doesn't change the data; it only controls how the document is presented. The controller interacts directly with the user interface to control the location of the displayed cursor, to control which page is displayed, and to select portions of the document. Use the controller to determine display-related information such as the current page or the currently selected text.

TIP

LO supports a mode called "headless," which has no startup screen, no default document, no user interface, and supports no user interaction. (Use "soffice -?" to see a list of supported modes.) In headless mode, there is no display component. Therefore, a controller is not required and may not exist. If a macro might be run in headless mode, check the controller to see if it is null before using it.

13.1. Service Manager

LO has a general global service manager, which is used to create and obtain instances of general UNO services. A service manager accepts a string that identifies a type of object, and it returns an instance of that object. The global service manager is made available in Basic through the function `createUnoService(String)`. The global service manager returns general objects such as a dispatch helper or an object for simple file access.

Listing 272. *Use the global service manager.*

```
oDispatcher = createUnoService("com.sun.star.frame.DispatchHelper")
oSimpleFileAccess = createUnoService("com.sun.star.ucb.SimpleFileAccess")
```

Documents are a service manager to create objects that they contain (or that are directly related to the document); for example, a Writer document is able to create a text table or text field that can be inserted into the document. In general, objects that were not created by a document cannot be inserted into the document. A document is unable to return global objects, and the global service manager is unable to return objects that are then inserted into a document.

Listing 273. *Use a document as a service manager.*

```
REM Let the document create the text table.
oTable = oDoc.createInstance( "com.sun.star.text.TextTable" )
```



```
oTable.initialize(3, 2) 'Three rows, two columns
REM Now insert the text table at the end of the document.
oDoc.Text.insertTextContent(oDoc.Text.getEnd(), oTable, False)
```

13.2. Services and interfaces

Think of a component as a window owned by the desktop. Use the Desktop object to enumerate the components, including documents, the Basic IDE and the help window along with the actual documents. If a component supports the `com.sun.star.frame.XModel` interface, the component is a document (rather than the Basic IDE or the help window). Every document supports the `XModel` interface and each document type has a unique service that it supports (see Table 102). Use `HasUnoInterfaces` to determine if an object supports the `XModel` interface; then use the object method `supportsService` to determine the document type.

TIP The method `supportsService()` is defined by the `com.sun.star.lang.XServiceInfo` interface. This interface also defines the method `getImplementationName()`, which returns a unique string identifying the object type.

Table 102. Unique services that identify the document's type.

Service	Document Type
<code>com.sun.star.text.TextDocument</code>	Writer text document.
<code>com.sun.star.sheet.SpreadsheetDocument</code>	Calc spreadsheet document.
<code>com.sun.star.drawing.DrawingDocument</code>	Draw drawing document.
<code>com.sun.star.presentation.PresentationDocument</code>	Impress presentation document.
<code>com.sun.star.formula.FormulaProperties</code>	Math formula document.
<code>com.sun.star.comp.dba.ODatabaseDocument</code>	Base Document

Numerous interfaces are used by many (if not all) of the document types. Consider this partial list of common interfaces.

Interface	Description
<code>com.sun.star.beans.XPropertySet</code>	Get and set object properties.
<code>com.sun.star.container.XChild</code>	For objects with a single parent, get and set that parent.
<code>com.sun.star.datatransfer.XTransferable</code>	Get data for data transfer; like copying to the clipboard.
<code>com.sun.star.document.XDocumentPropertiesSupplier</code>	Access a documents properties; for example, author and creation date.
<code>com.sun.star.document.XDocumentEventBroadcaster</code>	Register to be notified when events occur.
<code>com.sun.star.document.XEventsSupplier</code>	Get a list of events supported by this object.
<code>com.sun.star.document.XLinkTargetSupplier</code>	Get the list of link targets in a document.
<code>com.sun.star.document.XViewDataSupplier</code>	Get properties describing a document's open views.
<code>com.sun.star.drawing.XDrawPagesSupplier</code>	Get draw pages for documents that support multiple draw pages; for example, a drawing or a presentation.
<code>com.sun.star.frame.XLoadable</code>	Functionality to load documents.

com.sun.star.frame.XModel	Representation of a resource (document) in the sense that it was created/loaded from a resource; off hand, I am not aware of any document that does not implement XModel. This interface also provides access to the model's components.
com.sun.star.frame.XStorable	Methods to store a document.
com.sun.star.lang.XComponent	Allows an object to own and dispose (free) an object; for example, a text table.
com.sun.star.lang.XEventListener	Base interface for all listeners; provides a disposing method.
com.sun.star.lang.XMultiServiceFactory	Implemented by factories to create objects and determine what objects the factor can create.
com.sun.star.lang.XServiceInfo	Determine which services an object supports.
com.sun.star.lang.XTypeProvider	Determine the types (interfaces) an object supports.
com.sun.star.script.XStarBasicAccess	Deprecated, but provides access to libraries.
com.sun.star.style.XStyleFamiliesSupplier	Get style families supported by a document; for example, paragraph styles.
com.sun.star.util.XCloseBroadcaster	Allows an object to veto a close request.
com.sun.star.util.XCloseable	Ask a document to close; this is preferred to dispose.
com.sun.star.util.XModifiable	Determine if a document has been modified.
com.sun.star.util.XModifyBroadcaster	Register and be notified when a document is modified.
com.sun.star.util.XNumberFormatsSupplier	Get the document's number formats.
com.sun.star.view.XPrintJobBroadcaster	Register and be notified about print progress.
com.sun.star.view.XPrintable	Document print functionality.
com.sun.star.view.XRenderable	Functionality to render a document.

Ironically, searching is not supported by all document types. This is because searching functionality is very specific to the document. For example, searching a text document is very different than searching a spreadsheet (in terms of desired options). In Calc, searching is provided by the each sheet rather than by the document.

A Calc document is composed of multiple spreadsheets. A significant portion of the functionality, therefore, exists in the spreadsheet objects rather than in the parent Calc document. For example, searching text or obtaining draw pages both exist in spreadsheet objects — draw pages are discussed in depth in Chapter 16 Draw and Impress Documents.

The API Web site contains extensive, detailed help on most of the services and interfaces. The page is built by starting with the Web address “<http://www.openoffice.org/api/docs/common/ref/>” and then using the interface name to build the rest of the address. For example, the com.sun.star.beans.XPropertySet interface has an Internet address as follows:

<http://www.openoffice.org/api/docs/common/ref/com/sun/star/beans/XPropertySet.html>
https://api.libreoffice.org/docs/idl/ref/XPropertySet_8idl.html

13.3. Getting and setting properties

The `dbg_properties` property is a string containing a list of properties supported the object. Basic automatically makes these properties available for direct access; other languages may not. The `com.sun.star.beans.XPropertySet` interface provides methods to get, set, and enumerate the object's properties, as shown in Table 103.

Table 103. Object methods defined by the interface `XPropertySet`.

Object Method	Description
<code>getPropertySetInfo()</code>	Return an object supporting the <code>com.sun.star.beans.XPropertySetInfo</code> interface. This object describes the object's properties but may be null.
<code>setProperty(name, value)</code>	Set the value of the named property. A listener may veto this change.
<code>getProperty(name)</code>	Return the value of the named property.
<code>addPropertyChangeListener(name, listener)</code>	Add an <code>XPropertyChangeListener</code> for the named property. An empty name listens for all properties.
<code>removePropertyChangeListener(name, listener)</code>	Remove an <code>XPropertyChangeListener</code> .
<code>addVetoableChangeListener(name, listener)</code>	Add an <code>XVetoableChangeListener</code> for the named property. An empty name listens for all properties.
<code>removeVetoableChangeListener(name, listener)</code>	Removes an <code>XVetoableChangeListener</code> .

In Basic, properties are usually accessed directly. Listing 274 demonstrates two ways to obtain the `CharFontName` property from a Writer document — these methods return the name of the font style.

Listing 274. Two methods to obtain the name of the font style.

```
Print ThisComponent.CharFontName
Print CStr(ThisComponent.getPropertyValue("CharFontName"))
```

Accessing the property directly is the most expedient method while using Basic, but there are advantages to using the methods defined by the `XPropertySetInfo` interface. Some properties are defined as optional, so not every document contains every property. The `XPropertySetInfo` interface defines the object method `hasPropertyByName()`, which can be used to test for the existence of a property before use; errors can still be avoided by using error-handling routines. Another use is to generically enumerate all of the contained properties and possibly their values as shown in Listing 275. Figure 81 shows some properties of a Writer document using the macro in Listing 275.

Listing 275. Display general document properties.

```
Sub getPropertyValues
    Dim vPropInfo 'PropertySetInfo object
    Dim vProps    'Array of properties
    Dim vProp     'com.sun.star.beans.Property
    Dim v         'Value of a single property
    Dim i%        'Index variable
    Dim s$        'General message string
```

```

Dim nCount%

REM Object implements interface com.sun.star.beans.XPropertySetInfo
vPropInfo = ThisComponent.getPropertySetInfo()
REM sequence< Property > vPropInfo.getProperties()
REM Property vPropInfo.getPropertyByName(name)
REM boolean vPropInfo.hasPropertyByName(name)
vProps = vPropInfo.getProperties()
For i = 0 To UBound(vProps)
    If nCount = 30 Then
        nCount = 0
        MsgBox s, 0, "Properties"
        s = ""
    End If
    nCount = nCount + 1
    vProp = vProps(i) 'com.sun.star.beans.Property
    s = s & vProp.Name & " = "
    v = ThisComponent.getPropertyValue(vProp.Name)
    If IsNull(v) Then
        s = s & "Null"
    ElseIf IsEmpty(v) Then
        s = s & "Empty"
    ElseIf VarType(v) < 9 Then
        s = s & CStr(v)
    Else
        s = s & "Object or Array"
    End If
    s = s & CHR$(10)
Next
MsgBox s, 0, "Properties"
End Sub

```

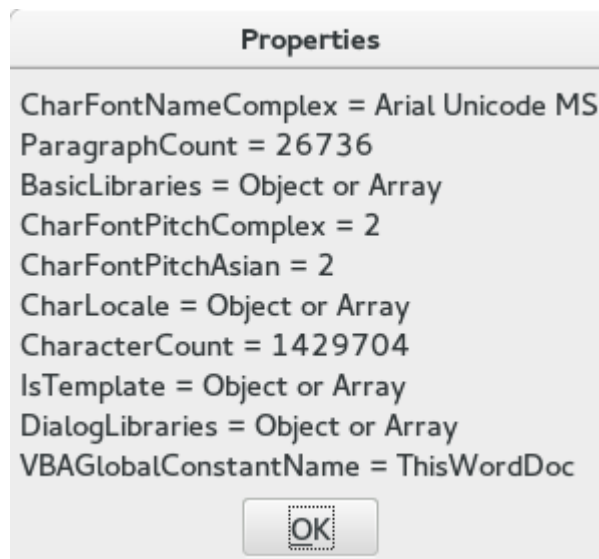


Figure 81. A few properties of ThisComponent.

TIP

In case you missed the implications of the macro in Listing 275, it allows you to generically inspect any object that supports the XPropertySet interface. The ability to inspect an object is invaluable when you aren't certain what you can do with an object.

After getting a property set information object,

- Use `getProperties()` to get an array of Property objects.
- Use `hasProperty(name)` to determine if a property exists.
- Use `getPropertyByName(name)` to get a specific property.

13.4. Document properties

Document properties contain information related to the document such as the author and creation time; as opposed to the properties discussed in the previous section that are related to the document contents such as the current font.

Previous methods for obtaining document properties have been deprecated; for example, using `getDocumentInfo` to obtain and modify user fields. Use **File | Properties** to open the document properties dialog. Use `getDocumentProperties` to obtain the `DocumentProperties` object.

Listing 276. Use the DocumentProperties object.

```
Sub getThisDocProperties()  
    ' com.sun.star.document.DocumentProperties  
    printDocProperties(ThisComponent.getDocumentProperties())  
End Sub  
  
Sub printDocProperties(oDocProps)  
    Dim s$  
    Dim oData  
  
    s = "Author : " & oDocProps.Author & CHR$(10) & _  
        "AutoloadSecs : " & oDocProps.AutoloadSecs & CHR$(10) & _  
        "AutoloadURL : " & oDocProps.AutoloadURL & CHR$(10) & _  
        "DefaultTarget : " & oDocProps.DefaultTarget & CHR$(10) & _  
        "Description : " & oDocProps.Description & CHR$(10) & _  
        "EditingCycles : " & oDocProps.EditingCycles & CHR$(10) & _  
        "EditingDuration : " & _  
            secondsAsPrettyTime(oDocProps.EditingDuration) & CHR$(10) & _  
        "Generator : " & oDocProps.Generator & CHR$(10) & _  
        "ModifiedBy : " & oDocProps.ModifiedBy & CHR$(10) & _  
        "Title : " & oDocProps.Title & CHR$(10) & _  
        "Language : " & oDocProps.Language.Country & ":" & _  
            oDocProps.Language.Language & CHR$(10) & _  
        "ModificationDate : " & oDocProps.ModificationDate.Day & "/" & _  
            oDocProps.ModificationDate.Month & "/" & _  
            oDocProps.ModificationDate.Year & " " & _  
            oDocProps.ModificationDate.Hours & ":" & _  
            oDocProps.ModificationDate.Minutes & ":" & _  
            oDocProps.ModificationDate.Seconds & "." & _  
            oDocProps.ModificationDate.NanoSeconds & CHR$(10) & _
```

```

        "PrintDate : " & oDocProps.PrintDate.Month & "/" & _
            oDocProps.PrintDate.Day & "/" & _
            oDocProps.PrintDate.Year & " " & _
            oDocProps.PrintDate.Hours & ":" & _
            oDocProps.PrintDate.Minutes & ":" & _
            oDocProps.PrintDate.Seconds & "." & _
            oDocProps.PrintDate.NanoSeconds & CHR$(10) & _
        "PrintedBy : " & oDocProps.PrintedBy & CHR$(10) & _
        "Subject : " & oDocProps.Subject & CHR$(10) & _
        "TemplateDate : " & oDocProps.TemplateDate.Month & "/" & _
            oDocProps.TemplateDate.Day & "/" & _
            oDocProps.TemplateDate.Year & " " & _
            oDocProps.TemplateDate.Hours & ":" & _
            oDocProps.TemplateDate.Minutes & ":" & _
            oDocProps.TemplateDate.Seconds & "." & _
            oDocProps.TemplateDate.NanoSeconds & CHR$(10) & _
        "TemplateName : " & oDocProps.TemplateName & CHR$(10) & _
        "TemplateURL : " & oDocProps.TemplateURL & CHR$(10) & _
        "Title : " & oDocProps.Title & CHR$(10)
MsgBox s, 0, "Document Properties"

Dim i%
oData = oDocProps.DocumentStatistics
s = ""
For i = LBound(oData) To UBound(oData)
    s = s & oData(i).Name & " : " & oData(i).Value & CHR$(10)
Next
MsgBox s, 0, "Document Statistics"

oData = oDocProps.Keywords
s = ""
For i = LBound(oData) To UBound(oData)
    s = s & oData(i) & CHR$(10)
Next
MsgBox s, 0, "Keywords"

'I have little experience with this object.
'Seems that the user defined properties are available here.
'com.sun.star.comp.comphelper.OPropertyBag
oData = oDocProps.UserDefinedProperties
End Sub

```

13.4.1. Document properties from a closed document

It is easy to access the document properties from a document that is not open.

Listing 277. *Read the document properties from a document that is not open.*

```

Sub loadExternalProperties
    Dim sPath$
    Dim sPathUrl
    Dim oDocProps
    sPath = ConvertToUrl("/andrew0/home/andy/MoveFigsFromFrames.odt")
    oDocProps = CreateUnoService("com.sun.star.document.DocumentProperties")

```

```

oDocProps.loadFromMedium(sPath, Array())
printDocProperties(oDocProps)
End Sub

```

13.4.2. Custom properties

You can create and remove your own custom document properties. Use the PropertyAttribute constants shown in Table 104 while creating an attribute to control the behavior of created properties.

Table 104. *com.sun.star.beans.PropertyAttribute constants.*

Constant	Value	Description
MAYBEVOID	1	The property value can be void.
BOUND	2	A PropertyChangeEvent will be fired to all registered property change listeners when the value of this property changes.
CONSTRAINED	4	A PropertyChangeEvent will be fired to all registered vetoable change listeners when the value of this property changes.
TRANSIENT	8	The property value is not saved with the document.
READONLY	16	The property value is read-only.
MAYBEAMBIGUOUS	32	The property value may be ambiguous.
MAYBEDEFAULT	64	The property can be set to default.
REMOVEABLE	128	The property can be removed. This used to be called REMOVABLE.
OPTIONAL	256	The property is optional.

Use addProperty(name, attributes, default_value) to create a new property and removeProperty(name) to remove a property from the document properties. When a property is created, that property is added to the document's properties, so you can use the property set information object

Listing 278. *Add a new document property.*

```

Sub AddNewDocumentProperty
    Dim oUDP
    oUDP = ThisComponent.getDocumentProperties().UserDefinedProperties
    If NOT oUDP.getPropertySetInfo().hasPropertyByName("AuthorLastName") Then
        oUDP.addProperty("AuthorLastName", _
            com.sun.star.beans.PropertyAttribute.MAYBEVOID + _
            com.sun.star.beans.PropertyAttribute.REMOVEABLE + _
            com.sun.star.beans.PropertyAttribute.MAYBEDEFAULT, _
            "Default Last Name")
    End If
End Sub

```

13.4.3. Deprecated document information object

The deprecated method getDocumentInfo was the previous method for obtaining and modifying user data. The macro in Listing 279 sets the value of a user field and then displays its value.

Listing 279. *Deprecated method for getting and setting user information data.*

```
Sub GetUserInfoFields
    Dim vDocInfo 'Document Information Object
    Dim s$       'General string to print
    Dim i%       'Index variable
    vDocInfo = ThisComponent.getDocumentInfo()
    vDocInfo.setUserFieldValue(1, "My special user value")
    For i% = 0 To vDocInfo().getUserFieldCount()-1
        s$ = s$ & vDocInfo.getUserFieldName(i) & " = " & _
            CStr(vDocInfo.getUserFieldValue(i)) & CHR$(10)
    Next
    MsgBox s$, 0, "Info Fields"
End Sub
```

13.5. List events

As LO runs, it generates events to inform listeners that something has occurred. The event listener/supplier framework allows a macro to be called or a listener to be notified when a specific event occurs — for example, when a document is loaded or modified, or when text selection changes. Each document type supports the two interfaces `com.sun.star.document.XEventBroadcaster` and `com.sun.star.document.XEventsSupplier`, allowing them to support event-related activities — for example, broadcasting events to listeners and providing a list of supported events.

The macro in Listing 280 lists the event listeners registered with the current document and for LO. Another use for this macro is to list the events supported by a document. Although there are no registered listeners, this macro provides a nice list of the supported event types.

Listing 280. *List the document's events.*

```
Sub DisplayAvailableEvents
    Dim oGEB ' GlobalEventBroadcaster
    Dim oDoc
    Dim s$
    Dim oText

    oDoc = StarDesktop.LoadComponentFromUrl("private:factory/swriter", "_blank", 0, Array())
    oGEB = createUnoservice("com.sun.star.frame.GlobalEventBroadcaster")
    s = Join(oGEB.Events.getElementNames(), CHR$(10))
    oText = oDoc.Text
    oText.InsertString(oText.End, "===Global Events" & CHR$(10), False)
    oText.InsertString(oText.End, s, False)

    s = Join(oDoc.Events.getElementNames(), CHR$(10))
    oText.InsertString(oText.End, CHR$(10) & CHR$(10) & "===Writer Events" & CHR$(10), False)
    oText.InsertString(oText.End, s, False)
End Sub
```

Here is a list of Writer specific events:

1. OnStartApp
2. OnCloseApp
3. OnCreate
4. OnNew
5. OnLoadFinished
6. OnLoad
7. OnPrepareUnload
8. OnUnload
9. OnSave
10. OnSaveDone
11. OnSaveFailed
12. OnSaveAs
13. OnSaveAsDone
14. OnSaveAsFailed
15. OnCopyTo
16. OnCopyToDone
17. OnCopyToFailed
18. OnFocus
19. OnUnfocus
20. OnPrint
21. OnViewCreated
22. OnPrepareViewClosing
23. OnViewClosed
24. OnModifyChanged
25. OnTitleChanged
26. OnVisAreaChanged
27. OnModeChanged
28. OnStorageChanged

29. OnPageCountChange

30. OnMailMerge

31. OnMailMergeFinished

32. OnFieldMerge

33. OnFieldMergeFinished

34. OnLayoutFinished

TIP

Many objects besides documents support event listeners. For example, you can listen to an individual cell in a Calc document.

13.5.1. Registering your own listener

The macro in Listing 280 uses the GlobalEventBroadcaster to access event listeners at the LO level. The macro in Listing 281 replaces the documents OnSave listener with a macro named MySave, in Module1, in the Standard library for My Macros. When the document is saved, MySave is automatically called. The association is maintained even after closing and opening the document.

Listing 281. Register your own listener.

```
Sub StealAnEvent
    Dim mEventProps(1) as new com.sun.star.beans.PropertyValue
    mEventProps(0).Name = "EventType"
    mEventProps(0).Value = "StarBasic"
    mEventProps(1).Name = "Script"
    mEventProps(1).Value = "macro:///Standard.Module1.MySave()"

    'oGlobalEventBroadcaster = createUnoservice("com.sun.star.frame.GlobalEventBroadcaster")
    'oGlobalEventBroadcaster.Events.ReplaceByName("OnStartApp", mEventProps())
    ThisComponent.Events.ReplaceByName("OnSave", mEventProps())
End Sub
```

Warn

Use the location “macro://hello/Standard.Module1.MySave()” to specify a macro in the document named “hello.odt”. Although this works fine, the association is not lost when the document is closed.

An event can just as easily be stored at the OOO level by using the GlobalEventBroadcaster.

Do not mistake the value location for the same URL type syntax used to associate a form control to a macro:

```
"vnd.sun.star.script:Standard.Module1.LocalMySave?language=Basic&location=document"
```

13.5.2. Intercepting dispatch commands

Replacing a listener event is not the same as intercepting the **File > Save** dispatch, this is merely a listener used after the command has been executed. Intercepting a dispatch will look more like

this code provided by Paolo Montavoni. One of the problems with this is that a status listener is not used, so, if a command is disabled, the menu will not show the command as disabled.

Warn Be very careful while writing listeners and intercepting dispatches. Assume that something will change, or that you did something wrong and that LO might crash; you have been warned.

Listing 282. Replace a few menu commands.

```
Global oDispatchInterceptor
Global oSlaveDispatchProvider
Global oMasterDispatchProvider
Global oFrame
Global bDebug As Boolean

Dim oCopyDispatch

Sub RegisterInterceptor
    oFrame = ThisComponent.currentController.Frame
    oDispatchInterceptor = CreateUnoListener("ThisFrame_", _
        "com.sun.star.frame.XDispatchProviderInterceptor")
    oFrame.registerDispatchProviderInterceptor(oDispatchInterceptor)
End Sub

Sub ReleaseInterceptor()
On Error Resume Next
    oFrame.releaseDispatchProviderInterceptor(oDispatchInterceptor)
End Sub

Function ThisFrame_queryDispatch ( oUrl, _
    sTargetFrameName As String, lSearchFlags As Long ) As Variant

Dim oDisp
Dim sUrl As String

'slot protocol causes OOo crash...
if oUrl.Protocol = "slot:" Then
    Exit Function
End If

If bDebug Then
    Print oUrl.complete
End If

'do your dispatch management here:
Select Case oUrl.complete

    Case ".uno:Copy"
        oDisp = GetCopyDispatch 'replace the original dispatch
    Case ".uno:Paste"
        oDisp = GetCopyDispatch 'replace the original dispatch
    Case ".uno:Save"
        oDisp = GetCopyDispatch 'replace the original dispatch
    Case ".uno:Undo"
```

```

        oDisp = GetCopyDispatch 'replace the original dispatch
    'Case ".uno:blabla"
        'do something
    Case Else
        oDisp = _
        oSlaveDispatchProvider.queryDispatch( oUrl, sTargetFrameName, lSearchFlags )

End Select

ThisFrame_queryDispatch = oDisp

End Function

Function ThisFrame_queryDispatches ( mDispatches ) As Variant
    'ThisFrame_queryDispatches = mDispatches()
End Function

Function ThisFrame_getSlaveDispatchProvider ( ) As Variant
    ThisFrame_getSlaveDispatchProvider = oSlaveDispatchProvider
End Function

Sub ThisFrame_setSlaveDispatchProvider ( oSDP )
    oSlaveDispatchProvider = oSDP
End Sub

Function ThisFrame_getMasterDispatchProvider ( ) As Variant
    ThisFrame_getMasterDispatchProvider = oMasterDispatchProvider
End Function

Sub ThisFrame_setMasterDispatchProvider ( oMDP )
    oMasterDispatchProvider = oMDP
End Sub

Sub toggleDebug
    bDebug = Not bDebug
End Sub

Function GetCopyDispatch()
    If Not IsNull(oCopyDispatch) Then
        oCopyDispatch = _
        CreateUnoListener("MyCustom_", "com.sun.star.frame.XDispatch")
    End If

    GetCopyDispatch = oCopyDispatch

End Function

Sub MyCustom_dispatch(URL, Arguments)
    Select Case URL.complete
    Case ".uno:Copy"
        MsgBox "Sorry, the original " & URL.complete & _
            " dispatch was stolen from Paolo M.", 48
    End Select
End Sub

```

```

Case ".uno:Paste"
    ThisComponent.CurrentSelection(0).String = _
        "**** ARBITRARY CLIPBOARD CONTENT PROVIDED BY PAOLO M. ****"
Case ".uno:Save"
    MsgBox "Sorry, the original " & URL.complete & _
        " dispatch was stolen from Paolo M.", 48

Case ".uno:Undo"
    MsgBox "Undo What????!!!???", 16

Case Else

End Select
End Sub

Sub MyCustom_addStatusListener(Control, URL)
End Sub

Sub MyCustom_removeStatusListener (Control, URL)
End Sub

```

A similar method has been used to intercept context menus using a `XContextMenuInterceptor`.

13.6. Link targets

Link targets, also called “jump marks,” can be used to jump directly to a specific location. The document navigator contains a list of link targets. The object method `getLinks()`, defined by the `com.sun.star.document.XLinkTargetSupplier` interface, provides access to the link targets. The `getLinks()` method returns an object that supports the `XNameAccess` interface. In other words, you can access the links using the methods `getByName()`, `getElementNames()`, `hasByName()`, and `hasElements()`.

The object returned by the `getLinks()` method does not access the links directly, but rather it provides access to other objects that can. To access all of the individual links, first use the `getLinks()` method to access the master list of links. Use this returned object to access each “family” of links based on the family name. For example, to obtain an object that can access all of the table links, use:

```
oDoc.getLinks().getByName("Tables")
```

After obtaining a family of links, you can also obtain the individual links by name. At this final step, you have access to both the name of the link and the object that is linked. The macro in Listing 283 obtains all of the link families and the links that they contain, and then displays the link’s name in a newly created Write document.

Listing 283. *Get jump targets for the current document.*

```

Sub GetJumpTargets
    Dim sLinkNames 'Tables, Text Frames, Headings, Bookmarks, etc...
    Dim vOneLink   'One link type
    Dim i%         'Index variable
    Dim s$         'General string

```

```

Dim vtemp
Dim oNewDoc
Dim noArgs()           'An empty array for the arguments
Dim vComp              'The loaded component
Dim sURLs()            'URLs of the new document types to load
Dim sURL As String     'URL of the document to load
Dim oText

sURL = "private:factory/swriter"
vComp = StarDesktop.LoadComponentFromUrl(sURL, "_blank", 0, noArgs())
oText = vComp.Text
oText.InsertString(oText.End, "LINK TYPES" & CHR$(10), False)
sLinkNames = ThisComponent.getLinks().getElementNames()
oText.InsertString(oText.End, Join(sLinkNames, Chr$(10)), False)
oText.InsertString(oText.End, _
    CHR$(10) & CHR$(10) & "JUMP TARGETS" & CHR$(10), _
    False)
For i = 0 To UBound(sLinkNames)
    vOneLink = ThisComponent.getLinks().getByName(sLinkNames(i))
    s = s & sLinkNames(i) & " = "
    If IsEmpty(vOneLink) Then
        s = s & "Empty"
    Else
        s = s & sLinkNames(i) & " : " & _
            Join(vOneLink.getElementNames(), CHR$(10) & sLinkNames(i) & " : " )
        REM To obtain the actual link object, such as a
        REM text table or a graphics object, use the following
        REM vtemp = vOneLink.getElementNames()
        REM vObj = vOneLink.getByName(vtemp(0))
    End If
    s = s & CHR$(10)
Next
'MsgBox s, 0, "Jump Targets"
oText.InsertString(oText.End, s, False)
End Sub

```

You can use jump marks (link targets) to jump directly to a specific location when a document is loaded. Use jump marks to focus the cursor at a specific location when a document is opened, as shown in Listing 284. The `JumpMark` attribute sets the name from the values shown in the created document (if you run the macro above); for example, “Table1|table” jumps to the specified table.

Listing 284. *Use the `JumpMark` property to jump to a link target.*

```

Dim Props(0)
Props(0).Name = "JumpMark"
Props(0).Value = "Table1|table"
sUrl = "file:///c:/docs/Special_doc.odt"
vDoc = StarDesktop.LoadComponentFromUrl(sUrl, "_blank", 0, Props())

```

You can also use the jump mark as part of the URL (see Listing 285) by placing it at the end of the URL separated by an octothorpe character (#). If the jump mark contains any special characters, such as spaces, they must be encoded using standard URL notation. For example, a space is encoded as %20.

Listing 285. Use the *JumpMark* property to jump to a link target.

```
sUrl = "file:///c:/docs/Special_doc.odt#Table1|table"  
vDoc = StarDesktop.LoadComponentFromUrl(sUrl, "_blank", 0, Props())
```

TIP The character “#” has many names, including: number sign, pound sign, hash, sharp, crunch, hex, grid, pigpen, tic-tac-toe, splat, crosshatch, and octothorpe, to name a few.

13.7. Accessing view data: *XViewDataSupplier*

Using OOo, open a document, edit it or change something (for example, move to another page or change the zoom factor), close the document, and open the document again. When you do this, the document opens at the same screen location, at the same size, with the same zoom factor as was active when it was last saved. This information is stored with the document and is available through the interface `com.sun.star.document.XViewDataSupplier`. This interface provides one object method, `getViewData()`. Listing 286 displays the view data for `ThisComponent` (see Figure 82).

Listing 286. *getViewData* is found in the

```
Sub GetViewData  
    Dim vViewData 'View data object  
    Dim i%        'Index variable  
    Dim j%        'Index variable  
    Dim s$        'General string  
    Dim vtemp     'View data for one object  
    vViewData = ThisComponent.getViewData()  
    REM For each view of the data  
    For i = 0 To vViewData.getCount() - 1  
        vtemp = vViewData.getByIndex(i)  
        For j = 0 To UBound(vtemp)  
            s = s & vtemp(j).Name & " = " & CStr(vtemp(j).Value) & CHR$(10)  
        Next  
        MsgBox s, 0, "View Data"  
    Next  
End Sub
```

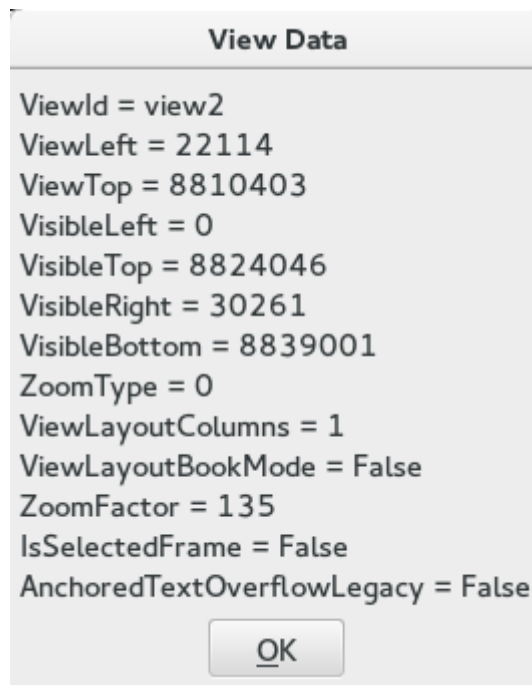


Figure 82. View data for a document.

13.8. Close a document: XCloseable

As of OOo 1.1.0, each document type supports the `com.sun.star.util.XCloseable` interface. To close these objects, call the object method `close(bForce)`. If `bForce` is `True`, the object must close. In other words, it may not refuse to close. If, however, `bForce` is `False`, the object may refuse to close.

Before a document is closed, a message is sent to all registered listeners, giving them an opportunity to prevent the closure. If the close request is not vetoed by any listener, a message is sent to each registered listener, advising them that the document will close. The `XCloseBroadcaster` interface provides methods to add and remove a close listener (see Table 105).

Table 105. Object methods defined by the interface XCloseBroadcaster.

Object Method	Description
<code>addCloseListener(XCloseListener)</code>	Add listener to receive or have a veto for “close” events.
<code>removeCloseListener(XCloseListener)</code>	Remove an object registered as a close listener using <code>addCloseListener()</code> .

The macro in Listing 287 demonstrates the safe way to close a document using the object’s `close()` method. For versions of OOo prior to 1.1.0, the `close()` method is not available, so you must use the `dispose()` method instead. The `dispose()` method is unconditional and is not the preferred method to close a document (by document I really mean any object that supports both `close` and `dispose`) because it doesn’t allow a registered user who might be using the document to veto the close and finish what he or she is doing.

Listing 287. *The safe way to close a document.*

```
If HasUnoInterfaces(oDoc, "com.sun.star.util.XCloseable") Then
    oDoc.close(true)
Else
    oDoc.dispose()
End If
```

TIP

Do not use dispose to close a document, it is there for legacy purposes. Assume, for example, that you start printing a document and then immediately dispose() the document. The document that is being used for printing is suddenly gone and LO crashes.

13.9. Draw Pages: XDrawPagesSupplier

13.9.1. Draw and Impress

Draw and Impress are almost identical in the interfaces that they support. Draw is specifically designed to handle independent graphic objects, whereas Impress is designed for business effects and presentations. The drawing functionality of Draw and Impress is identical, however. The graphic objects are drawn and displayed on “draw pages.” By their design, both Draw and Impress support multiple draw pages. The functionality to retrieve a DrawPage object is defined by the com.sun.star.drawing.XDrawPagesSupplier interface. The interface com.sun.star.drawing.XDrawPages defines methods to retrieve, insert, and remove individual pages (see Table 106).

Table 106. Object methods defined by the interface XDrawPages.

Object Method	Description
InsertNewByIndex(index)	Create and insert a new draw page or master page.
remove(XDrawPage)	Remove a draw page or master page.
getCount()	Return the number of draw pages.
getByIndex(index)	Return a specific draw page.
hasElements()	Return True if there are documents.

The macro in Listing 288 demonstrates how to iterate through each of the draw pages. Each draw page is exported to a JPG file. The export type is specified by the MediaType property.

Listing 288. *Export each graphics page to a JPG.*

```
oFilter=CreateUnoService("com.sun.star.drawing.GraphicExportFilter")
Dim args(1) as new com.sun.star.beans.PropertyValue
For i=0 to oDoc.getDrawPages().getcount()-1
    oPage = oDoc.getDrawPages().getByIndex(i)
    name = oPage.name
    oFilter.setSourceDocument(opage)
    args(0).Name = "URL"
    args(0).Value = "file:///c|/"&oPage.name&".JPG"
    args(1).Name = "MediaType"
```

```
args(1).Value = "image/jpeg"
oFilter.filter(args())
Next
```

TIP

The index used to access draw pages is zero based. This means that the first draw page is at location 0. If a document contains four draw pages, they are numbered 0 through 3. This is why the For loop in Listing 288 is from 0 To oDoc.getDrawPages().getcount()-1.

The macro in Listing 289 creates a new Impress document and then adds a graphic image to the first draw page. The draw image is sized to retain the aspect ratio.

Listing 289. *Add a proportionally sized graphic to a draw page.*

```
Sub AddProportionalGraphic
    Dim oDoc          'Newly created Impress document
    Dim oDrawPage     'The draw page that will contain the graphic image
    Dim oGraph        'The created graphic image

    REM Create an Impress presentation document!
    oDoc = StarDesktop.loadComponentFromURL("private:factory/simpres", _
                                           "_default", 0, Array())

    REM Insert a second draw page if desired,
    REM leaving the first draw page untouched!
    REM Could use the property DrawPages
    REM oDrawPage = oDoc.DrawPages.insertNewByIndex(1)
    REM oDrawPage = oDoc.getDrawPages().insertNewByIndex(1)
    REM In this case, simply use the first draw page!
    oDrawPage = oDoc.getDrawPages().getByIndex(0)

    REM Create a graphics object that can be inserted into the document
    oGraph = oDoc.createInstance("com.sun.star.drawing.GraphicObjectShape")

    REM Set the URL of the image so that it can be added to the document
    oGraph.GraphicURL = "http://www.openoffice.org/images/AOO_logos/orb.jpg"
    oDrawPage.add(oGraph)

    REM If I stop here, there will be a very small graphic image in the
    REM upper-left corner of the document. This is pretty much useless.
    REM Although I could simply size the graphic to the same size as the bitmap
    REM size, I choose instead to size this so that it is as large as possible
    REM without changing the aspect ratio.
    REM Determine the ratio of the height to the width for the image and
    REM also for the draw page.

    Dim oNewSize As New com.sun.star.awt.Size    'New Image size
    Dim oBitmapSize As New com.sun.star.awt.Size 'Bitmap size

    Dim dImageRatio As Double    'Ratio of the height to width
    Dim dPageRatio As Double     'Ratio of the height to width

    oBitmapSize = oGraph.GraphicObjectFillBitmap.GetSize
    dImageRatio = Cdbl(oBitmapSize.Height) / Cdbl(oBitmapSize.Width)
```

```

dPageRatio = CDb1(oDrawPage.Height) / CDb1(oDrawPage.Width)

REM Compare the ratios to see which is wider, relatively speaking
If dPageRatio > dImageRatio Then
    oNewSize.Width = oDrawPage.Width
    oNewSize.Height = CLng(CDb1(oDrawPage.Width) * dImageRatio)
Else
    oNewSize.Width = CLng(CDb1(oDrawPage.Height) / dImageRatio)
    oNewSize.Height = oDrawPage.Height
End If

REM Center the image on the Impress page!
Dim oPosition as new com.sun.star.awt.Point
oPosition.X = (oDrawPage.Width - oNewSize.Width)/2
oPosition.Y = (oDrawPage.Height - oNewSize.Height)/2

oGraph.SetSize(oNewSize)
oGraph.SetPosition(oPosition)
End Sub

```

As already stated, Impress and Draw documents are very similar in the API that they support. The macro in Listing 290 draws lines in a Draw document (see Figure 83).

Listing 290. Draw lines in a new graphic document.

```

Sub DrawLinesInDrawDocument
    Dim oDoc          'Newly created Draw document
    Dim oDrawPage     'The draw page that will contain the graphics image
    Dim oShape        'Shape to insert

    REM Create a new Draw document!
    oDoc = StarDesktop.loadComponentFromURL("private:factory/sdraw", _
                                           "_default", 0, Array())

    REM Use the first draw page
    oDrawPage = oDoc.getDrawPages().getByIndex(0)

    Dim i As Long
    Dim oPos as new com.sun.star.awt.Point
    Dim oSize as new com.sun.star.awt.Size
    Dim dStepSize As Double
    dStepSize = CDb1(oDrawPage.Height) / 10

    For i = 0 To 10
        oShape = oDoc.createInstance("com.sun.star.drawing.LineShape")
        oShape.LineColor = rgb(0, 255 - 20 * i, 20 * i)
        oShape.LineWidth = 250

        oPos.X = 0
        oPos.Y = CLng(CDb1(i) * dStepSize)
        oShape.setPosition(oPos)

        oSize.width = oDrawPage.Width
        oSize.height = oDrawPage.Height - 2 * oPos.Y
    Next i
End Sub

```

```

oShape.setSize(oSize)
oDrawPage.add(oShape)
Next
End Sub

```

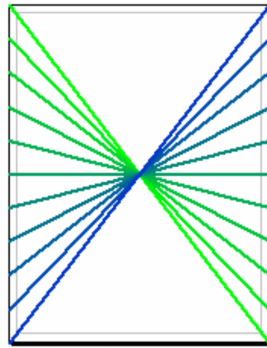


Figure 83. These lines drawn in a Draw document overlap.

13.9.2. Draw lines with arrows in Calc

Each Calc document contains a single draw page for each sheet. In Calc, each sheet is like a transparent layer containing drawing data on top of the standard document data. The following macro demonstrates a couple of useful things:

- Each cell has a position on the draw page. This macro draws a line between specific cells by using the Position attribute on a cell. The line is drawn from the upper left corner of cell B2 to the upper left corner of cell D4.
- A LineShape is created by the document.
- After creation, attributes are set on the LineShape; for example, position, size, and color.
- The LineShape is added to the draw page.
- Arrows, and other line endings, are added *after* the LineShape has been added to the draw page – at least this is required in OOo version 3.3.0.

Arrow line endings are set by setting the LineEndName and LineStartName. I was not able to find a list of supported arrow names, so, I used the GUI to find the names mentioned in the GUI, and the few that I tried worked. Next, I searched the source code and I found `filter/source/msfilter/escherex.cxx` and `svx/source/dialog/sdstring.src`, which contains the following names:

- Arrow
- Arrow concave
- Circle
- Dimension Lines

- Double Arrow
- Line Arrow
- Rounded large Arrow
- Rounded short Arrow
- Small Arrow
- Square
- Square 45
- Symmetric Arrow

In testing, the en-US names worked in other locales. Locale specific names work, but the value that is stored is based on the en-US locale name. In other words, using "Double flèche" with a French locale sets the property to "Double Arrow".

Listing 291. Draw lines in a Calc document.

```
Sub InsertLineInCalcDocument
    Dim oLine
    Dim oCell1
    Dim oCell2
    Dim oSheet

    Dim oPos as new com.sun.star.awt.Point
    Dim oSize as new com.sun.star.awt.Size
    Dim oPage

    oSheet = ThisComponent.Sheets(0)
    oCell1 = oSheet.getCellByPosition(1, 1)
    oCell2 = oSheet.getCellByPosition(3, 3)

    oLine = ThisComponent.createInstance("com.sun.star.drawing.LineShape")

    oPos.x = oCell1.Position.X
    oPos.y = oCell1.Position.Y
    oLine.Position = oPos

    oSize.Width = oCell2.Position.X - oCell1.Position.X
    oSize.Height = oCell2.Position.Y - oCell1.Position.Y
    oLine.Size = oSize

    oLine.LineWidth = 4
    oLine.LineColor = RGB(128, 0, 0)
    oPage = oSheet.getDrawPage()

    oPage.add(oLine)

    REM You must do this AFTER inserting the line into the page.
    oLine.LineEndName = "Arrow"
```

```
oLine.LineStartName = "Double Arrow"
End Sub
```

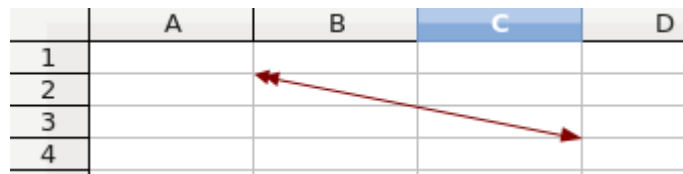


Figure 84. Draw a line in a Calc document.

13.9.3. Writer

Each Writer document contains a single draw page for the entire document. In Writer, the page is like a transparent layer containing drawing data on top of the standard document data.

Writer documents do not support the `XDrawPagesSupplier` interface, because they only contain a single draw page. They do, however, support the `XDrawPageSupplier` interface, which defines the single object method `getDrawPage()`.

The macro in Listing 290 uses optional draw page properties — namely height and width. The draw page from a Writer document does not contain these properties. The draw page in a Writer document has other peculiarities, however. For example, adding lines to the draw page — as done in Listing 290 — adds them as characters at the cursor position rather than interpreting the positions as specific locations in the document. The macro in Listing 292 draws lines to demonstrate this behavior (also see Figure 85).

Listing 292. Draw lines in a Write document.

```
Sub DrawLinesInWriteDocument
    Dim oDoc          'Newly created Writer document
    Dim oDrawPage     'The draw page that will contain the graphics image
    Dim oShape        'Shape to insert

    REM Create a new Writer document!
    oDoc = StarDesktop.loadComponentFromURL("private:factory/swriter", _
                                           "_default", 0, Array())

    oDrawPage = oDoc.getDrawPage()

    Dim i As Long
    Dim oSize as new com.sun.star.awt.Size
    Dim dStepSize As Double
    dStepSize = 800

    For i = 0 To 10
        oShape = oDoc.createInstance("com.sun.star.drawing.LineShape")
        oShape.LineColor = rgb(255, 255 - 20 * i, 20 * i)
        oShape.LineWidth = 50

        oSize.width = dStepSize - CLng(CDbl(i) * dStepSize / 10) / 2
        oSize.width = CLng(dStepSize / 5 * i - dStepSize)
```

```

oSize.height= dStepSize
oShape.setSize(oSize)
oDrawPage.add(oShape)
Next
End Sub

```

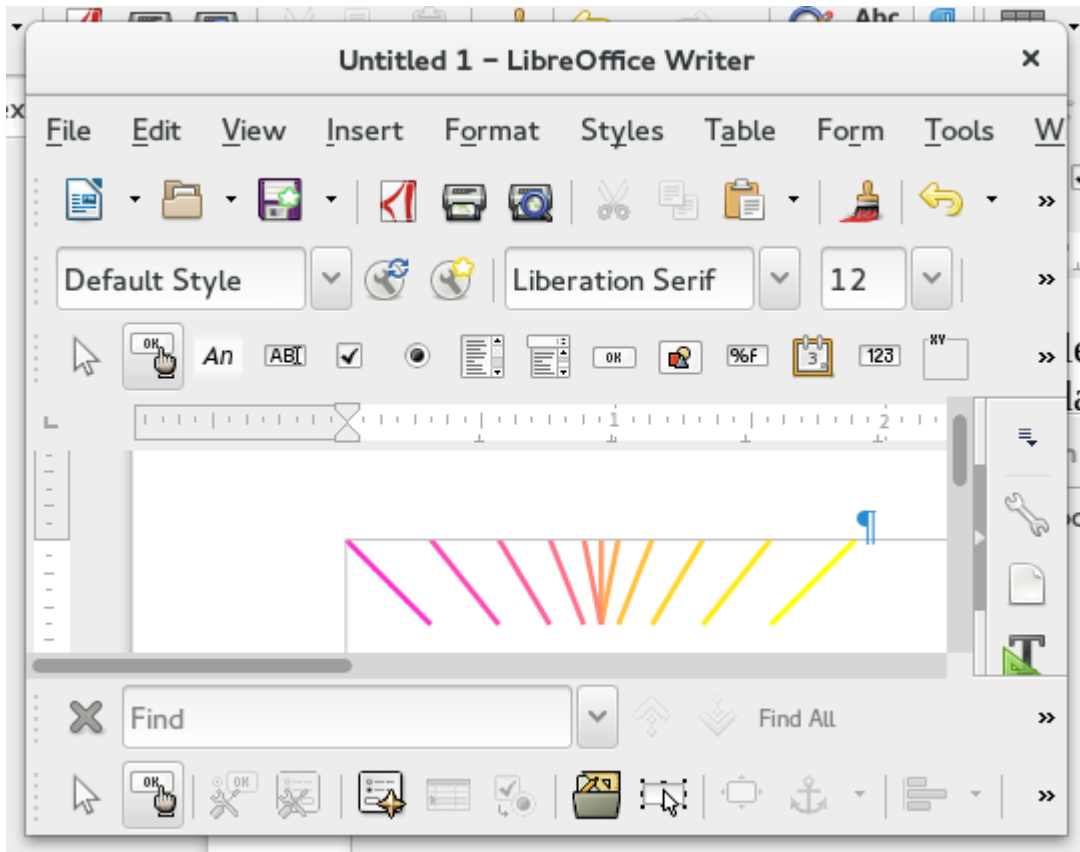


Figure 85. These lines drawn in a Writer document are treated as characters.

13.10. The model

XModel is the primary interface that distinguishes a component as a document, as opposed to the Basic IDE or the included help pages. Objects that implement the interface `com.sun.star.frame.XModel` represent a component created from a URL. OOo document objects may be controlled by a controller, which is also considered a view of the document. The XModel interface defines the object methods in Table 107.

Table 107. Object methods defined by the interface *com.sun.star.frame.XModel*.

Object Method	Description
getURL()	URL of the document returned as a String.
getArgs()	Return a copy of the <i>com.sun.star.document.MediaDescriptor</i> for this model (document).
lockControllers()	Prevent some display updates — macros may run faster. Be sure to unlock the controllers when you are finished.
unlockControllers()	Call this once for each call to <i>lockControllers()</i> .
hasControllersLocked()	Is there at least one lock remaining?
getCurrentController()	The controller that currently controls this model.
getCurrentSelection()	Current selection on the current controller.

13.10.1. Document arguments

Arguments passed to the document loader control how the document is loaded; for example, opening a document as read-only. Use *getArgs()* to obtain the documents media descriptor, which details how the document was loaded. The *MediaDescriptor* service can be accessed as a series of optional properties or as an array of properties (see Listing 293 and Figure 86).

Listing 293. *Print the document media descriptor.*

```
Sub printDocumentArgs()
    REM This says to ignore any lines that cause an error.
    REM Obtaining a value may cause an error, but that is
    REM not a problem. It just prevents a value from being printed.
    On Error Resume Next

    Dim vArgs 'Media descriptor as an array of com.sun.star.beans.PropertyValue
    Dim s$    'Display string
    Dim i%    'Index variable

    REM Obtain the media descriptor. It turns out that this
    REM can be represented as an array of PropertyValue services.
    vArgs = ThisComponent.getArgs()

    For i = 0 To UBound(vArgs)
        s = s & vArgs(i).Name & " = "
        s = s & vArgs(i).Value
        s = s & CHR$(10)
    Next
    MsgBox s, 0, "Args"
End Sub
```

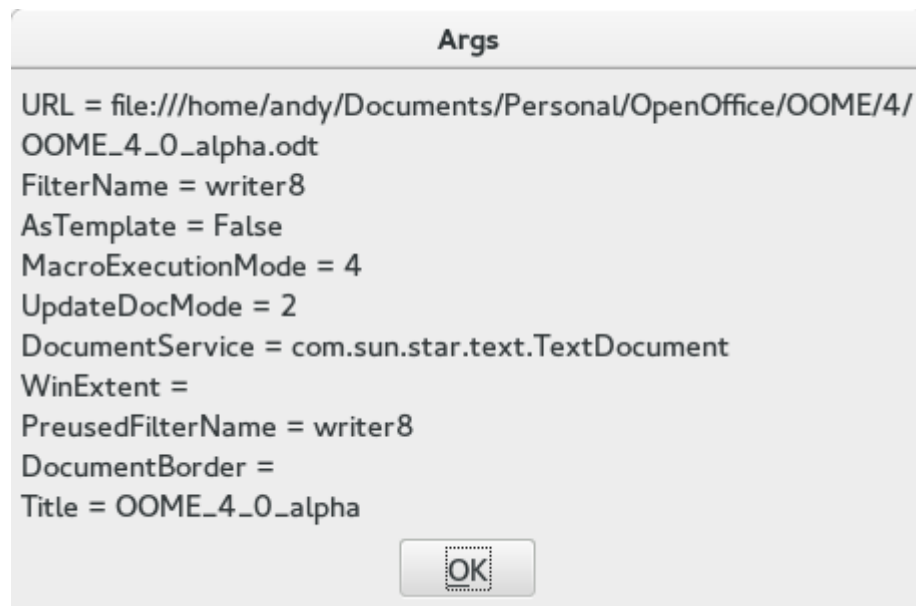



Figure 86. Properties returned by the `getArgs()` object method.

A text file is not a standard Writer file, but rather, a simple file that has typically been created by an editor such as Notepad. Text files contain no information that identify the character set that was used, or how lines are terminated. Text files usually use the file extension TXT. When a text file is opened, LO displays a dialog and asks questions about the file so that it knows how the text file is encoded. I was asked how to load a text file using a macro, avoiding the dialog and explicitly specifying the import values. Although I was able to correctly determine the import filter name, I had no idea how to specify the required filter options. I had no idea, that is, until I found the `getArgs()` object method.

TIP

If you know how to open a document using the GUI, but are uncertain how the arguments must be set to load the document using `loadComponentFromURL`, then first load the document using the GUI, then inspect the document's media descriptor. Hint: look at `FilterOptions`.

Figure 86 shows the arguments for an imported text file. The `FilterName` property indicates the name of the import filter, and the `FilterOptions` property indicates the filter options that were used to open the file. Using these properties with the desktop object method `LoadComponentFromUrl()`, the file is correctly imported and the dialog is not displayed. Table 108 contains a list of the properties that may be returned from the `getArgs()` object method. The OOo API Web site contains a few more properties, but they are obscure or deprecated. ??? We already discussed the `MediaDescriptor`

Table 108. Object properties defined by the service `MediaDescriptor`.

Property	Description
Aborted	May be set if document load is aborted while entering a password.
AsTemplate	Document was loaded as a template.

Property	Description
Author	Author of this document version; this is for versioning.
CharacterSet	Document character set for single-byte characters.
Comment	Comment on the current document version; this is for versioning.
DocumentTitle	If the document title is specified, it is included here.
FilterName	Filter used to import or save this document.
FilterOptions	Filter used to import this document.
FilterData	Additional import properties if the FilterOptions string is not sufficient.
Hidden	If the Hidden argument is specified during load, it is included here.
HierarchicalDocumentName	The hierarchical path to the embedded document from topmost container.
InputStream	If the InputStream is specified during load, it is included here.
InteractionHandler	Exception handler if an error occurs during import.
JumpMark	Jump to this marked position after loading the document.
MediaType	MIME type of this document.
OpenNewView	Opens a new view for an already loaded document, rather than just opening the document twice. In other words, request two views of the same data.
OutputStream	Stream to use while writing the document.
Overwrite	Overwrite any existing file on save.
Password	Password for loading or storing the document.
Preview	Document loaded in preview mode; optimizes for preview-only use.
ReadOnly	Document opened as read-only; controller will not change the document.
Referer	URL of document referrer — for example, if opened by clicking an HTTP link.
RepairPackage	Open the document in repair mode.
StartPresentation	Immediately after loading an Impress document, start the presentation.
StatusIndicator	If a status indicator was specified when the document was loaded, it is included here.
Unpacked	If True, an OOo document is stored as a folder rather than a ZIP file.
URL	URL of the document.
Version	Current document version, if versioning is supported.
ViewData	The view data to use.
ViewId	The ID of the initial view.
MacroExecutionMode	Specifies how macros are treated when the document is loaded.

13.11. Saving a document

The location where a document is saved is called its Uniform Resource Locator (URL) — in other words, its file name. A file URL usually contains the complete path to the file. When the file name is referred to as the URL, it has a form similar to “file:///c:/myfile.sxw” rather than “c:\myfile.sxw”. A URL is a general way of writing a storage location, one that can be extended conveniently to include a wide range of storage location types in a manufacturer-independent and computer-independent manner. Basic supports the functions ConvertToURL and ConvertFromURL to

convert between the two notations. The XStorable interface defines object methods to save a document to a URL (see Table 109).

Table 109. Object methods defined by the service *com.sun.star.frame.XStorable*.

Object Method	Description
hasLocation()	True if the document has a storage location and False if this is a new blank document.
getLocation()	Return the URL where the object was stored after calling storeAsURL().
isReadOnly()	You cannot call store() if the file was called from a read-only location.
store()	Stores the data to the current URL.
storeAsURL(URL, args)	Stores the document to the specified URL, which becomes the current URL.
storeToURL(URL, args)	Stores the document to the specified URL, but the current URL does not change.

Use the object method `hasLocation()` to determine if a document knows where to save itself, and use the method `store()` to save it to the current URL. The macro in Listing 294 uses methods defined by both `XStorable` and `XModifiable` to save a document to disk. The document is stored only if it knows where to store itself, it has changed, and it is not read-only.

Listing 294. Proper method to save a document.

```
If (ThisComponent.isModified()) Then
  If (ThisComponent.hasLocation() AND (Not ThisComponent.isReadOnly())) Then
    ThisComponent.store()
  Else
    REM Either the document does not have a location or you cannot
    REM save the document because the location is read-only.
    setModified(False)
  End If
End If
```

A document does not have a storage location immediately after it is created. A document loaded from disk, however, has a known location. Use the object method `storeAsURL()` or `storeToURL()` to save a document to a specified location. The difference between the methods is that `storeAsURL()` sets the current location (URL) and `storeToURL()` does not. The sequence of actions in Table 110 helps clarify the difference.

Table 110. Difference between *storeToURL* and *storeAsURL*.

Step	Action	Comment
1	Create document	Cannot use the <code>store()</code> method because the document has no location.
2	Use <code>storeToURL</code>	Document saved, but cannot use the <code>store()</code> method because it has no location.
3	Use <code>storeAsURL</code>	Can use the <code>store()</code> method because now the document has a location.
4	Use <code>storeToURL</code>	Document saved, but the location is the same as set in step 3.

TIP	The method <code>storeAsURL()</code> is similar to the menu option File Save As, which changes the current location. The method <code>storeToURL()</code> is usually used to export a document so that the file URL does not change and contain a non-OOo document extension.

The two object methods for storing a document, `storeAsURL()` and `storeToURL()`, accept the same arguments; learn to use one and you'll know how to use the other.

```
ThisComponent.storeAsURL(url, args())  
ThisComponent.storeToURL(url, args())
```

The second argument is an array of property values (see Table 108) that direct how the document is saved (see Listing 295). The files can just as easily be stored with no arguments (see Listing 296).

Listing 295. *Save a document to a new location.*

```
Dim args(0) As New com.sun.star.beans.PropertyValue  
Dim sUrl As String  
sUrl = "file:///c:/My%20Documents/test_file.sxw"  
args(0).Name = "Overwrite" 'This property is defined in Table 108  
args(0).Value = False      'Do not overwrite an existing document.  
ThisComponent.storeAsURL(sUrl, args())
```

TIP

The `com.sun.star.frame.XComponentLoader` interface defines the object method `LoadComponentFromUrl()`, which is used to load a file. The different document types do not implement this interface, but the document frame and the desktop both implement the interface. The method `LoadComponentFromUrl()` also uses the values in Table 108 to direct how the file is loaded.

Listing 296. *Save the document with an inappropriate file extension.*

```
ThisComponent.storeToURL("file:///c:/two.xls", Array())
```

TIP

The macro in Listing 296 uses the file extension “xls”, which is typically used by Microsoft Excel. This does not cause the file to be stored using the Microsoft Excel format. The file is saved using the standard OOo file format if an export filter is not explicitly stated.

When you open a file, OOo checks to see if the file is in a standard OOo file format. If not, the file type is determined based on the file extension. I cannot even count the number of times that I have been asked why OOo is not able to open a comma-delimited text file. The usual answer is that a comma-delimited file must have the file extension CSV or OOo cannot recognize the file. Although the file extension is important while loading a file from the GUI, it is not important while saving. If you want to save a file in a non-OpenOffice.org native format, you must explicitly tell OOo to save in a different file format (see Listing 297).

Listing 297. *Export a document to the specified Microsoft Excel file format.*

```
Dim args(0) as new com.sun.star.beans.PropertyValue  
args(0).Name = "FilterName" 'I am so excited, which filter will we use?  
args(0).Value = "MS Excel 97" 'Oh, the Excel 97 format!  
ThisComponent.storeToURL("file:///c:/one.xls", args())
```

TIP

Although the export filter names are the same as the import filter names, not every import filter can export and not every export filter can import.

Impress and Draw, used to edit graphical content, support multiple draw pages. To export a draw page to a specific graphics format requires the use of a graphics export filter (see Listing 298). The media type must be specified or the export will fail.

Listing 298. *Export the first draw page to a JPG file.*

```
Dim oFilter
Dim args(1) as new com.sun.star.beans.PropertyValue
oFilter=CreateUnoService("com.sun.star.drawing.GraphicExportFilter")
oFilter.SetSourceDocument(ThisComponent.DrawPages(0))
args(0).Name = "URL" 'Where the file will be saved
args(0).Value = "file:///c:/one.JPG" 'The destination URL
args(1).Name = "MediaType" 'What type of file
args(1).Value = "image/jpeg" 'The file type
oFilter.filter(args())
```

13.12. Manipulating styles

Styles provide a method of grouping formatting information. For example, a paragraph style defines the font, character size, margins, and many other formatting options. Changing a style changes every object using the style. The interface `com.sun.star.style.XStyleFamiliesSupplier` provides access to the styles used by a document. The macro in Listing 299 displays the names of all styles in the current document; Figure 87 shows the results for one of my documents.

Listing 299. *Display styles used in a document.*

```
Sub DisplayAllStyles
    Dim oFamilies 'Families is interface com.sun.star.container.XNameAccess
    Dim oFamilyNames 'Names of family types. Array of string
    Dim oStyleNames 'Names of styles. Array of string
    Dim oStyles 'Styles is interface com.sun.star.container.XNameAccess
    Dim oStyle 'An individual style
    Dim s As String 'Utility string variable
    Dim n As Integer 'Index variable
    Dim i As Integer 'Index variable

    oFamilies = ThisComponent.StyleFamilies
    oFamilyNames = oFamilies.getElementNames()

    REM First, display the style types and the number
    REM of each style type.
    For n = LBound(oFamilyNames) To UBound(oFamilyNames)
        oStyles = oFamilies.getByIndex(oFamilyNames(n))
        s = s & oStyles.getCount() & " " & oFamilyNames(n) & CHR$(10)
    Next
    MsgBox s, 0, "Style Families"

    REM Now, display all of the different style names
    For n = LBound(oFamilyNames) To UBound(oFamilyNames)
        s = ""
        oStyles = oFamilies.getByIndex(oFamilyNames(n))
        oStyleNames = oStyles.getElementNames()
        For i = LBound(oStyleNames) To UBound(oStyleNames)
```

```

s=s + i + " : " + oStyleNames(i) + CHR$(10)
If ((i + 1) Mod 30 = 0) Then
    MsgBox s,0,oFamilyNames(n)
    s = ""
End If
Next i
If Len(s) <> 0 Then MsgBox s,0,oFamilyNames(n)
Next n
End Sub

```



Figure 87. The style families in one of my Writer documents.

The different document types contain different types of styles. Figure 87 shows the style families in a Writer document. Calc documents contain the style families CellStyles and PageStyles; Impress documents contain the style families Graphics and Default; Draw documents contain the style family Graphic. Although each style type is different, they do have similarities. For example, each style implements both the `com.sun.star.style.Style` service and the `com.sun.star.style.XStyle` interface. The common methods and properties provide very rudimentary functionality (see Table 111).

Table 111. Object methods defined in the `com.sun.star.style.Style` service.

Method or Property	Description
<code>isUserDefined()</code>	Is this style user-defined? If not, it is included with OOo.
<code>isInUse()</code>	Is this style used in the document?
<code>getParentStyle()</code>	What is the parent style?
<code>setParentStyle(name)</code>	Set the parent style.
<code>IsPhysical</code>	Is the style physically created?
<code>FollowStyle</code>	Style name applied to the next paragraph. For example, while using a heading style I might want the next paragraph to be regular text.
<code>DisplayName</code>	Name of the style as displayed in the user interface.
<code>IsAutoUpdate</code>	If the properties of an object using this style are changed (for example, if I change the font), are these changes automatically updated to the style?

Table 111 shows methods and properties that can be used to answer the common question, “How do I obtain a list of styles that are currently used by a document?” See Listing 300 and Figure 88.

TIP

Listing 299 accesses styles by name; Listing 300 accesses styles by index. Notice that when `getCount()` returns 10, there are 10 items present, indexed 0 through 9.

Listing 300. Display all used paragraph styles.

```
Sub DisplayAllUsedParagraphStyles
    Dim oStyles          'Styles is interface com.sun.star.container.XNameAccess
    Dim oStyle           'An individual style
    Dim s As String      'Utility string variable
    Dim i As Integer     'Index variable

    oStyles = ThisComponent.StyleFamilies.getByName("ParagraphStyles")

    REM If getCount() says that there are 10 styles, this means from 0 to 9
    For i = 1 To oStyles.getCount()
        oStyle = oStyles.getByIndex(i-1)
        If oStyle.isInUse() Then s = s & oStyle.DisplayName & CHR$(10)
    Next
    MsgBox s, 0, "Paragraph Styles In Use"
End Sub
```

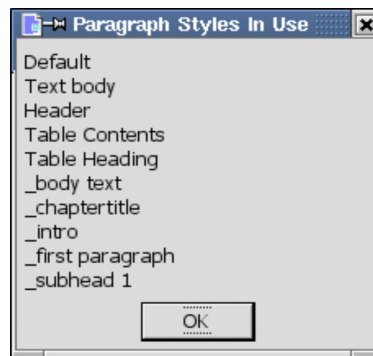


Figure 88. Paragraph styles used in a Writer document.

Initially, I was confused by the results shown in Figure 88 because it included paragraph styles that I thought I wasn't using. I assumed that I had made a mistake while writing my document, so I started looking for these accidentally used styles. To find the incorrectly styled text, I started with the Find & Replace dialog (Edit | Find & Replace). If you check the box labeled "Search for Styles," all of the styles used in the document become available in the "Search for" drop-down box. I used this method to search for the style "Text body" (shown in Figure 88), but it was not found in the document. After some moments of confusion — in this case, roughly five minutes — I realized that I had not found a bug in OpenOffice.org, but rather I had uncovered an interesting behavior of which I had not been previously aware. When a style is used in a document, the parent style is listed as used even if it is not directly used. For example, in my document, the style "Table Contents" uses "Text body" as the parent style.

The different style types contain methods and properties appropriate for their type. See <http://www.openoffice.org/api/docs/common/ref/com/sun/star/style/module-ix.html> for the

common services, interfaces, structs, and constants related to styles. The types shown in the style module are the base on which the other styles are built. For example, the two services `com.sun.star.text.TextPageStyle` and `com.sun.star.sheet.TablePageStyle` both have the service `com.sun.star.style.PageStyle` in common. To get a feel for a style it is frequently expedient to start by inspecting the object.

```
MsgBox vObj.dbg_methods
MsgBox vObj.dbg_supportedInterfaces
MsgBox vObj.dbg_properties
```

The style objects, like many other objects, implement the `XPropertySet` interface. The macro in Listing 301 uses this interface to display a list of properties contained in the “_body text” paragraph style. Listing 301 does not display the value of each property; it only displays the name of each property. It is an interesting exercise to modify Listing 301 to also display the value of each property that is a standard data type — a property may be a complex object.

Listing 301. *Display Properties for a paragraph style.*

```
Sub StyleProperties
    Dim oStyles          'Styles is interface com.sun.star.container.XNameAccess
    Dim s As String      'Utility string variable
    Dim i As Integer     'Index variable
    Dim Props            'Array of properties

    REM Each style supports the com.sun.star.beans.XPropertySet interface
    REM This supports the method getPropertySetInfo(), which allows
    REM an enumeration of the contained properties.
    REM Get properties returns an array of com.sun.star.beans.Property
    oStyles = ThisComponent.StyleFamilies.getByName("ParagraphStyles")
    Props = oStyles.getByName("_body text").getPropertySetInfo().getProperties()

    For i = 0 To UBound(Props)          'For each property
        s = s & Props(i).Name & CHR$(10) 'Include the property name and new line
        If (i+1) MOD 30 = 0 Then         'If the string becomes too large
            MsgBox s,0,"Style Properties" 'display the current string
            s = ""                       'Reset the list
        End If
    Next
    REM In case the entire list wasnot printed.
    If Len(s) <> 0 Then MsgBox s,0,"Style Properties"
End Sub
```

In my experience, it’s rare to access and inspect styles. It’s even less common to modify a style from a macro. There are times, however, when this is done. For example, the page size is defined by the current page style. Use the current controller to determine the current page style, and then use that to determine the page dimensions. Listing 302 displays the size of the page, the margins, and the current cursor position on the page. Figure 89 shows the results.

TIP The most common reason that I have seen for modifying a page style is to add a header or footer to the current page style.

Portions of this code are speculative and have changed since I originally created the code. The current controller returns the current cursor position based on the leftmost corner of the first page in the document. On page 200, therefore, the Y coordinate may be very large. The following code attempts to compensate for this, but, it does so poorly – where poorly means that I believe the answer is incorrect with respect to the top and the bottom of the page.

Listing 302. PrintPageInformation

```
Sub PrintPageInformation
    Dim oViewCursor      'Current view cursor
    Dim oStyle           'Current page style
    Dim lHeight As Long  'Page height from Page Style in 1/100 mm
    Dim lWidth  As Long  'Page width  from Page Style in 1/100 mm
    Dim s As String      'Temporary string variable
    REM The current controller interfaces with the human – that's you!
    REM Well, we hope you are human anyway.
    REM Obtain the current view cursor from the controller. It is
    REM the thing that knows where the current cursor is, after all.
    oViewCursor = ThisComponent.CurrentController.getViewCursor()

    REM That view cursor knows a lot of things, including the
    REM current page style. Use the page style name to get
    REM a reference to the current page style.
    s = oViewCursor.PageStyleName
    oStyle = ThisComponent.StyleFamilies.getByName("PageStyles").getByName(s)
    s = "Page Style = " & s & CHR$(10)

    lHeight = oStyle.Height  'Page height in 1/100 mm
    lWidth  = oStyle.Width   'Page width  in 1/100 mm

    REM Page dimensions in mm, inches, and picas.
    s = s & "Page size is " & CHR$(10) & _
        "      " & CStr(lWidth / 100.0) & " mm By " & _
        "      " & CStr(lHeight / 100.0) & " mm" & CHR$(10) & _
        "      " & CStr(lWidth / 2540.0) & " inches By " & _
        "      " & CStr(lHeight / 2540.0) & " inches" & CHR$(10) & _
        "      " & CStr(lWidth * 72.0 / 2540.0) & " picas By " & _
        "      " & CStr(lHeight * 72.0 / 2540.0) & " picas" & CHR$(10)

    Dim dCharHeight As Double 'Character height in inches
    Dim iCurPage As Integer  'Current page

    Dim dXCursor As Double    'Distance from cursor to left in inches
    Dim dYCursor As Double    'Distance from cursor to top in inches
    Dim dXRight  As Double    'Distance from cursor to right in inches
    Dim dYBottom As Double    'Distance from cursor to bottom in inches
    Dim dBottom  As Double    'Bottom margin in inches
    Dim dLeft    As Double    'Left margin in inches
    Dim dRight   As Double    'Right margin in inches
    Dim dTop     As Double    'Top margin in inches

    dCharHeight = oViewCursor.CharHeight / 72.0 'Convert points to inches
    iCurPage = oViewCursor.getPage()           'Page number
```

```

s = s & "Current page = " & iCurPage & CHR$(10)

dBottom = oStyle.BottomMargin / 2540.0 : dLeft = oStyle.LeftMargin / 2540.0
dRight = oStyle.RightMargin / 2540.0 : dTop = oStyle.TopMargin / 2540.0
s = s & "Margin (inches): Left= " & dLeft & " Right= " & dRight & CHR$(10)
s = s & "Margin (inches): Top= " & dTop & " Bottom= " & dBottom & CHR$(10)

Dim v
REM Cursor's coordinates relative to the top left position of the page
REM Return type is com.sun.star.awt.Point
REM Units are in twips. Convert them to inches.
v = oViewCursor.getPosition()

REM Page number from the view cursor includes "phantom" pages because
REM two pages in a row must both be on say a right side.
REM v.Y does not include this.
Dim realPageNumber As Long
realPageNumber = Fix(v.Y / lHeight)

Dim realY : realY = v.Y - realPageNumber * lHeight

REM Place the cursor as the distance to the margin and then add the margin.
REM For the vertical height add half the character height. I should
REM probably do this for the character width as well for horizontal.
dYCursor = realY/2540.0 + dTop + dCharHeight / 2
dYBottom = (lHeight - realY)/2540.0 - dTop - dCharHeight / 2
dXCursor = v.X/2540.0 + dLeft
dXRight = (lWidth - v.X)/2540.0 - dLeft

s=s &"Cursor is "&Format(dXCursor, "0.##") & " inches from left "&CHR$(10)
s=s &"Cursor is "&Format(dXRight, "0.##") & " inches from right "&CHR$(10)
s=s &"Cursor is "&Format(dYCursor, "0.##") & " inches from top "&CHR$(10)
s=s &"Cursor is "&Format(dYBottom, "0.##") & " inches from bottom "&CHR$(10)
s=s &"Char height = " & Format(dCharHeight, "0.#####") & " inches"&CHR$(10)
MsgBox s, 0, "Page information"
End Sub

```

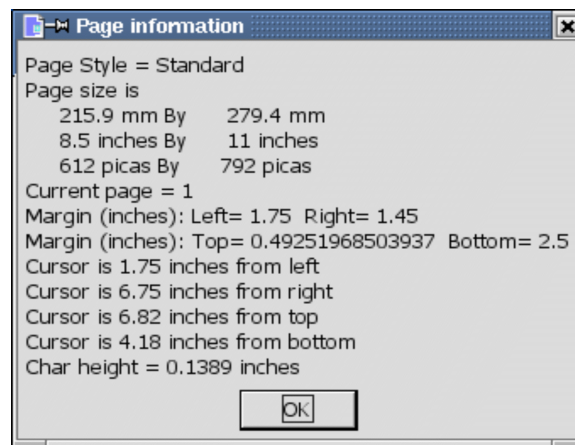


Figure 89. Information from the page style.

Calc documents are composed of spreadsheets. Each sheet can use a different page style. The macro in Listing 302 obtains the current page style using the view cursor. To obtain the style from a Calc document, use the active sheet.

```
REM Use the currently active sheet to obtain the page style.
REM In a Calc document, the current controller knows which sheet
REM is active.
Print "Style = " & ThisComponent.CurrentController.getActiveSheet().PageStyle
```

13.12.1. Style utilities

Although modifying styles using macros is easy, small details can cause big problems; for example, specifying a paragraph style is not available on the current computer.

Listing 303. Check a document for a paragraph style.

```
Function DocHasParStyle(oDoc, sName$) As Boolean
    Dim oStyles
    oStyles = oDoc.StyleFamilies.getByName("ParagraphStyles")
    DocHasParStyle() = oStyles.hasByName(sName)
End Function
```

Checking for the character style is equally trivial.

Listing 304. Check a document for a character style.

```
Function DocHasCharStyle(oDoc, sName$) As Boolean
    Dim oStyles
    oStyles = oDoc.StyleFamilies.getByName("CharacterStyles")
    DocHasCharStyle() = oStyles.hasByName(sName)
End Function
```

To determine if a document supports a specific font, check the font descriptors available from the container window.

Listing 305. Check a document for a font.

```
Function DocHasFontName(oDoc, sName$) As Boolean
    Dim oWindow
    Dim oFonts()
    Dim i%

    oWindow = oDoc.getCurrentController().getFrame().getContainerWindow()
    oFonts() = oWindow.getFontDescriptors()
    For i = LBound(oFonts()) To UBound(oFonts())
        If oFonts(i).Name = sName Then
            DocHasFontName() = True
            Exit Function
        End If
    Next
    DocHasFontName() = False
End Function
```

A property is a structure with a name and a value. The following macro accepts a name and a value and returns a property with the name and value.

Listing 306. Create a property with the specified name and value.

```
*****
'*** Create and return a PropertyValue structure.
*****
Function CreateProperty( Optional cName As String, Optional uValue ) As
com.sun.star.beans.PropertyValue
    Dim oPropertyValue As New com.sun.star.beans.PropertyValue
    If Not IsMissing( cName ) Then
        oPropertyValue.Name = cName
    EndIf
    If Not IsMissing( uValue ) Then
        oPropertyValue.Value = uValue
    EndIf
    CreateProperty() = oPropertyValue
End Function
```

Listing 306 allows properties to be created directly in an array.

Listing 307. Creating properties used to create a character style.

```
REM Base style for all.
REM computer code that is not color coded and used in regular text
REM uses this style.
oProps() = Array(CreateProperty("CharFontName", sFontName), _
    CreateProperty("CharColor", RGB(0, 0, 0)), _
    CreateProperty("CharNoHyphenation", True) )

CreateCharacterStyle("OOoComputerCode", oProps())

REM Base style for normal listings.
oProps() = Array(CreateProperty("ParentStyle", "OOoComputerCode"))

CreateCharacterStyle("_OOoComputerBase", oProps())
```

The difficult part is determining what properties to set and what properties to not set. It is a useful exercise to inspect a manually created style containing the values of interest. A property that is not specifically set inherits its value from the parent style or existing format; for example, setting a character style to bold but not specifying the font or font size, produces a character style that will not affect font or font size and only set the font to be bold.

Listing 308. Character styles used to format code examples.

```
*****
'*** Create character styles for StarBasic using the same colors
'*** as the OOo IDE.
*****
Function CreateStarBasicCharStyles()
    Dim oProps()

    REM If you do not want something to have a language, which prevents
    REM a spell check, set CharLocale to noLocale.
    Dim noLocale As New com.sun.star.lang.Locale
    noLocale.Country = ""
    noLocale.Language = "zxx"
```

```

If NOT CreateBaseCharStyles() Then
    CreateStarBasicCharStyles() = False
    Exit Function
End If

oProps() = Array(CreateProperty("ParentStyle", "_OOoComputerBase"), _
    CreateProperty("CharColor", RGB(76, 76, 76)))
CreateCharacterStyle("_OOoComputerComment", oProps())

oProps() = Array(CreateProperty("ParentStyle", "_OOoComputerBase"), _
    CreateProperty("CharColor", RGB(255, 0, 0)))
CreateCharacterStyle("_OOoComputerLiteral", oProps())

oProps() = Array(CreateProperty("ParentStyle", "_OOoComputerBase"), _
    CreateProperty("CharLocale", noLocale), _
    CreateProperty("CharColor", RGB(0, 0, 128)))
CreateCharacterStyle("_OOoComputerKeyWord", oProps())

oProps() = Array(CreateProperty("ParentStyle", "_OOoComputerBase"), _
    CreateProperty("CharColor", RGB(0, 128, 0)))
CreateCharacterStyle("_OOoComputerIdent", oProps())
CreateStarBasicCharStyles() = True
End Function

```

The following macro creates a character style if it does not exist. A special check is performed to verify that the parent style exists; the parent style must be created before the child style.

Listing 309. Create a character style if it does not exist.

```

Sub CreateCharacterStyle(sStyleName$, oProps())
    Dim i%
    Dim oFamilies
    Dim oStyle
    Dim oStyles

    oFamilies = ThisComponent.StyleFamilies
    oStyles = oFamilies.getByName("CharacterStyles")
    If oStyles.HasByName(sStyleName) Then
        'PrintColor(oStyles.getByName(sStyleName).CharColor)
        Exit Sub
    End If
    oStyle = ThisComponent.createInstance("com.sun.star.style.CharacterStyle")
    For i=LBound(oProps) To UBound(oProps)
        If oProps(i).Name = "ParentStyle" Then

            If oStyles.HasByName(oProps(i).Value) Then
                oStyle.ParentStyle = oProps(i).Value
            Else
                Print "Parent character style (" & oProps(i).Value & _
                    ") does not exist, ignoring parent."
            End If
            oStyle.ParentStyle = oProps(i).Value
        Else

```

```

        oStyle.setPropertyValue(oProps(i).Name, oProps(i).Value)
    End If
Next
oStyles.insertByName(sStyleName, oStyle)
End Sub

```

A paragraph style is a bit more complicated because a few elements require special consideration; for example, setting tab stops.

Listing 310. Create a paragraph style if it does not exist.

```

Sub CreateParStyle(sStyleName$, oProps())
    Dim i%, j%
    Dim oFamilies
    Dim oStyle
    Dim oStyles
    Dim tabStops%

    oFamilies = ThisComponent.StyleFamilies
    oStyles = oFamilies.getByName("ParagraphStyles")
    If oStyles.HasByName(sStyleName) Then
        Exit Sub
    End If
    oStyle = ThisComponent.createInstance("com.sun.star.style.ParagraphStyle")
    For i=LBound(oProps) To UBound(oProps)
        If oProps(i).Name = "ParentStyle" Then
            If oStyles.HasByName(oProps(i).Value) Then
                oStyle.ParentStyle = oProps(i).Value
            Else
                Print "Parent paragraph style (" & oProps(i).Value & _
                    ") does not exist, ignoring parent"
            End If
        ElseIf oProps(i).Name = "ParaTabStops" Then
            tabStops = oProps(i).Value
            Dim tab(0 To 19) As New com.sun.star.style.TabStop
            For j =LBound(tab) To UBound(tab)
                tab(j).Alignment = com.sun.star.style.TabAlign.LEFT
                tab(j).DecimalChar = ASC(".")
                tab(j).FillChar = 32
                tab(j).Position = (j+1) * tabStops
            Next
            oStyle.ParaTabStops = tab
        ElseIf oProps(i).Name = "FollowStyle" Then
            If oStyles.HasByName(oProps(i).Value) OR oProps(i).Value = sStyleName Then
                oStyle.setPropertyValue(oProps(i).Name, oProps(i).Value)
            Else
                Print "Next paragraph style (" & oProps(i).Value & _
                    ") does not exist, ignoring for style " & sStyleName
            End If
        Else
            oStyle.setPropertyValue(oProps(i).Name, oProps(i).Value)
        End If
    Next
    oStyles.insertByName(sStyleName, oStyle)
End Sub

```

End Sub

This code creates the primary paragraph styles used to format code styles in this document.

Listing 311. Properties to create a paragraph style.

```
REM Tab stops are set in the paragraph style
' 1/4 of an inch
tabStopLoc% = 2540 / 4

oProps() = Array(CreateProperty("ParaTopMargin", CLng(0)), _
    CreateProperty("ParaBottomMargin", CLng(2540 * 0.03)), _
    CreateProperty("ParaLeftMargin", CLng(2540 * 0.20)), _
    CreateProperty("ParaRightMargin", CLng(0)), _
    CreateProperty("ParaFirstLineIndent", CLng(0)), _
    CreateProperty("CharFontName", sFontName), _
    CreateProperty("ParaTabStops", tabStopLoc), _
    CreateProperty("ParaLineNumberCount", False), _
    CreateProperty("WritingMode", com.sun.star.text.WritingMode.LR_TB), _
    CreateProperty("CharAutoKerning", False), _
    CreateProperty("CharHeight", fParNormalCharHeight) )
CreateParStyle("_OOoComputerCode", oProps())

oProps() = Array(CreateProperty("ParentStyle", "_OOoComputerCode"), _
    CreateProperty("ParaTopMargin", CLng(0)), _
    CreateProperty("ParaBottomMargin", CLng(2540 * 0.10)), _
    CreateProperty("ParaLeftMargin", CLng(2540 * 0.20)), _
    CreateProperty("ParaRightMargin", CLng(0)), _
    CreateProperty("ParaFirstLineIndent", CLng(0)), _
    CreateProperty("CharFontName", sFontName), _
    CreateProperty("ParaTabStops", tabStopLoc), _
    CreateProperty("ParaLineNumberCount", False), _
    CreateProperty("WritingMode", com.sun.star.text.WritingMode.LR_TB), _
    CreateProperty("CharAutoKerning", False), _
    CreateProperty("CharHeight", fParNormalCharHeight), _
    CreateProperty("FollowStyle", sNextStyle) )
CreateParStyle("_OOoComputerCodeLastLine", oProps())
```

13.13. Dealing with locale

A locale represents a specific geographical, political, or cultural region. Numbers and dates are considered locale-sensitive, so number formats are associated with a locale. Use Tools | Options | Language Settings | Languages to see which locale is used for your computer. Creating a locale is very easy.

```
Dim aLocale As New com.sun.star.lang.Locale
aLocale.Language = "fr"
aLocale.Country = "FR"
```

TIP OOo may not support every possible locale, but it will attempt to use the best possible match.

The locale depends upon both the language and the country. Some countries use multiple languages and some languages are used in multiple countries. Table 112 contains the two-

character code that identifies each language, and Table 113 contains the two-character code that identifies each country.

TIP Although the locale codes are not case sensitive, they are typically written in lowercase for the language and uppercase for the country.

Table 112. Locale language codes, alphabetized by code.

Code	Language	Code	Language	Code	Language
aa	Afar	ab	Abkhazian	af	Afrikaans
am	Amharic	ar	Arabic	as	Assamese
ay	Aymara	az	Azerbaijani	ba	Bashkir
be	Byelorussian	bg	Bulgarian	bh	Bihari
bi	Bislama	bn	Bengali; Bangla	bo	Tibetan
br	Breton	ca	Catalan	co	Corsican
cs	Czech	cy	Welsh	da	Danish
de	German	dz	Bhutani	el	Greek
en	English	eo	Esperanto	es	Spanish
et	Estonian	eu	Basque	fa	Persian
fi	Finnish	fj	Fiji	fo	Faroese
fr	French	fy	Frisian	ga	Irish
gd	Scots Gaelic	gl	Galician	gn	Guarani
gu	Gujarati	ha	Hausa	he	Hebrew (formerly iw)
hi	Hindi	hr	Croatian	hu	Hungarian
hy	Armenian	ia	Interlingua	id	Indonesian (formerly in)
ie	Interlingue	ik	Inupiak	is	Icelandic
it	Italian	iu	Inuktitut	ja	Japanese
jw	Javanese	ka	Georgian	kk	Kazakh
kl	Greenlandic	km	Cambodian	kn	Kannada
ko	Korean	ks	Kashmiri	ku	Kurdish
ky	Kirghiz	la	Latin	ln	Lingala
lo	Laothian	lt	Lithuanian	lv	Latvian, Lettish
mg	Malagasy	mi	Maori	mk	Macedonian
ml	Malayalam	mn	Mongolian	mo	Moldavian
mr	Marathi	ms	Malay	mt	Maltese
my	Burmese	na	Nauru	ne	Nepali
nl	Dutch	no	Norwegian	oc	Occitan
om	(Afan) Oromo	or	Oriya	pa	Punjabi
pl	Polish	ps	Pashto, Pushto	pt	Portuguese
qu	Quechua	rm	Rhaeto-Romance	rn	Kirundi

Code	Language	Code	Language	Code	Language
ro	Romanian	ru	Russian	rw	Kinyarwanda
sa	Sanskrit	sd	Sindhi	sg	Sangho
sh	Serbo-Croatian	si	Sinhalese	sk	Slovak
sl	Slovenian	sm	Samoan	sn	Shona
so	Somali	sq	Albanian	su	Sundanese
ss	Siswati	st	Sesotho	sv	Swedish
sw	Swahili	ta	Tamil	te	Telugu
tg	Tajik	th	Thai	ti	Tigrinya
tk	Turkmen	tl	Tagalog	tn	Setswana
to	Tonga	tr	Turkish	ts	Tsonga
tt	Tatar	tw	Twi	ug	Uighur
uk	Ukrainian	ur	Urdu	uz	Uzbek
vi	Vietnamese	vo	Volapuk	wo	Wolof
xh	Xhosa	yi	Yiddish (formerly ji)	yo	Yoruba
za	Zhuang	zh	Chinese	zu	Zulu

Table 113. Locale country codes, capitalized by country.

Code	Country	Code	Country
AF	Afghanistan	AL	Albania
DZ	Algeria	AS	American Samoa
AD	Andorra	AO	Angola
AI	Anguilla	AQ	Antarctica
AG	Antigua and Barbuda	AR	Argentina
AM	Armenia	AW	Aruba
AU	Australia	AT	Austria
AZ	Azerbaijan	BS	Bahamas
BH	Bahrain	BD	Bangladesh
BB	Barbados	BY	Belarus
BE	Belgium	BZ	Belize
BJ	Benin	BM	Bermuda
BT	Bhutan	BO	Bolivia
BA	Bosnia and Herzegovina	BW	Botswana
BV	Bouvet Island	BR	Brazil
IO	British Indian Ocean Territory	BN	Brunei Darussalam
BG	Bulgaria	BF	Burkina Faso
BI	Burundi	KH	Cambodia
CM	Cameroon	CA	Canada
CV	Cape Verde	KY	Cayman Islands
CF	Central African Republic	TD	Chad
CL	Chile	CN	China
CX	Christmas Island	CC	Cocos (Keeling) Islands
CO	Colombia	KM	Comoros
CD	Congo, Democratic Republic of (was Zaire)	CG	Congo, People's Republic of
CK	Cook Islands	CR	Costa Rica
CI	Cote D'Ivoire	HR	Croatia (local name: Hrvatska)
CU	Cuba	CY	Cyprus
CZ	Czech Republic	DK	Denmark
DJ	Djibouti	DM	Dominica
DO	Dominican Republic	TL	East Timor
EC	Ecuador	EG	Egypt
SV	El Salvador	GQ	Equatorial Guinea
ER	Eritrea	EE	Estonia
ET	Ethiopia	FK	Falkland Islands (Malvinas)
FO	Faroe Islands	FJ	Fiji

Code	Country	Code	Country
FI	Finland	FR	France
FX	France, Metropolitan	GF	French Guiana
PF	French Polynesia	TF	French Southern Territories
GA	Gabon	GM	Gambia
GE	Georgia	DE	Germany
GH	Ghana	GI	Gibraltar
GR	Greece	GL	Greenland
GD	Grenada	GP	Guadeloupe
GU	Guam	GT	Guatemala
GN	Guinea	GW	Guinea-Bissau
GY	Guyana	HT	Haiti
HM	Heard and Mc Donald Islands	HN	Honduras
HK	Hong Kong	HU	Hungary
IS	Iceland	IN	India
ID	Indonesia	IR	Iran (Islamic Republic Of)
IQ	Iraq	IE	Ireland
IL	Israel	IT	Italy
JM	Jamaica	JP	Japan
JO	Jordan	KZ	Kazakhstan
KE	Kenya	KI	Kiribati
KP	Korea, Democratic People's Republic Of	KR	Korea, Republic Of
KW	Kuwait	KG	Kyrgyzstan
LA	Lao People's Democratic Republic	LV	Latvia
LB	Lebanon	LS	Lesotho
LR	Liberia	LY	Libyan Arab Jamahiriya
LI	Liechtenstein	LT	Lithuania
LU	Luxembourg	MK	Macedonia, The Former Yugoslav Republic Of
MG	Madagascar	MW	Malawi
MY	Malaysia	MV	Maldives
ML	Mali	MT	Malta
MH	Marshall Islands	MQ	Martinique
MR	Mauritania	MU	Mauritius
YT	Mayotte	MX	Mexico
FM	Micronesia, Federated States Of	MD	Moldova, Republic Of
MC	Monaco	MN	Mongolia
MS	Montserrat	MA	Morocco
MZ	Mozambique	MM	Myanmar

Code	Country	Code	Country
NA	Namibia	NR	Nauru
NP	Nepal	NL	Netherlands
AN	Netherlands Antilles	NC	New Caledonia
NZ	New Zealand	NI	Nicaragua
NE	Niger	NG	Nigeria
NU	Niue	NF	Norfolk Island
MP	Northern Mariana Islands	NO	Norway
OM	Oman	PK	Pakistan
PW	Palau	PS	Palestinian Territory, Occupied
PA	Panama	PG	Papua New Guinea
PY	Paraguay	PE	Peru
PH	Philippines	PN	Pitcairn
PL	Poland	PT	Portugal
PR	Puerto Rico	QA	Qatar
RE	Reunion	RO	Romania
RU	Russian Federation	RW	Rwanda
KN	Saint Kitts And Nevis	LC	Saint Lucia
VC	Saint Vincent and The Grenadines	WS	Samoa
SM	San Marino	ST	Sao Tome and Principe
SA	Saudi Arabia	SN	Senegal
SC	Seychelles	SL	Sierra Leone
SG	Singapore	SK	Slovakia (Slovak Republic)
SI	Slovenia	SB	Solomon Islands
SO	Somalia	ZA	South Africa
GS	South Georgia and The South Sandwich Islands	ES	Spain
LK	Sri Lanka	SH	St. Helena
PM	St. Pierre And Miquelon	SD	Sudan
SR	Suriname	SJ	Svalbard And Jan Mayen Islands
SZ	Swaziland	SE	Sweden
CH	Switzerland	SY	Syrian Arab Republic
TW	Taiwan	TJ	Tajikistan
TZ	Tanzania, United Republic Of	TH	Thailand
TG	Togo	TK	Tokelau
TO	Tonga	TT	Trinidad and tobago
TN	Tunisia	TR	Turkey
TM	Turkmenistan	TC	Turks And Caicos Islands
TV	Tuvalu	UG	Uganda

Code	Country	Code	Country
UA	Ukraine	AE	United Arab Emirates
GB	United Kingdom	US	United States
UM	United States Minor Outlying Islands	UY	Uruguay
UZ	Uzbekistan	VU	Vanuatu
VA	Vatican City State (Holy See)	VE	Venezuela
VN	Viet Nam	VG	Virgin Islands (British)
VI	Virgin Islands (U.S.)	WF	Wallis and Futuna Islands
EH	Western Sahara	YE	Yemen
YU	Yugoslavia	ZM	Zambia
ZW	Zimbabwe		

I make extensive use of styles in the OOo documents that I write. I use a specific paragraph style to format my code samples. Each paragraph style allows you to set the default character attributes to use in the paragraph. In OOo, characters specify the locale so I set the locale for this paragraph style to unknown; this prevents the spell-checker from checking my code samples.

To tell OOo that a word is French, and should be checked as French, set the locale of the characters to French. The code in Listing 312 traverses the paragraphs in a document and sets the locale on each one.

Listing 312. *Set a simple Writer document to use a French locale.*

```
Sub SetDocumentLocale
    Dim aLocale As New com.sun.star.lang.Locale
    aLocale.Language = "fr"      'Set the Locale to use the French language
    aLocale.Country = "FR"      'Set the Locale to use France as the country
    Dim oCursor      'Cursor used to traverse the document.
    Dim oText        'The document text object

    oText = ThisComponent.Text      'Writer documents have a Text object
    oCursor = oText.createTextCursor() 'Create a text cursor

    REM Move cursor to the start of the document and do not select text.
    oCursor.GoToStart(False)

    REM Move to the end of the paragraph, selecting the entire paragraph.
    REM gotoNextParagraph() returns False if it fails.
    Do While oCursor.gotoNextParagraph(True)
        oCursor.CharLocale = aLocale 'This can fail for some paragraph types
        oCursor.goRight(0, False)    'Deselect all text
    Loop
End Sub
```

The spell-checker, hyphenation, and thesaurus all require a locale to function. They will not function, however, if they aren't properly configured. Use **Tools | Options | Language Settings |**

Writing Aids to configure these in OOo. The macro in Listing 313 obtains a SpellChecker, Hyphenator, and Thesaurus, all of which require a Locale object.

Listing 313. Spell Check, Hyphenate, and Thesaurus.

```
Sub SpellCheckExample
    Dim s()                'Contains the words to check
    Dim vReturn            'Value returned from SpellChecker, Hyphenator, and Thesaurus
    Dim i As Integer       'Utility index variable
    Dim msg$               'Message string

    REM Although I create an empty argument array, I could also
    REM use Array() to return an empty array.
    Dim emptyArgs() as new com.sun.star.beans.PropertyValue

    Dim aLocale As New com.sun.star.lang.Locale
    aLocale.Language = "en" 'Use the English language
    aLocale.Country = "US"  'Use the United States as the country

    REM Words to check for spelling, hyphenation, and thesaurus
    s = Array("hello", "anesthesiologist", _
        "PNEUMONULTRAMICROSCOPICSILICOVOLCANOCONIOSIS", _
        "Pitonyak", "misspell")

    REM *****Spell Check Example!
    Dim vSpeller As Variant
    vSpeller = createUnoService("com.sun.star.linguistic2.SpellChecker")
    'Use vReturn = vSpeller.spell(s, aLocale, emptyArgs()) if you want options!
    For i = LBound(s()) To UBound(s())
        vReturn = vSpeller.isValid(s(i), aLocale, emptyArgs())
        msg = msg & vReturn & " for " & s(i) & CHR$(10)
    Next
    MsgBox msg, 0, "Spell Check Words"
    msg = ""

    '*****Hyphenation Example!
    Dim vHyphen As Variant
    vHyphen = createUnoService("com.sun.star.linguistic2.Hyphenator")
    For i = LBound(s()) To UBound(s())
        'vReturn = vHyphen.hyphenate(s(i), aLocale, 0, emptyArgs())
        vReturn = vHyphen.createPossibleHyphens(s(i), aLocale, emptyArgs())
        If IsNull(vReturn) Then
            'hyphenation is probably off in the configuration
            msg = msg & " null for " & s(i) & CHR$(10)
        Else
            msg = msg & vReturn.getPossibleHyphens() & " for " & s(i) & CHR$(10)
        End If
    Next
    MsgBox msg, 0, "Hyphenate Words"
    msg = ""
```

```

'*****Thesaurus Example!
Dim vThesaurus As Variant
Dim j As Integer, k As Integer
vThesaurus = createUnoService("com.sun.star.linguistic2.Thesaurus")
s = Array("hello", "stamp", "cool")
For i = LBound(s()) To UBound(s())
    vReturn = vThesaurus.queryMeanings(s(i), aLocale, emptyArgs())
    If UBound(vReturn) < 0 Then
        Print "Thesaurus found nothing for " & s(i)
    Else
        msg = "Word " & s(i) & " has the following meanings:" & CHR$(10)
        For j = LBound(vReturn) To UBound(vReturn)
            msg=msg & CHR$(10) & "Meaning = " & vReturn(j).getMeaning() & CHR$(10)
            msg = msg & Join(vReturn(j).querySynonyms(), " ") & CHR$(10)
        Next
        MsgBox msg, 0, "Althernate Meanings"
    End If
Next
End Sub

```

It is possible to obtain the default locale that is configured for OOo. Laurent Godard, an active OpenOffice.org volunteer, wrote the macro in Listing 314, which obtains OOo's currently configured locale.

Listing 314. Currently configured language (Locale).

```

Sub OOoLang()
    'Retreives the running OOO version
    'Author : Laurent Godard
    'e-mail : listes.godard@laposte.net
    ,

    Dim aSettings, aConfigProvider
    Dim aParams2(0) As new com.sun.star.beans.PropertyValue
    Dim sProvider$, sAccess$
    sProvider = "com.sun.star.configuration.ConfigurationProvider"
    sAccess = "com.sun.star.configuration.ConfigurationAccess"
    aConfigProvider = createUnoService(sProvider)
    aParams2(0).Name = "nodepath"
    aParams2(0).Value = "/org.openoffice.Setup/L10N"
    aSettings = aConfigProvider.createInstanceWithArguments(sAccess, aParams2())

    Dim OOLangue as string
    OOLangue= aSettings.getbyname("ooLocale")      'en-US
    MsgBox "OOo is configured with Locale " & OOLangue, 0, "OOo Locale"
End Sub

```

The above macro is reputed to fail on AOO, but, the following should work correctly. The macro does function correctly with LO. The macro shown below should work for both AOO and LO.

Listing 315. You can use the tools library to get the current locale as a string.

```

GlobalScope.BasicLibraries.loadLibrary( "Tools" )
Print GetRegistryKeyContent("org.openoffice.Setup/L10N",FALSE).getByName("ooLocale")

```

13.14. Enumerating printers

The ability to enumerate printers has long been lacking in OOo. To obtain a printer list in OOo version 1.x or 2.x I used the .uno:print dispatch to open the print dialog, and then I directly access the dialog to extract the list of available printers.

The PrinterServer service was introduced, but it was not properly constructed, so the methods in the XPrinterServer interface are not directly available; according to Ariel Constenla-Haile this is slated to be ready in OOo 3.5. Thankfully, Niklas Nebel provided a working solution. The solution is a bit tricky, but it demonstrates how to work around the problem.

Listing 316. Display available printers.

```
Sub PrintAllPrinterNames()  
    Dim oPrintServer ' The print server service.  
    Dim oCore         ' Get classes and other objects by name.  
    Dim oClass        ' XPrinterServer class object.  
    Dim oMethod       ' getPrinterNames method from the XPrinterServer class.  
    Dim aNames        ' List of printer names.  
  
    ' Create the object that will not be directly usable until OOo 3.5.  
    oPrintServer = CreateUnoService("com.sun.star.awt.PrinterServer")  
    oCore = CreateUnoService("com.sun.star.reflection.CoreReflection")  
  
    ' Get the class object for the XPrinterServer interface.  
    oClass = oCore.forName("com.sun.star.awt.XPrinterServer")  
  
    ' Get the getPrinterNames method for the XPrinterServer class.  
    oMethod = oClass.getMethod("getPrinterNames")  
  
    ' Call the getPrinterNames method on the PrinterServer object.  
    aNames = oMethod.invoke(oPrintServer, Array())  
    MsgBox Join(aNames, CHR$(10))  
End Sub
```

TIP	If OOo does not recognize your printer by it's name, surround the printer name with angle brackets "<printer_name>", in the past some printers required this.

Prior to OOo 3.5, the PrinterServer cannot be used from Basic, because XTypeProvider is not declared to be inherited by that class; although it is implemented in the class. Basic cannot, therefore, recognize the methods supported by the object. The code in Listing 316 demonstrates how to call the method directly.

13.15. Printing documents

The primary printing functionality is common to all OOo document types. The interface com.sun.star.view.XPrintable defines three methods (see Table 114).

Table 114. Object methods defined by com.sun.star.view.XPrintable.

Object Method	Description
getPrinter()	Default printer as an array of properties (com.sun.star.view.PrinterDescriptor).
setPrinter(properties)	Assign a new printer to the object (com.sun.star.view.PrinterDescriptor).
print(properties)	Print the document (com.sun.star.view.PrintOptions).

The object method getPrinter() returns an array of properties that describe the printer (see Figure 90). The macro in Listing 317 demonstrates how to access and interpret each of the properties (see Table 115 for supported properties).

Listing 317. Display printer properties.

```
Sub DisplayPrinterProperties
    Dim Props 'Array of com.sun.star.beans.PropertyValue
    Dim i%    'Index variable of type Integer
    Dim s$    'Display string
    Dim v     '
    Dim sName$ '
    On Error Resume Next
    Props = ThisComponent.getPrinter()
    For i = 0 To UBound(Props)
        sName = props(i).Name
        v = props(i).Value
        s = s & sName & " = "
        If sName = "PaperOrientation" Then
            REM com.sun.star.view.PaperOrientation.LANDSCAPE also supported
            s = s & IIf(v=com.sun.star.view.PaperOrientation.PORTRAIT, _
                "Portrait", "Landscape") & " = " & CStr(v)
        ElseIf sName = "PaperFormat" Then
            Select Case v
                Case com.sun.star.view.PaperFormat.A3
                    s = s & "A3"
                Case com.sun.star.view.PaperFormat.A4
                    s = s & "A4"
                Case com.sun.star.view.PaperFormat.A5
                    s = s & "A5"
                Case com.sun.star.view.PaperFormat.B4
                    s = s & "B4"
                Case com.sun.star.view.PaperFormat.B5
                    s = s & "B5"
                Case com.sun.star.view.PaperFormat.LETTER
                    s = s & "LETTER"
                Case com.sun.star.view.PaperFormat.LEGAL
                    s = s & "LEGAL"
                Case com.sun.star.view.PaperFormat.TABLOID
                    s = s & "TABLOID"
                Case com.sun.star.view.PaperFormat.USER
                    s = s & "USER"
                Case Else
                    s = s & "Unknown value"
            End Select
        End If
    Next i
End Sub
```

```

End Select
s = s & " = " & CStr(v)
ElseIf sName = "PaperSize" Then
    REM type is com.sun.star.awt.Size
    REM The size is in TWIPS and there are 1440 twips per inch
    s=s & CDb1(v.Width)/1440.0 & "x" & CDb1(v.Height)/1440.0 & " (inches)"
Else
    s = s & CStr(v)
End If
s = s & CHR$(10)
Next
MsgBox s, 0, "Printer Properties"
End Sub

```

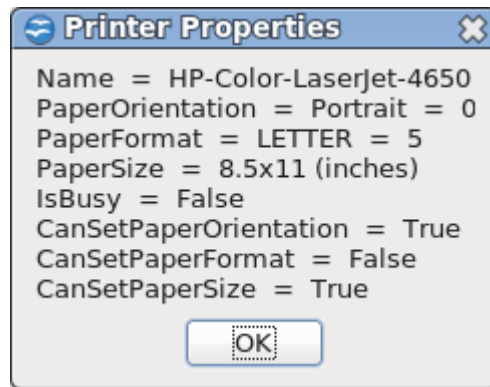


Figure 90. Properties of the default printer.

Table 115. Properties defined by the service `com.sun.star.view.PrinterDescriptor`.

Property	Description
Name	Name of the printer queue.
PaperOrientation	Paper orientation (<code>com.sun.star.view.PaperOrientation</code>).
PaperFormat	Predefined paper sizes (<code>com.sun.star.view.PaperFormat</code>).
PaperSize	Paper size in twips (<code>com.sun.star.awt.Size</code>).
IsBusy	Is the printer busy?
CanSetPaperOrientation	Can the paper orientation be set?
CanSetPaperFormat	Are other paper formats supported?
CanSetPaperSize	Can the paper size be set?

Call the `print()` method with no properties to print a single copy of the document to the current printer. All of the document types support the properties in Table 116. The Pages property supports the standard format used in the Print dialog. The format string “1, 3, 4-7, 9-” prints pages 1, 3, 4 through 7, and 9 through the last page.

Table 116. Properties defined by `com.sun.star.view.PrintOptions`.

Property	Description
Collate	Collate the printed pages (set to True or False).

Property	Description
CopyCount	Number of copies to print.
DuplexMode	Use the com.sun.star.view.DuplexMode constants to set duplex mode. Supported values include UNKNOWN (0), OFF (1), LONGEDGE (2), and SHORTEDGE (3).
FileName	Send the output to a file rather than to the printer.
Pages	Specifies pages and page ranges to print.
Wait	If TRUE, the print request is synchronous; wait for the job to finish before returning.

Listing 318. *Print pages 30 and 31 of the current document.*

```
Dim Props(1) As New com.sun.star.beans.PropertyValue
Props(0).Name = "Pages" : Props(0).Value = "30-31"
ThisComponent.print(Props())
```

When a document is printed, control returns to the caller before the printing is complete. If you close the document before printing is done, OpenOffice is likely to crash because the OO internals are still using the document. It's possible to set up an event listener that watches for the print job to finish, but there's an easier way that isn't currently documented. The print() method accepts an array of properties that direct the printing. The "wait" argument with a Boolean value of True instructs the print() method to not return until after printing is complete. As of May 17, 2011 with Oo 3.3, the wait argument does not work; print listener is demonstrated in section 13.15.3 A Calc example with a Print listener.

TIP Closing a document while OpenOffice.org is printing the document can cause OpenOffice.org to crash. Use the "wait" property to avoid this problem.

On Unix-type computers, printers are configured to work with OpenOffice.org using the "spadmin" utility. After a printer has been configured for Oo, it is available for use by name. You can still use printers that have not been configured for Oo, but you must enclose the printer name between the characters "<" and ">". To print with a printer other than the default, use code similar to the following:

```
Public oProps(0) as New com.sun.star.beans.PropertyValue
Public oOpts(1) as New com.sun.star.beans.PropertyValue
Dim oDoc          'Document to print.
Dim oPrinter      'Array of properties that define the printer.
Dim sUrl$         'URL of the document to load and print.
Dim sPrinter$     'Name of the printer.

REM Set the name of the printer as known by the system.
sPrinter = "HP-Color-LaserJet-4650DN"

REM Load the document in hidden mode so it is not visible
REM on the screen.
oProps(0).Name = "Hidden"
oProps(0).Value = True
```

```

REM Now load the document.
sUrl = "file:///c:/test_doc.sxw"
oDoc = oDesk.LoadComponentFromUrl(sUrl, "_blank", 63, oProps())

REM Obtain the current printer object from the document.
REM This is really an array of property values.
REM Change the name of the object to reference the printer that
REM you want to use. Notice that the printer name is the system name.
oPrinter = oDoc.getPrinter()
For i = LBound(oPrinter) to UBound(oPrinter)
    If oPrinter(i).Name = "Name" Then
        oPrinter(i).Value = sPrinter
    End If
Next i

REM Set the printer back into the document. The only thing
REM that has changed is the printer name.
oDoc.setPrinter(oPrinter)

REM Now, set up the print options for actual printing.
REM Notice that the printer name is surrounded by the characters < and >.
REM Also notice that the print() method is set to not return until
REM after printing is completed.
oOpts(0).Name = "Name"
oOpts(0).Value = "<" & sPrinter & ">"
oOpts(1).Name = "Wait"
oOpts(1).Value = True
oDoc.Print(oOpts())

```

TIP Historically, it has been experimentally determined that you must set the destination printer in the document before trying to print to a printer other than the default.

13.15.1. Printing Writer documents

Different document types support extra options for printing. Text documents support the interface `com.sun.star.text.XPagePrintable` (see Table 117). The `XPagePrintable` interface implements an alternate method of printing the document that provides more control of the output. The primary advantage is that you can print multiple pages from the document on a single output page.

Table 117. Methods defined by the `com.sun.star.text.XPagePrintable` interface.

Object Methods	Description
<code>getPagePrintSettings()</code>	Returns an array of properties (see Table 118).
<code>setPagePrintSettings(properties)</code>	Change the settings (see Table 118).
<code>printPages(properties)</code>	Print using properties in Table 116.

The object method `printPages()` accepts the same properties as the `print()` method (see Table 116). The methods to get and set the page print properties are outlined in Table 118. The macro in Listing 319 obtains and prints the current page print properties, shown in Figure 91.

Table 118. Properties used by the `com.sun.star.text.XPagePrintable` interface.

Property	Description
PageRows	Number of rows of pages on each printed page.
PageColumns	Number of columns of pages on each printed page.
LeftMargin	Left margin.
RightMargin	Right margin.
TopMargin	Top margin.
BottomMargin	Bottom margin.
HoriMargin	Margin between rows of pages.
VertMargin	Margin between columns of pages.
IsLandscape	True or False; print in landscape format.

Listing 319. Display Page Print Properties.

```

Sub DisplayPagePrintProperties
    Dim Props 'Array of com.sun.star.beans.PropertyValue
    Dim i%    'Index variable of type Integer
    Dim s$    'Display string
    If HasUnoInterfaces(ThisComponent, "com.sun.star.text.XPagePrintable") Then
        Props = ThisComponent.getPagePrintSettings()
        For i = 0 To UBound(Props)
            s = s & props(i).Name & " = " & CStr(props(i).Value) & CHR$(10)
        Next
        MsgBox s, 0, "Page Print Properties"
    Else
        Print "Sorry, this document does not support the XPagePrintable interface"
    End If
End Sub

```

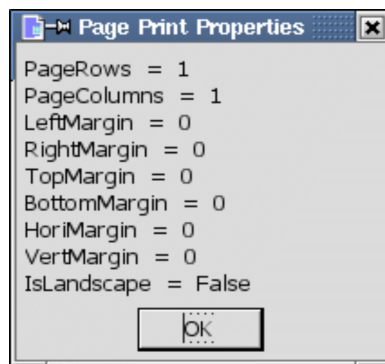


Figure 91. Page print properties of a Writer document.

The macro in Listing 320 attempts to print a document with two pages on each printed page in landscape mode. With my latest testing in OOo version 3.3, the document is printed normally, which is an improvement over the previous behavior of crashing OOo.

Listing 320. Print Two Per Page.

```
Sub PrintTwoPerPage
    Dim Props(0 To 1) As New com.sun.star.beans.PropertyValue
    Props(0).Name = "PageColumns" : Props(0).Value = 2
    Props(1).Name = "IsLandscape" : Props(1).Value = True
    If HasUnoInterfaces(ThisComponent, "com.sun.star.text.XPagePrintable") Then
        ThisComponent.setPagePrintSettings(Props()) '
        ThisComponent.printPages(Array())           'Use default properties
    Else
        Print "Sorry, this document does not support the XPagePrintable interface"
    End If
End Sub
```

Although the macro in Listing 320 stopped working with OOo 1.1.1, a small modification allows the macro to run. The new macro still causes OOo to crash under Linux and to close the document under Windows, but it does manage to print the document. As with Listing 320, the document is printed in portrait mode rather than landscape mode — this is an OOo bug.

13.15.2. Printing Calc documents

To perform special printing functions with a Writer document, a special object method is called. To perform special printing functions with Calc documents, you must modify the document properties and page-style properties and then use the standard print() method. For example, it is common for a Calc sheet to be too large to fit on a single sheet of paper. To scale the sheet to fit on a specified number of pages, set the ScaleToPages property to contain the number of pages that should contain the sheet. To simply scale the page based on a percentage, use the PageScale property (see Listing 321).

Listing 321. Print a spreadsheet at 25 percent; this is very small!

```
Sub PrintScaledSpreadsheet
    Dim s$ 'Style name
    Dim oStyle 'The current page style

    REM Use the currently active sheet to obtain the page style.
    REM In a Calc document, the current controller knows which sheet
    REM is active.
    s = ThisComponent.CurrentController.getActiveSheet().PageStyle
    oStyle = ThisComponent.StyleFamilies.getByName("PageStyles").getByName(s)
    REM oStyle.PageScale = 100 Default value is 100 (as in 100%)
    REM oStyle.ScaleToPages = 0 Default value is 0, as in don't scale
    oStyle.PageScale = 25 'Scale document to 25% (very very very small)
    ThisComponent.Print(Array()) 'Print the document
End Sub
```

The second aspect to printing Calc documents involves setting the area to print along with the column and row titles (see Table 119).

Table 119. Properties used by the com.sun.star.sheet.XPrintAreas interface.

Object Method	Description
getPrintAreas()	Return array of type com.sun.star.table.CellRangeAddress.
setPrintAreas(ranges)	Set print areas for the sheet with array of type CellRangeAddress. Print everything if nothing is set.
getPrintTitleColumns()	Return True if title columns are repeated to the right.
setPrintTitleColumns(boolean)	Set True if title columns are repeated on all print pages to the right.
getTitleColumns()	Array of type com.sun.star.table.CellRangeAddress.
setTitleColumns(ranges)	Set columns to use as titles. Rows are ignored; only columns matter.
getPrintTitleRows()	Return True if title rows are repeated on all print pages.
setPrintTitleRows(boolean)	Set to True if row titles are repeated on all print pages to the bottom.
getTitleRows()	Return array of type com.sun.star.table.CellRangeAddress.
setTitleRows(ranges)	Set rows to use as titles. Columns are ignored; only rows matter.

The methods in Table 119 are based on sheets in a Calc document; as opposed to the entire Calc document. The macro in Listing 322 sets two print ranges and then prints the document. Each print range is printed on a new page.

Listing 322. *Set and print multiple ranges in a Calc document.*

```
Sub PrintSpreadsheetAreas
    Dim oRanges(1) As New com.sun.star.table.CellRangeAddress
    oRanges(0).Sheet = 0
    oRanges(0).StartColumn = 0 : oRanges(0).StartRow = 0 'A1
    oRanges(0).EndColumn = 3 : oRanges(0).EndRow = 4 'D5

    oRanges(1).Sheet = 0
    oRanges(1).StartColumn = 0 : oRanges(1).StartRow = 8 'A9
    oRanges(1).EndColumn = 3 : oRanges(1).EndRow = 10 'D11

    ThisComponent.CurrentController.getActiveSheet().setPrintAreas(oRanges())
    ThisComponent.Print(Array())
End Sub
```

13.15.3. A Calc example with a Print listener

Sheet1 in a Calc document contains a button that prints Sheet2 and leaves Sheet1 as the active sheet. To print Sheet2, it must be made the active sheet. The code was structured as follows:

1. Set Sheet2 active.
2. Call the document level print method.
3. Set Sheet1 active.

The print method returns immediately and the document is printed in the background. Sheet1, therefore, becomes the active sheet before printing begins, so Sheet1 is printed rather than

Sheet2. The correct solution is to register a print listener that sets Sheet1 active after printing is complete.

A listener is stored in a Global variable so that it lives after the macro is finished running. Be warned, however, that if you edit any macro the global variable will be erased but the listener will still be registered; and you will have no way to unregister it without closing the document. So, first, create the variables to reference the listener. The document is also stored, but I could have referenced ThisComponent instead.

Listing 323. Global variables that reference the listener and the document.

```
Global oPrintListener
Global oPrintJobListnerDoc
```

The current active sheet is set as part of the real world macro, so it is also shown here.

Listing 324. Utility routine to set a Calc document's active sheet.

```
REM *****
REM ** oDoc - Calc sheet on which to operate. No error checking is performed
REM ** sSheetName - Sheet name to make active. The sheet is verified to exist.
REM *****
Sub set_active_sheet(oDoc, sSheetName)
    Dim oSheets
    oSheets = oDoc.Sheets
    If oSheets.hasByName(sSheetName) Then
        oDoc.currentController.setActiveSheet(oSheets.getByName(sSheetName))
    End If
End Sub
```

An event is sent to the listener if the listener is being disposed. The text “print_listener_” prefixes the routines used to implement the listener.

Listing 325. Print listener disposing method.

```
REM *****
REM ** The print job is disposing, so, remove it.
REM *****
Sub print_listener_disposing(oEvent)
    On Error Resume Next
    Dim emptyObj
    If NOT IsNull(oPrintJobListnerDoc) AND NOT IsEmpty(oPrintJobListnerDoc) Then
        oPrintJobListnerDoc.removePrintJobListener(oPrintListener)
        oPrintJobListnerDoc = emptyObj
    End If
End Sub
```

The listener is called every time the print job changes state.

Listing 326. Print listener status changed event.

```
REM *****
REM ** Called everytime the state of the print job changes.
REM ** Error messages are printed if they occur.
```



```

REM ** If the event announces an error or that the job is finished,
REM ** the print job listener is removed and the "Input Data" sheet is set to active.
REM *****
Sub print_listener_printJobEvent(oPrintJobEvent)
    Dim bCleanup As Boolean ' Set to true if the listener should be removed.
    Dim sMessage$          ' If not empty, a message is printed.

    REM All supported event state changes are shown.
    Select Case oPrintJobEvent.State
        Case com.sun.star.view.PrintableState.JOB_STARTED
            ' Rendering the document to print.
            bCleanup = False
        Case com.sun.star.view.PrintableState.JOB_COMPLETED
            ' Rendering is finished, start spooling.
            bCleanup = False
        Case com.sun.star.view.PrintableState.JOB_SPOOLED
            ' Success!
            'sMessage = "Print job spooled to the printer."
            bCleanup = True
        Case com.sun.star.view.PrintableState.JOB_ABORTED
            sMessage = "Printing was aborted."
            bCleanup = True
        Case com.sun.star.view.PrintableState.JOB_FAILED
            sMessage = "Error printing."
            bCleanup = True
        Case com.sun.star.view.PrintableState.JOB_SPOOLING_FAILED
            sMessage = "Document failed to print or spool to the print."
            bCleanup = True
        Case Else
            sMessage = "Unknown unexpected print state."
            bCleanup = True
    End Select

    REM Remove the listener if printing is finished, then call a worker macro
    REM to set Sheet1 as active.
    If bCleanup AND NOT IsNull(oPrintJobListnerDoc) AND NOT IsEmpty(oPrintJobListnerDoc) Then
        On Error Resume Next
        Dim emptyObj
        oPrintJobListnerDoc.removePrintJobListener(oPrintListener)
        Call set_active_sheet(oPrintJobListnerDoc, "Sheet1")
        oPrintJobListnerDoc = emptyObj
    End If
    If sMessage <> "" Then
        MsgBox sMessage
    End If
End Sub

```

If the sheet name exists, the desired sheet is set to be active, the print job listener is created and registered, and then document printing is requested.

Listing 327. Print the specified sheet.

```

Sub PrintSheet(oDoc, sSheetToPrint)
    Dim sPrefix$ ' Prefix used to identify the print listener routines.

```

```

Dim sService$ ' Print listener service name.

sPrefix = "print_listener_"
sService = "com.sun.star.view.XPrintJobListener"

If NOT oDoc.sheets().hasByName(sSheetToPrint) Then
    MsgBox "Document does not contain a sheet named " & sSheetToPrint
    Exit sub
End If

Call set_active_sheet(oDoc, sSheetToPrint)

oPrintListener = CreateUnoListener(sPrefix, sService)
oDoc.addPrintJobListener(oPrintListener)
oPrintJobListnerDoc = oDoc
oDoc.Print(Array())
End Sub

```

13.15.4. Print examples by Vincent Van Houtte

I found some very nice examples by Vincent Van Houtte, that he has graciously allowed me to reproduce here. Some highlights of the contained macros:

- printDoc – Print a document with specific paper size, with specific paper trays, and with (or without) a page background image. Numerous stub methods are included to call this one routine. Note that the method directly specifies the destination printer.
- printPage – Same as printDoc, but prints a single page.
- closeDocument – Close the specified document.
- ExportAsPdfAndSendEmail – Export as a PDF document and send an email.

TIP The very popular routine FindCreateNumberFormatStyle (see Listing 391) is used by Listing 328.

Listing 328. Print examples by Vincent Van Houtte.

```

Sub PrintWithCopyStamp()
' -----
' This macro inserts a 'COPY'-stamp and prints the document
' without the background(-image) to tray1.
'
' Written by Vincent Van Houtte (2010)
' -----
    REM Insert timestamp of sending
    Dim sActionText AS String
    sActionText = "KOPIE"
    InsertDTstamp(sActionText)

```

```

REM Print the page
PrintDocWithoutBgToTray1()

REM Remove the copystamp-frame
RemoveDTstamp()
End Sub

Sub InsertDTstamp(sActionText)
' -----
' This macro inserts a 'DATE/TIME'-stamp with sActionText 'Printed' or 'Sent'
'
' Written by Vincent Van Houtte (2011)
' -----
    DIM oCursor, oText, oDoc AS Object
    oDoc = ThisComponent
    oText = oDoc.getText()
    oCursor = oText.createTextCursor()
    oCursor.goToStart(FALSE)

    REM Create the date and time objects
    DIM oDate, oTime AS Object
    oDate = oDoc.createInstance("com.sun.star.text.TextField.DateTime")
    oDate.IsFixed = TRUE
    oDate.IsDate = TRUE
    oDate.NumberFormat = FindCreateNumberFormatStyle("D MMMM JJJJ", oDoc)

    oTime = oDoc.createInstance("com.sun.star.text.TextField.DateTime")
    oTime.IsFixed = True
    oTime.IsDate = False
    oTime.NumberFormat = FindCreateNumberFormatStyle("UU:MM", oDoc)

    REM Create the frame
    DIM oFrameDT AS Object
    oFrameDT = oDoc.createInstance("com.sun.star.text.TextFrame")
    With oFrameDT
        .setName("FrameDT")
        .AnchorType = com.sun.star.text.TextContentAnchorType.AT_PAGE
        .HoriOrient = com.sun.star.text.HoriOrientation.NONE
        .VertOrient = com.sun.star.text.VertOrientation.NONE
        .HoriOrientPosition = -4900
        .VertOrientPosition = -1600
        .width = 4000
        .height = 1500
        .BorderDistance = 100
    End With

    REM Insert the frame into the text document
    oText.insertTextContent( oCursor, oFrameDT, True )

    REM Write the text inside the frame
    DIM oCursor2 AS Object
    oCursor2 = oFrameDT.createTextCursor()
    With oCursor2

```

```

.charHeight = 16
.charWeight = com.sun.star.awt.FontWeight.BOLD
.paraAdjust = com.sun.star.style.ParagraphAdjust.CENTER
End With

oFrameDT.insertString( oCursor2, sActionText, False )

With oCursor2
.charHeight = 9
.charWeight = com.sun.star.awt.FontWeight.NORMAL
.paraAdjust = com.sun.star.style.ParagraphAdjust.CENTER
End With

oFrameDT.insertControlCharacter( oCursor2,
com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, False )
oFrameDT.insertTextContent( oCursor2, oDate, False )

oFrameDT.insertControlCharacter( oCursor2, com.sun.star.text.ControlCharacter.LINE_BREAK,
False )
oFrameDT.insertTextContent( oCursor2, oTime, False )
End Sub

Sub RemoveDTstamp()
' -----
' This macro removes the 'DATE/TIME'-stamp created with the previous macro
'
' Written by Vincent Van Houtte (2011)
' -----
DIM oDoc, oTextFrames, oFrameDT AS Object
oDoc = ThisComponent

REM Look for the datetimestamp-frame and remove it
oTextFrames = oDoc.getTextFrames
If oTextFrames.hasByName("FramedT") Then
    oFrameDT = oTextFrames.getByName("FramedT")
    oFrameDT.dispose()
EndIf
End Sub

Sub PrintDocWithBgToTray1
' -----
' This macro prints the document with the background(-image) to the
' first papertray. This is only used to save on stationary, if you
' have run out on stationary or if the second printertray is jammed ;)
'
' Written by Vincent Van Houtte (2010)
' -----
DIM sTray1 AS String
DIM sTray2 AS String
DIM bBg1 AS Boolean
DIM bBg2 AS Boolean
DIM wait AS Boolean

```

```

    sTray1 = "Tray1"
    sTray2 = "Tray1"
    bBg1 = True
    bBg2 = False
    wait = True
    printDoc(sTray1, sTray2, bBg1, bBg2, wait)
End Sub

Sub PrintDocWithoutBgToTray1
' -----
' This macro prints the document without the background(-image) to the
' first papertray. This is useful for copies of the letters you sent out.
'
' Written by Vincent Van Houtte (2010)
' -----
    DIM sTray1 AS String
    DIM sTray2 AS String
    DIM bBg1 AS Boolean
    DIM bBg2 AS Boolean
    DIM wait AS Boolean
    sTray1 = "Tray1"
    sTray2 = "Tray1"
    bBg1 = False
    bBg2 = False
    wait = True
    printDoc(sTray1, sTray2, bBg1, bBg2, wait)
End Sub

Sub PrintDocWithoutBgToTray2_old
' -----
' This macro prints the document without the background(-image) to the
' second papertray. This is useful when you have pre-printed stationary:
' you can set an image as a background, that you don't want to print,
' but that you want to show up when converted to PDF.
'
' Written by Vincent Van Houtte (2010)
' -----
    DIM sTray1 AS String
    DIM sTray2 AS String
    DIM bBg1 AS Boolean
    DIM bBg2 AS Boolean
    DIM wait AS Boolean
    sTray1 = "Tray2"
    sTray2 = "Tray1"
    bBg1 = False
    bBg2 = False
    wait = True
    printDoc(sTray1, sTray2, bBg1, bBg2, wait)
End Sub

Sub PrintDocWithoutBgToTray2
' -----
' This macro prints the first page (without the backgroundimage) to the

```

```

' second papertray and all other pages (without the backgroundimage) to the
' first papertray.
' This is useful when you have pre-printed stationary, but want to save on it
' by only printing the first page on expensive stationary
' you can set an image as a background, that you don't want to print,
' but that you want to show up when converted to PDF.
'
' Written by Vincent Van Houtte (2011)
' -----

DIM oDoc AS Object
DIM iPageCount, n AS Integer
DIM sPage AS String
oDoc = ThisComponent

REM Count the amount of pages
iPageCount = oDoc.GetCurrentController().getPropertyValue("PageCount")

REM Loop over every page
n = 1
Do Until n > iPageCount

REM Print every page to the right tray
    If n = 1 Then
        Print_stat(n)
    Else
        Print_plain(n)
    End If
    n = n + 1
Loop

End Sub

Sub Print_stat(sPageNr AS String)
' -----
' This macro prints the first page without the background(-image) to the
' second papertray. This is useful when you have pre-printed stationary:
' you can set an image as a background, that you don't want to print,
' but that you want to show up when converted to PDF.
'
' Written by Vincent Van Houtte (2011)
' -----

DIM sTray1 AS String
DIM sTray2 AS String
DIM bBg1 AS Boolean
DIM bBg2 AS Boolean
DIM wait AS Boolean
sTray1 = "Tray2"
sTray2 = "Tray1"
bBg1 = False
bBg2 = False
wait = True
printPage(sTray1, sTray2, bBg1, bBg2, wait, sPageNr)

```

```

End Sub

Sub Print_plain(sPageNr AS String)
' -----
' This macro prints the next page without the background(-image) to the
' first papertray. This is useful when you want to saveon your pre-printed stationary:
' you can set an image as a background, that you don't want to print,
' but that you want to show up when converted to PDF.
'
' Written by Vincent Van Houtte (2011)
' -----
    DIM sTray1 AS String
    DIM sTray2 AS String
    DIM bBg1 AS Boolean
    DIM bBg2 AS Boolean
    DIM wait AS Boolean
    sTray1 = "Tray1"
    sTray2 = "Tray1"
    bBg1 = False
    bBg2 = False
    wait = True
    printPage(sTray1, sTray2, bBg1, bBg2, wait, sPageNr)
End Sub

Sub printDoc(sTray1, sTray2, bBg1, bBg2, wait)
' -----
' This macro prints the document given the chosen arguments
'
' Written by Vincent Van Houtte (2010)
' -----
    DIM oDoc AS Object
    oDoc = ThisComponent

    REM Set backgroundImage-option in DocumentSettings to False
    DIM oSettings AS Object
    oSettings = oDoc.CreateInstance("com.sun.star.text.DocumentSettings")
    oSettings.PrintPageBackground = bBg1

    REM choose a certain printer
    DIM mPrinterOpts(2) AS NEW com.sun.star.beans.PropertyValue
    mPrinterOpts(0).Name = "Name"
    mPrinterOpts(0).Value = "MFC8880DN"
    mPrinterOpts(1).Name = "PaperFormat"
    mPrinterOpts(1).Value = com.sun.star.view.PaperFormat.A4
    mPrinterOpts(2).Name = "PaperOrientation"
    mPrinterOpts(2).Value = com.sun.star.view.PaperOrientation.PORTRAIT
    oDoc.Printer = mPrinterOpts()

    REM set Papertray in Styles
    DIM oStyle AS Object
    DIM sPageStyle AS String
    sPageStyle = oDoc.CurrentController.getViewCursor().PageStyleName
    ostyle = oDoc.StyleFamilies.getByName("PageStyles").getByName(sPageStyle)

```

```

oStyle.PrinterPaperTray = sTray1

REM Set printOptions
DIM mPrintOpts(2) AS NEW com.sun.star.beans.PropertyValue
mPrintOpts(0).Name = "CopyCount"
mPrintOpts(0).Value = 1
mPrintOpts(1).Name = "Collate"
mPrintOpts(1).Value = True
mPrintOpts(2).Name = "Wait"
mPrintOpts(2).Value = True

REM Print
oDoc.Print(mPrintOpts())

REM RESET OPTIONS
REM Reset backgroundImage-option in DocumentSettings
oSettings.PrintPageBackground = bBg2
REM Reset Papertray in Styles
oStyle.PrinterPaperTray = sTray2
REM Do a 0-print to complete the reset
DIM mPrintOpts2(0) AS NEW com.sun.star.beans.PropertyValue
mPrintOpts2(0).Name = "CopyCount"
mPrintOpts2(0).Value = 0
oDoc.Print(mPrintOpts2())
End Sub

Sub printPage(sTray1, sTray2, bBg1, bBg2, wait, sPageNr)
' -----
' This macro prints the document given the chosen arguments
'
' Written by Vincent Van Houtte (2010)
' -----
    DIM oDoc AS Object
    oDoc = ThisComponent

    REM Set backgroundImage-option in DocumentSettings to False
    DIM oSettings AS Object
    oSettings = oDoc.CreateInstance("com.sun.star.text.DocumentSettings")
    oSettings.PrintPageBackground = bBg1

    REM choose a certain printer
    DIM mPrinterOpts(3) AS NEW com.sun.star.beans.PropertyValue
    mPrinterOpts(0).Name = "Name"
    mPrinterOpts(0).Value = "MFC8880DN"
    mPrinterOpts(1).Name = "PaperFormat"
    mPrinterOpts(1).Value = com.sun.star.view.PaperFormat.A4
    mPrinterOpts(2).Name = "PaperOrientation"
    mPrinterOpts(2).Value = com.sun.star.view.PaperOrientation.PORTRAIT
    oDoc.Printer = mPrinterOpts()

    REM set Papertray in Styles
    DIM oStyle AS Object

```



```

DIM sPageStyle AS String
sPageStyle = oDoc.CurrentController.getViewCursor().PageStyleName
ostyle = oDoc.StyleFamilies.getByName("PageStyles").getByName(sPageStyle)
oStyle.PrinterPaperTray = sTray1

REM Set printOptions
DIM mPrintOpts(3) AS NEW com.sun.star.beans.PropertyValue
mPrintOpts(0).Name = "CopyCount"
mPrintOpts(0).Value = 1
mPrintOpts(1).Name = "Collate"
mPrintOpts(1).Value = True
mPrintOpts(2).Name = "Pages"
mPrintOpts(2).Value = sPageNr
mPrintOpts(3).Name = "Wait"
mPrintOpts(3).Value = True

REM Print
oDoc.Print(mPrintOpts())

REM RESET OPTIONS
REM Reset backgroundImage-option in DocumentSettings
oSettings.PrintPageBackground = bBg2
REM Reset Papertray in Styles
oStyle.PrinterPaperTray = sTray2
REM Do a 0-print to complete the reset
DIM mPrintOpts2(0) AS NEW com.sun.star.beans.PropertyValue
mPrintOpts2(0).Name = "CopyCount"
mPrintOpts2(0).Value = 0
oDoc.Print(mPrintOpts2())
End Sub

Sub closeDocument(oDoc AS Object)
' -----
' This macro closes the current document
'
' Written by Andrew Pitonyak (2010)
' Adapted by Vincent Van Houtte (2011)
' -----

REM Check if the document exists
If IsNull(oDoc) Then
    Exit Sub
End If

REM Store the document if it was modified
If (oDoc.isModified) Then
    If (oDoc.hasLocation AND (Not oDoc.isReadOnly)) Then
        oDoc.store()
    Else
        oDoc.setModified(False)
    End If
End If

```

```

End If

REM Close the document
oDoc.close(true)
End Sub

Sub ExportAsPdfAndSendEmail
' -----
' This macro converts the active document to PDF and then adds the PDF
' to a new emailmessage. The recipientaddress and subject are
' generated automagically from a textfield inside the document
'
' This macro assumes the default email application is set
' in Tools -> Options -> OpenOffice.org -> External Programs.
'
' This macro uses SimpleCommandMail, which might not work in windows
' Try SimpleSystemMail instead
'
' Written by Vincent Van Houtte (2010)
' -----
    DIM oDoc, MailClient, MailAgent, MailMessage AS Object
    DIM sDocURL, sPDFURL, sTo, sSubject AS String

    REM Get location of the doc
    oDoc = ThisComponent
    If (Not oDoc.hasLocation()) Then
        oDoc.store()
    End if

    REM Insert timestamp of sending
    Dim sActionText AS String
    sActionText = "VERZONDEN"
    InsertDTstamp(sActionText)

    REM Print the page
    PrintDocWithoutBgToTray1()

    REM Replace .odt with .pdf
    sDocURL = oDoc.getURL()
    sPDFURL = Left$(sDocURL, Len(sDocURL)-4) + ".pdf"

    REM Save as PDF
    DIM args(0) AS NEW com.sun.star.beans.PropertyValue
    args(0).Name = "FilterName"
    args(0).Value = "writer_pdf_Export"
    oDoc.storeToURL(sPDFURL, args())

    REM Remove the Date/time-stamp
    RemoveDTstamp()

    REM Get the values of the textfields inside the document to form the subject line
    DIM enuTF, aTextField AS Object
    DIM sDosName, sDosNum, sDosUref AS String

```

```

enuTF = oDoc.TextFields.createEnumeration
Do While enuTF.hasMoreElements
    aTextField = enuTF.nextElement
    if aTextField.supportsService("com.sun.star.text.TextField.Input") then
        Select Case aTextField.getPropertyValue("Hint")
            Case "DOS_NAAM":
                sDosName = aTextField.getPropertyValue("Content")
            Case "DOS_NUM":
                sDosNum = aTextField.getPropertyValue("Content")
            Case "REF_O":
                sDosNum = aTextField.getPropertyValue("Content")
            Case "UREF":
                sDosUref = aTextField.getPropertyValue("Content")
            Case "EMAIL_ADDR":
                sTo = aTextField.getPropertyValue("Content")
        End Select
    end if
Loop
sSubject = sDosName + " - " + sDosUref + " - " + sDosNum

REM Send the PDF as an attachment
MailAgent = CreateUnoService("com.sun.star.system.SimpleCommandMail")
MailClient = MailAgent.querySimpleMailClient()
    MailMessage=MailClient.createSimpleMailMessage()
    MailMessage.setRecipient(sTo)
    MailMessage.setSubject(sSubject)
    MailMessage.setAttachement(array(sPDFURL))
MailClient.sendSimpleMailMessage(MailMessage, 0)

REM Save and close the document
closeDocument(oDoc)
End Sub

```

13.16. Creating services

An object that supports the XMultiServiceFactory interface may create services. An object is usually created by the object that will own it. As an example, a text table is created by the text document that will contain the text table. Similarly, the DocumentSettings object is created by the document of interest. The function `CreateUnoService` creates services in the scope of the OOo application.

The `oSettings` object in Listing 329 supports the service `com.sun.star.text.DocumentSettings`. The `oSettings` object is related to `ThisComponent` in that it reflects the settings for `ThisComponent` — the current document — but not for any other document.

Listing 329. *Create an object that supports the DocumentSettings service.*

```
oSettings=ThisComponent.createInstance("com.sun.star.text.DocumentSettings")
```

The `createInstance()` method returns an object that supports the requested service if it can; if it cannot, it returns `NULL`. The returned object may support other services as well (see Listing 318 and Figure 92).

Listing 330. Inspect a text document settings object.

```
Sub WriteDocumentSettings
    Dim oSettings 'Settings object to create
    Dim s$        'Utility string
    Dim i%        'Utility index variable
    Dim v         'This will contain an array of service names

    REM Create an object that supports the DocumentSettings service
    oSettings=ThisComponent.createInstance("com.sun.star.text.DocumentSettings")

    v = oSettings.getSupportedServiceNames()
    s = "**** Specified Supported Services ****" & CHR$(10) & Join(v, CHR$(10))
    s = s & CHR$(10) & CHR$(10) & "**** Tested Services ****" & CHR$(10)

    REM Now check to see if this created object supports any other services.
    v = Array("com.sun.star.comp.Writer.DocumentSettings",_
        "com.sun.star.text.PrintSettings")

    For i = 0 To UBound(v)
        If oSettings.supportsService(v(i)) Then
            s = s & "Supports service " & v(i) & CHR$(10)
        End If
    Next
    MsgBox s, 0, "Some services for " & oSettings.getImplementationName()

    REM What is the status of the PrintControls property?
    Print oSettings.PrintControls

    REM I could set this to True or False
    'oSettings.Printcontrols = True
End Sub
```

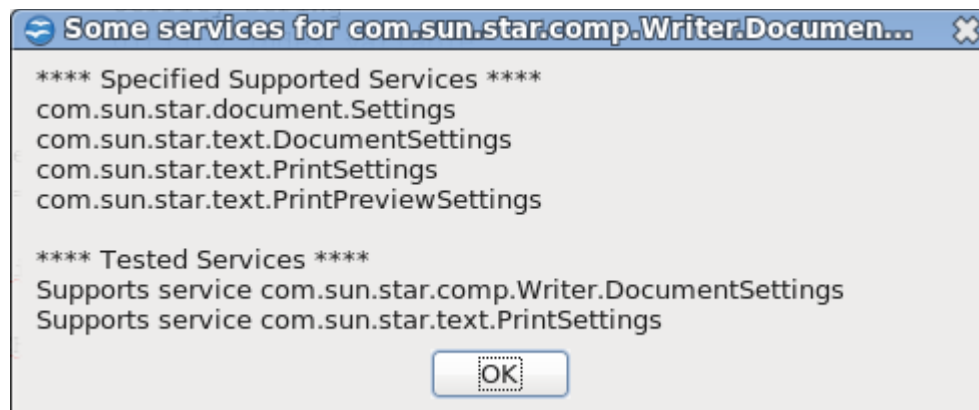


Figure 92. Some Writer document settings.

A careful inspection of Listing 330 and Figure 92 demonstrate that the object supports multiple services.

The method `getSupportedServiceNames()` lists the services supported by the object.

The method `getImplementationName()` returns the service that uniquely identifies the returned service (`com.sun.star.comp.Writer.DocumentSettings`). The unique service name is not in the list of supported service names.

Although not shown in Listing 330, the two “Tested Services” in Figure 92 cannot be created using the `createInstance()` object method. I discovered the `PrintSettings` service entirely by accident. I inspected the `dbg_properties` property, which contained `PrintControls`. I then searched Google for “`site:http://www.openoffice.org/api/docs PrintSettings`”. The important things to notice are as follows:

1. There are different methods of inspecting an object; use all of them.
2. The documentation is not complete. You must manually inspect an object to see what it supports. However, undocumented methods and properties may be deprecated and removed later — so do this carefully.
3. You can search the Internet for services and properties.

13.17. Document settings

OOo contains numerous document options; these are available in the **Tools | Options** dialog. For example, Text Document contains a **Print** tab, which allows graphics to be printed, or not. The typical method for modifying properties in a document involves using either “get” and “set” methods, or directly accessing object properties. To access advanced document settings, however, you must use a completely different method. First the document has to create an object that supports the document settings. Every document type, except for the Math type, has the ability to create a service using the `createInstance()` method. Table 120 contains a list of the properties common to all of the document setting types; there are more.

Table 120. Common document properties in `com.sun.star.document.Settings`.

Property	Description
<code>ForbiddenCharacters</code>	Allow access to otherwise forbidden characters.
<code>LinkUpdateMode</code>	The update mode for links when loading text documents.
<code>PrinterName</code>	Printer used by the document.
<code>PrinterSetup</code>	Platform- and driver-dependent printer setup data.
<code>IsKernAsianPunctuation</code>	Is kerning applied to Asian punctuation?
<code>CharacterCompressionType</code>	Compression (character spacing) used for Asian characters.
<code>ApplyUserData</code>	Should user-specific settings saved in a document be loaded again?
<code>SaveVersionOnClose</code>	Is a new version created when a modified document is closed?

Property	Description
UpdateFromTemplate	Should a document be updated when its creation template changes?
FieldAutoUpdate	Are text document fields automatically updated?
CurrentDatabaseDataSource	Name of the data source from which the current data is taken.
CurrentDatabaseCommand	Displayed object's name (or the SQL statement used).
CurrentDatabaseCommandType	Specifies how to interpret the DataTableName property.
DefaultTabStop	Default tab width.
IsPrintBooklet	Is the document printed as a booklet (brochure)?
IsPrintBookletFront	If True, only the front pages of a booklet are printed.
IsPrintBookletBack	If True, only the back pages of a booklet are printed.
PrintQuality	Quality to use when printing.
ColorTableURL	URL of the color table (SOC file) showing a palette in dialogs that use colors.
PrinterIndependentLayout	If true, do not use printer metrics for layout.

Specific document settings services exist for Writer, Calc, Draw, and Impress documents (see Table 121). Although each of these services implements the Settings service as shown in Table 120, all of the properties except for PrinterName and PrinterSetup are optional.

Table 121. Specific document settings service types.

Document Settings Service	Document Type
com.sun.star.text.DocumentSettings	Writer
com.sun.star.sheet.DocumentSettings	Calc
com.sun.star.drawing.DocumentSettings	Draw
com.sun.star.presentation.DocumentSettings	Impress

As shown in Listing 330 and Figure 92, the Writer document settings support the PrintSettings service (see Table 122). The Draw and Impress document settings services each contain special settings that apply only to their respective document types.

Table 122. Properties defined by the com.sun.star.text.PrintSettings service.

Property	Description
PrintGraphics	If True, print graphic object.
PrintTables	If True, print text tables.
PrintDrawings	If True, print shapes.
PrintLeftPages	If True, print left pages.
PrintRightPages	If True, print right pages.
PrintControls	If True, print control contained in the document.
PrintReversed	If True, print the pages in reverse order, starting with the last page.
PrintPaperFromSetup	If True, use the paper tray specified for the system printer. If False, use the paper tray specified by the page style.

Property	Description
PrintFaxName	Name of the fax.
PrintAnnotationMode	Specify how notes are printed. Uses com.sun.star.text.NotePrintMode enumerated value. Possible values include NOT, ONLY, DOC_END, or PAGE_END.
PrintProspect	If True, prospect printing is used; however, I cannot find a definition for prospect printing.
PrintPageBackground	If True, the background color and/or background graphic is printed.
PrintBlackFonts	If True, characters are always printed in black.

13.18. The coolest trick I know

Sadly, this only works in LibreOffice, and not in OOo. While editing a document:

1. Use **File > Open** to open the Open dialog.
2. Enter “vnd.sun.star.tdoc:” for the file name and click Open.
3. Currently open documents are shown for selection. Select a document; and click open.

You are looking at the document's internal structures. You can even open an image embedded in a document. A document containing pictures is a good example to try.

13.19. Converting to a URL in other languages

Basic provides the methods `ConvertToURL` and `ConvertFromURL`, other languages do not. Some URI schemes leave unspecified important aspects of how to interpret URIs of those schemes. For example, it is unspecified for “file” URLs how to map the byte sequences that constitute the path segments of a “file” URL to filenames on a given platform: The UNO environment always assumes that path segments of “file” URLs represent UTF-8–encoded strings (which have to be mapped to filenames in a platform-specific way), while other applications typically assume that path segments of “file” URLs directly represent a platform's byte-sequence filenames. The `ExternalUriReferenceTranslator` offers methods to translate between such internal URIs (e.g., UTF-8–encoded “file” URLs used within the UNO environment) and external URIs (e.g., byte-sequence–oriented “file” URLs used by other applications). Typically, only “file” URLs are affected by this translation.

```
x = CreateUnoService("com.sun.star.uri.ExternalUriReferenceTranslator")
Print x.translateToExternal("file:///c:/MyDoc.oot")
Print x.translateToInternal("file:/c:/MyDoc.oot")
```

13.20. Conclusion

The interfaces and services introduced in this chapter provide a good introduction to the capabilities of OOo that are not directly related to a specific document type. OOo contains numerous other capabilities that I could have included in this chapter, but it isn't possible to

exhaustively cover every subject. Use these topics and methods as a starting point to investigate the other capabilities of OpenOffice.org.

14. Writer Documents

Writer documents primarily deal with text content arranged in paragraphs. This chapter introduces appropriate methods to manipulate, traverse, search, format, and modify the content contained in an OpenOffice.org Writer document. First, a bit of review that is applicable to most document types.

Conceptually, all document types have two components: the data they contain and the controller that determines how the data is displayed. Writer documents primarily contain simple formatted text. In addition to simple text, Writer documents may contain other content such as tables, frames, graphics, text fields, bookmarks, footnotes, endnotes, text sections, index entries, tracked document changes (called redlining), objects for styling, and objects for numbering. OOo uses the same methods and interfaces to interact with most of these capabilities. Consequently, learning to manipulate a few types of content will give you a solid basis for dealing with all of them.

TIP In OOo, the data contained in a document is called the “model.” The base model interface is `com.sun.star.frame.XModel`.

In OpenOffice.org, the data contained in a document is called the model. Each model has a controller that is responsible for the visual presentation of the data. The controller knows the location of the visible text cursor, the current page, and what is currently selected.

TIP While trying to determine which portion of the OOo API to use to solve a specific problem, first ask if the problem is display-centric or data-centric. For example, a paragraph boundary is specified in the data, but a new line is usually determined by the controller when the data is formatted.

Every text document supports the `com.sun.star.text.TextDocument` service. When I write a macro that must be user friendly and requires a text document, I verify that the document is the correct type by using the object method `supportsService` (see Listing 331).

Listing 331. *Text documents support the `com.sun.star.text.TextDocument` service.*

```
REM If it really matters, you should check the document type
REM to avoid a run-time error.
If NOT ThisComponent.supportsService("com.sun.star.text.TextDocument") Then
    MsgBox "The current document is not a Writer text document", 48, "Error"
Exit Sub
End If
```

An interface defines a series of methods. If an object implements an interface, it also implements every method defined by that interface. A service defines an object by specifying the interfaces that it implements, the properties that it contains, and the other services that it exports. A service indirectly specifies the implemented methods by specifying interfaces. The interfaces supported by the `TextDocument` service provide a good overview of the provided functionality (see Table 123).

Table 123. *Interfaces supported by text documents.*

Service	Description
com.sun.star.text.XTextDocument	Main text document interface.
com.sun.star.text.XBookmarksSupplier	Access the bookmarks.
com.sun.star.text.XChapterNumberingSupplier	Numbering rules for chapters.
com.sun.star.text.XDocumentIndexesSupplier	Access the collection of indexes.
com.sun.star.text.XTextEmbeddedObjectsSupplier	Access embedded objects.
com.sun.star.text.XEndnotesSupplier	Access endnote content.
com.sun.star.text.XFootnotesSupplier	Access footnote content.
com.sun.star.text.XLineNumberingSupplier	Numbering rules for line numbers.
com.sun.star.text.XPagePrintable	Print multiple pages on one page.
com.sun.star.text.XReferenceMarksSupplier	Access the document reference marks, which are used to refer to text positions in a text document.
com.sun.star.text.XTextFieldsSupplier	Access contained fields.
com.sun.star.text.XTextFramesSupplier	Access contained text frames.
com.sun.star.text.XTextGraphicObjectsSupplier	Access embedded and linked graphics.
com.sun.star.text.XTextSectionsSupplier	Access contained text sections.
com.sun.star.text.XTextTablesSupplier	Access contained tables.
com.sun.star.style.XStyleFamiliesSupplier	Access contained styles by type.
com.sun.star.util.XNumberFormatsSupplier	Access contained number formats.
com.sun.star.util.XRefreshable	Refresh data that can be refreshed from a database.
com.sun.star.util.XReplaceable	Replace text found with a search descriptor.
com.sun.star.util.XSearchable	Search a text range for a specific string pattern.
com.sun.star.beans.XPropertySet	Access document properties by name.

14.1. Basic building blocks

While dealing with Writer documents, you'll see that a few simple interfaces and concepts recur. These basic building blocks are interrelated in that their respective interfaces are circularly defined (they refer to each other). Fortunately, the concepts are intuitive and therefore easy to grasp, even with a brief introduction. This section briefly introduces these basic building blocks, laying the groundwork for detailed coverage later.

14.1.1. Primary text content: the XText interface

Text content is contained in an object that implements the XText interface. The primary purpose of a text object is to contain text content, create text cursors to move through text, insert text, and remove text (see **Table 124**).

Table 124. *Methods defined by the com.sun.text.XText interface.*

Method	Description
<code>createTextCursor()</code>	Return a TextCursor service used to traverse the text object.
<code>createTextCursorByRange(XTextRange)</code>	Return a TextCursor that is located at the specified TextRange.
<code>insertString(XTextRange, String, boolean)</code>	Insert a string of characters into the text at the specified text range. Each CR (ASCII 13) inserts a new paragraph and each LF (ASCII 10) inserts a new line. If the Boolean value is True, the text in the range is overwritten; otherwise, the text characters are inserted after the text range.
<code>insertControlCharacter(XTextRange, Short, boolean)</code>	Insert a control character (such as a paragraph break or a hard space) into the text. The short integer is a value from the constant group <code>com.sun.star.text.ControlCharacter</code> : • PARAGRAPH_BREAK = 0 – Start a new paragraph. • LINE_BREAK = 1 – Start a new line in a paragraph. • HARD_HYPHEN = 2 – Insert a dash that will not hyphenate. • SOFT_HYPHEN = 3 – Define a preferred hyphenation point if the word must be split at the end of a line. • HARD_SPACE = 4 – Insert a space that prevents two words from splitting at a line break. • APPEND_PARAGRAPH = 5 – Append a new paragraph. If the Boolean value is True, the text in the text range is overwritten; otherwise, the control character is inserted after the text range.
<code>insertTextContent(XTextRange, XTextContent, boolean)</code>	Insert text content such as a text table, text frame, or text field. In general, the text content should be created by the text object. If the Boolean value is True, the text in the text range is overwritten; otherwise, the text content is inserted after the text range.
<code>removeTextContent(XTextContent)</code>	Remove the specified text content from the text object.

Because the XText interface is derived from the XTextRange interface, all objects that implement the XText interface also support the object methods defined by the XTextRange interface (see Table 125).

14.1.2. Text ranges: the XTextRange interface

A text range is one of the most important concepts in a text document because so many interfaces derive from the XTextRange interface. The primary purpose of a text range is to define a start and end position in the text (see Table 125). It is possible that the start position and the end position in a text range are the same. When the start and end positions are the same, the text range identifies a position in the text—for example, the cursor in a text document when no text is selected. If the start and end positions of a text range are not the same, they represent a section of text.

Table 125. *Methods defined by the com.sun.text.XTextRange interface.*

Method	Description
getText()	Return the XText interface that contains the text range.
getStart()	A text range has a start and end position. The getStart() method returns a text range that contains only the start position of this text range.
getEnd()	A text range has a start and end position. The getStart() method returns a text range that contains only the end position of this text range.
setString(String)	A text range has a start and end position. The setString() method replaces all of the text between the start and end positions with the argument string. All styles are removed and all text in this text range is replaced.
getString()	Return a string that represents the text in this text range. Strings in Basic are limited to 64KB in size, but text ranges and text objects are not; use this with care.

Every XTextRange object is associated with a text object. It can be said that an XTextRange object is contained in an object that implements the XText interface. The XText interface is itself derived from the XTextRange interface. Use the getText() object method to obtain the text object that is associated with the text range.

TIP Common tasks include obtaining a bookmark, obtaining the bookmark's anchor position (a text range), and then inserting text content at the bookmark's anchor position. Although the setString() method is the quickest way to add text at the anchor position, inserting text content requires an object that supports the XText interface. The getText() method returns an object that can insert text content using the text range.

Writer documents primarily contain formatted text. To access the formatted text, either call the document's object method getText() or directly access the document's Text property. I usually directly access the Text property because it requires less typing.

```
ThisComponent.Text      'Current document's text object
ThisComponent.getText() 'Current document's text object
```

The document's text object implements the XTextRange interface. The simplest method to obtain all of the text characters contained in a text document is to call the getString() object method (see Table 125). The getString() method returns a single string containing a text version of the document. Each cell in a text table is returned as a single paragraph, and all formatting is lost. The object method setString() can be used to set the entire document's text in one call—when setString() is used, all of the existing text is lost! See Listing 332.

Listing 332. *Get and set the entire document text with no formatting.*

```
MsgBox ThisComponent.Text.getString(), 0, "Document Text String"
ThisComponent.Text.setString("This is text to set")
```

TIP Use getString() only on small text documents. I wrote a macro that computed word-usage statistics by first calling the getString() object method. Because Basic strings are limited to 64KB characters, the macro failed on large documents.

The getString() and setString() object methods are limited, because Basic strings are limited to 64KB in size and they contain no formatting information. Due to these limitations, other methods

are usually used to get and set the document text. Generally, large or complex documents are best handled with the `getText()` method and related methods, because they support arbitrary size and modular management of complex documents.

14.1.3. Inserting simple text

With the limited information already presented, you can already insert simple text content at the beginning and the end of a document. The `getStart()` and `getEnd()` object methods both return a text range that can be used to insert text into a text object (see Table 124 and Table 125). The code in Listing 333 inserts simple text at the start of the document and a new paragraph at the end of the document.

Listing 333. *Insert simple text at the start and end of the document.*

```
Sub InsertSimpleText
    Dim oText As Object
    oText = ThisComponent.Text

    REM Insert some simple text at the start
    oText.insertString(oText.getStart(), "Text object start." & CHR$(13), False)
    REM Append a new paragraph at the end
    oText.InsertControlCharacter(oText.getEnd(), _
        com.sun.star.text.ControlCharacter.APPEND_PARAGRAPH, False)
End Sub
```

14.1.4. Text content: the TextContent service

The primary purpose of the TextContent service is to anchor an object (text content) to its surrounding text. Conceptually there are two types of text content: content that is part of the surrounding text (a text field, for example), and content that is more like a floating field (a graphic image, for example).

To determine where text content is anchored in the text, call the `getAnchor()` object method. This returns a text range that defines the anchor location. For floating text content objects, the text needs to know how to flow around the object and how the object is anchored to the text (see Table 126). The behavior of text content inserted as a character should be well understood. The content acts just like a character would act: The text content is moved along between two other characters. When text content is anchored at a paragraph, however, the text content is not required to move as the characters before and after it are moved. It is only required that the object stay attached to a paragraph, and the object is not inserted into a paragraph. I usually anchor an object to a paragraph when I want text to flow around the object; for example, a graphic on the right and text on the left.

Table 126. Properties supported by the *com.sun.star.text.TextContent* service.

Property	Description
AnchorType	Enumeration of type <i>com.sun.star.text.TextContentAnchorType</i> that defines how this text content is attached to the surrounding text. • <i>AT_PARAGRAPH</i> – The anchor is set at the top left position of the paragraph. The object moves if the paragraph moves. • <i>AS_CHARACTER</i> – The text content object is anchored as a character. The size of the object influences the height of the text line and the object can move as a character if the surrounding text moves. • <i>AT_PAGE</i> – The text content object is anchored to the page. The object does not move even if the text content around it changes. • <i>AT_FRAME</i> – The text content object is anchored to a text frame. • <i>AT_CHARACTER</i> – The text content object is anchored to a character. The object moves if the character moves.
AnchorTypes	Array of <i>TextContentAnchorType</i> that contains the anchor types of the text content.
TextWrap	Enumeration of type <i>com.sun.star.text.WrapTextMode</i> that defines how the surrounding text wraps around this text content object. • <i>NONE</i> – Text does not flow around the object. • <i>THROUGHT</i> – Text flow ignores the object. (Yes, it is <i>THROUGHT</i>.) This can be thought of as <i>THROUGH</i> it, as in, “the text flows through the object.” • <i>PARALLEL</i> – Text flows to the left and right of the object. • <i>DYNAMIC</i> – The text formatting decides the best wrapping method. • <i>LEFT</i> – Text flows to the left of the object. • <i>RIGHT</i> – Text flows to the right of the object.

The text object contains methods to insert *TextContent* objects at specified locations (see Table 124). In general, the *TextContent* type must be created by the document before it is inserted (see Listing 334).

Listing 334. Insert text content (a text table) at the end of the current document.

```

Sub InsertSimpleTableAtEnd
    Dim oTable 'Newly created table to insert

    REM Let the document create the text table.
    oTable = ThisComponent.CreateInstance( "com.sun.star.text.TextTable" )
    oTable.initialize(3, 2) 'Three rows, two columns

    REM Now insert the text table at the end of the document.
    ThisComponent.Text.insertTextContent(
        ThisComponent.Text.getEnd(), oTable, False)
End Sub

```

TIP

In general, the *TextContent* type must be created by the document before it is inserted.

Call the *removeTextContent(XTextContent)* object method (see Table 124) to remove text content. Another method of removing text content is to write new text content in its place. A simple

example of this is to call the `setString()` method on a text range that includes the text content (see Listing 335).

Listing 335. *Clear the entire document of all text content.*

```
ThisComponent.Text.setString("") 'Clear an entire document!
```

14.2. Enumerating paragraphs

Writer documents primarily contain formatted text that is arranged into paragraphs. Writer methods can operate on words, sentences, paragraphs, and entire text objects. Paragraphs are the most basic organizational unit for formatted text, and methods on paragraphs are often the most reliable, meaning they contain fewer bugs. The paragraphs can be enumerated sequentially using the `XEnumerationAccess` interface defined in the document's text object. OOo treats tables as a special type of paragraph, and they are returned while enumerating paragraphs (see Listing 336).

Listing 336. *Count paragraphs and text tables.*

```
Sub enumerateParagraphs
    Dim oEnum          'com.sun.star.container.XEnumerationAccess
    Dim oPar            'Paragraph of some sort
    Dim nPars As Integer 'Number of paragraphs
    Dim nTables As Integer 'Number of tables

    REM ThisComponent refers to the current OOo document
    REM Text is a property of ThisComponent for a text document
    REM The getText() object method returns the same thing.
    REM createEnumeration() is an object method.
    oEnum = ThisComponent.Text.createEnumeration()
    Do While oEnum.hasMoreElements()
        oPar = oEnum.nextElement()

        REM The returned paragraph will be a paragraph or a text table
        If oPar.supportsService("com.sun.star.text.Paragraph") Then
            nPars = nPars + 1
        ElseIf oPar.supportsService("com.sun.star.text.TextTable") Then
            nTables = nTables + 1
        End If
    Loop
    MsgBox CStr(nPars) & " Paragraph(s)" & CHR$(13) & _
        CStr(nTables) & " Table(s)" & CHR$(13), 0, _
        "Paragraph Types In Document"
End Sub
```

TIP

Visual Basic for Applications (VBA) supports accessing paragraphs using an index; OOo does not.

While enumerating the paragraphs in a Writer document, both paragraphs and tables are returned. The object method `supportsService()` is used to determine if a paragraph or a table is returned. Paragraph objects support both the `XTextRange` interface and the `XTextContent` interface. TextTable objects, however, support only the `XTextContent` interface.

The macro in Listing 336 enumerates paragraphs in the high level text object. Many objects contains their own text object; fore example, each cell in a text table and each frame.

14.2.1. Paragraph properties

Paragraphs contain numerous paragraph-related properties encapsulated in services. The properties that are primarily related to the entire paragraph are encapsulated in the ParagraphProperties service (see Table 127).

TIP The Paragraph service is not the only service that supports the ParagraphProperties service. Other services, especially those that are also a text range, also support paragraph properties. Techniques used to modify paragraph properties in paragraphs apply to these services as well.

Table 127. Properties supported by the *com.sun.style.ParagraphProperties* service.

Property	Description
ParaAdjust	Specify how the paragraph is aligned (or justified). Five values are supported from the <i>com.sun.star.style.ParagraphAdjust</i> enumeration: • LEFT – Left-align the paragraph. • RIGHT – Right-align the paragraph. • CENTER – Center-align the paragraph. • BLOCK – Fill-justify every line except for the last line. • STRETCH – Fill-justify every line including the last line.
ParaLastLineAdjust	Adjust the last line if the ParaAdjust is set to BLOCK.
ParaLineSpacing	Specify the paragraph line spacing. The property is a structure of type <i>com.sun.star.style.LineSpacing</i> , which contains two properties of type Short. The Height property specifies the height and the Mode property specifies how to use the Height property. The Mode property supports values defined in the <i>com.sun.star.style.LineSpacingMode</i> constants group. • PROP = 0 – The height is proportional. • MINIMUM = 1 – The height is the minimum line height. • LEADING = 2 – The height is the distance to the previous line. • FIX = 3 – The height is fixed.
ParaBackColor	Specify the paragraph background color as a Long Integer.
ParaBackTransparent	If True, set the paragraph background color to transparent.
ParaBackGraphicURL	Specify the URL of the paragraph background graphic.
ParaBackGraphicFilter	Specify the name of the graphic filter for the paragraph background graphic.
ParaBackGraphicLocation	Specify the position of the background graphic using the enumeration <i>sun.star.style.GraphicLocation</i> : • NONE – A location is not yet assigned. • LEFT_TOP – The graphic is in the top left corner. • MIDDLE_TOP – The graphic is in the middle of the top edge. • RIGHT_TOP – The graphic is in the top right corner. • LEFT_MIDDLE – The graphic is in the middle of the left edge. • MIDDLE_MIDDLE – The graphic is in the center of the surrounding object.

Property	Description
	• RIGHT_MIDDLE – The graphic is in the middle of the right edge. • LEFT_BOTTOM – The graphic is in the bottom left corner. • MIDDLE_BOTTOM – The graphic is in the middle of the bottom edge. • RIGHT_BOTTOM – The graphic is in the bottom right corner. • AREA – The graphic is scaled to fill the whole surrounding area. • TILED – The graphic is repeated over the surrounding object like tiles.
ParaExpandSingleWord	If True, single words may be stretched.
ParaLeftMargin	Specify the left paragraph margin in 0.01 mm as a Long Integer.
ParaRightMargin	Specify the right paragraph margin in 0.01 mm as a Long Integer.
ParaTopMargin	Specify the top paragraph margin in 0.01 mm as a Long Integer. The distance between two paragraphs is the maximum of the bottom margin of the previous paragraph and the top margin of the next paragraph.
ParaBottomMargin	Specify the bottom paragraph margin in 0.01 mm as a Long Integer. The distance between two paragraphs is the maximum of the bottom margin of the previous paragraph and the top margin of the next paragraph.
ParaLineNumberCount	If True, this paragraph is included in line numbering.
ParaLineNumberStartValue	Specify the start value for line numbering as a Long Integer.
PageDescName	Setting this string causes a page break to occur before the paragraph. The new page uses the specified page style name.
PageNumberOffset	Specify a new page number if a page break occurs.
ParaRegisterModeActive	If True, and if the paragraph's page style also has the register mode set to True, the register mode is active for this paragraph. If register mode is active, each line has the same height.
ParaTabStops	Specify the tab stops for this paragraph. This is an array of structures of type <code>com.sun.star.style.TabStop</code>. Each structure contains the following properties: • Position – Long integer position relative to the left border. • Alignment – Alignment of the text range before the tab. This is an enumeration of type <code>com.sun.star.style.TabAlign</code>. Valid values include LEFT, RIGHT, CENTER, DECIMAL, and DEFAULT. • DecimalChar – Specifies which character is the decimal. • FillChar – Character used to fill space between the text.
ParaStyleName	Specify the name of the current paragraph style.
DropCapFormat	Structure that determines if the first characters of the paragraph use dropped capital letters. The <code>com.sun.star.style.DropCapFormat</code> contains the following properties: • Lines – Number of lines used for a drop cap. • Count – Number of characters in the drop cap. • Distance – Distance between the drop cap and the following text.
DropCapWholeWord	If True, the DropCapFormat is applied to the entire first word.
ParaKeepTogether	If True, prevents a page or column break after this paragraph—for example, to prevent a title from being the last line on a page or column.
ParaSplit	If False, prevents the paragraph from splitting into two pages or columns.
NumberingLevel	Specify the numbering level of the paragraph.
NumberingRules	Specify the numbering rules applied to this paragraph. This object implements the <code>com.sun.star.container.XIndexReplace</code> interface.

Property	Description
NumberingStartValue	Specify the start value for numbering if ParaIsNumberingRestart is True.
ParaIsNumberingRestart	Specify if numbering restarts at the current paragraph (see NumberingStartValue).
NumberingStyleName	Specify the name of the style for numbering (see ParaLineNumberCount).
ParaOrphans	Specify the minimum number of lines at the bottom of a page if the paragraph spans more than one page.
ParaWidows	Specify the minimum number of lines at the top of a page if the paragraph spans more than one page.
ParaShadowFormat	Specify the paragraph shadow format as a com.sun.star.table.ShadowFormat: • Location – Specify the shadow location as an enumeration of type com.sun.star.table.ShadowLocation. Valid values include NONE, TOP_LEFT, TOP_RIGHT, BOTTOM_LEFT, and BOTTOM_RIGHT. • ShadowWidth – Specify the size of the shadow as an Integer. • IsTransparent – If True, the shadow is transparent. • Color – Specify the color of the shadow as a Long Integer.
LeftBorder	Specify the left border as a com.sun.star.table.BorderLine (see Table 164).
RightBorder	Specify the right border (see Table 164).
TopBorder	Specify the top border (see Table 164).
BottomBorder	Specify the bottom border (see Table 164).
BorderDistance	Specify the distance from the border to the object (in 0.01 mm).
LeftBorderDistance	Specify the distance from the left border to the object (in 0.01 mm).
RightBorderDistance	Specify the distance from the right border to the object (in 0.01 mm).
TopBorderDistance	Specify the distance from the top border to the object (in 0.01 mm).
BottomBorderDistance	Specify the distance from the bottom border to the object (in 0.01 mm).
BreakType	Specify the type of break that is applied at the start of the paragraph. This is an enumeration of type com.sun.star.style.BreakType with the following values: • NONE – No column or page break is applied. • COLUMN_BEFORE – A column break is applied before the current paragraph. The current paragraph, therefore, is the first in the column. • COLUMN_AFTER – A column break is applied after the current paragraph. The current paragraph, therefore, is the last in the column. • COLUMN_BOTH – A column break is applied before and after the current paragraph. The current paragraph, therefore, is the only paragraph in the column. • PAGE_BEFORE – A page break is applied before the current paragraph. The current paragraph, therefore, is the first on the page. • PAGE_AFTER – A page break is applied after the current paragraph. The current paragraph, therefore, is the last on the page. • PAGE_BOTH – A page break is applied before and after the current paragraph. The current paragraph, therefore, is the only paragraph on the page.
DropCapCharStyleName	Specify the name of the character style for drop caps.
ParaFirstLineIndent	Specify the indent for the first line in a paragraph.
ParaIsAutoFirstLineIndent	If True, the first line is indented automatically.
ParaIsHyphenation	If True, automatic hyphenation is applied.

Property	Description
ParaHyphenationMaxHyphens	Specify the maximum number of consecutive hyphens for each word contained in the current paragraph.
ParaHyphenationMaxLeadingChars	Specify the maximum number of characters to remain before a hyphen character.
ParaHyphenationMaxTrailingChars	Specify the maximum number of characters to remain after a hyphen character.
ParaVertAlign	Specify the vertical alignment of the paragraph. This is a constant group of type <code>com.sun.star.text.ParagraphVertAlign</code> with valid values: - AUTOMATIC = 0 – In automatic mode, horizontal text is aligned to the baseline. The same applies to text that is rotated 90 degrees. Text that is rotated 270 degrees is aligned to the center. - BASELINE = 1 – The text is aligned to the baseline. - TOP = 2 – The text is aligned to the top. - CENTER = 3 – The text is aligned to the center. - BOTTOM = 4 – The text is aligned to bottom.
ParaUserDefinedAttributes	Stores XML attributes that are saved and restored from the automatic styles inside XML files. The object implements the <code>com.sun.star.container.XNameContainer</code> interface.
NumberingIsNumber	If True, the numbering of a paragraph is a number but has no symbol. This is void if the paragraph is not part of a paragraph numbering sequence.
ParaIsConnectBorder	If True, paragraph borders are merged with the previous paragraph if the borders are identical. This property may be void.

TIP

Paragraph properties are usually set using paragraph styles—at least they should be.

Many of the properties in Table 127 are structures; they require special care if you want to modify them, because a structure is copied by value rather than by reference. For example, the `ParaLineSpacing` property is a structure. Although the code in Listing 337 looks correct, it fails; this usage is a very common error committed by Basic programmers.

Listing 337. *This code fails because `ParaLineSpacing` is a structure.*

```
oPar.ParLineSpacing.Mode = com.sun.star.style.LineSpacing.LEADING
```

The code in Listing 337 fails because the code “`oPar.ParLineSpacing`” made a copy of the structure. The `Mode` is set, but it is set only on the copy, leaving the original intact. The code in Listing 338 demonstrates the proper way to modify the value of a structure when it is used as a property. A copy of the structure is stored in the variable `v`, which is then modified and copied back.

Listing 338. *This works because it makes a copy and then copies it back.*

```
v = oPar.ParLineSpacing
v.Mode = com.sun.star.style.LineSpacing.LEADING
oPar.ParLineSpacing = v
```

Insert a page break

To insert a page break, set the PageDescName property to the name of the page style to use after the page break. This style may be the same as the current page style; it is the act of setting the PageDescName property—not changing it to a new value—that causes a page break to occur. The page style name must exist in the document or a page break is not inserted. When you insert a page break, you can also set a new page number by setting the PageNumberOffset property to the new page number. See Listing 339.

TIP It is rare to insert a page break while enumerating paragraphs. It is more common to insert a page break using a text cursor or a text range. You can use any service that supports paragraph properties to insert a page break.

Listing 339. Insert a page break after the last paragraph.

```
Sub SetPageBreakAtEndFromEnumeration
    Dim oEnum      'com.sun.star.container.XEnumerationAccess
    Dim oParTest   'Paragraph of some sort
    Dim oPar        'Last Paragraph object

    REM Find the last paragraph
    oEnum = ThisComponent.Text.createEnumeration()
    Do While oEnum.hasMoreElements()
        oParTest = oEnum.nextElement()
        If oParTest.supportsService("com.sun.star.text.Paragraph") Then
            oPar = oParTest
        End If
    Loop

    REM Note that this does not actually change the page style name
    oPar.PageDescName = oPar.PageStyleName

    REM Set the new page number to be 7
    oPar.PageNumberOffset = 7
End Sub
```

Set the paragraph style

The ParaStyleName property indicates the paragraph style for that paragraph. This property may be directly set.

Listing 340. Enumerate the paragraph styles in the current document.

```
Sub EnumerateParStyles()
    Dim oEnum      'com.sun.star.container.XEnumerationAccess
    Dim oCurPar    'Last Paragraph object.
    Dim s$         'General string variable.

    oEnum = ThisComponent.Text.createEnumeration()
    Do While oEnum.hasMoreElements()
        oCurPar = oEnum.nextElement()
        s = s & oCurPar.ParaStyleName & CHR$(10)
    Loop
```

14.2.2. Character properties

Paragraphs contain numerous character-related properties. Like the paragraph-specific properties, these properties are optional and are encapsulated in services. The properties that are primarily related to characters are found in the CharacterProperties service (see **Table 128**).

TIP Many of the properties are represented by a value in a constant group. Constant groups associate meaningful names to constant values. For example, the CharFontFamily accepts the value `com.sun.star.awt.FontFamily.ROMAN` to specify a Roman font with serifs. A value of 3 may also be used. In almost all cases, the first value is 0, the second value is 1, and so on. Code that uses the descriptive names is easier to read and understand.

Table 128. Properties supported by the *com.sun.style.CharacterProperties* service.

Property	Description
CharFontName	Specify the name of the font in western text. This may be a comma-separated list of names.
CharFontStyleName	Specify the name of the font style.
CharFontFamily	Specify the name of the font family as specified in <code>com.sun.star.awt.FontFamily</code> constant group. - DONTKNOW = 0 – The font family is not known. - DECORATIVE = 1 – The font family uses decorative fonts. - MODERN = 2 – The font family is a modern font; this is a specific style. - ROMAN = 3 – The font family is a Roman font with serifs. - SCRIPT = 4 – The font family is a script font. - SWISS = 5 – The font family is a Roman font without serifs. - SYSTEM = 6 – The font family is a system font.
CharFontCharSet	Specify the text encoding of the font using the <code>com.sun.star.awt.CharSet</code> constant group. The values are self-explanatory: DONTKNOW, ANSI, MAC, IBMPC_437 (IBM PC character set number 437), IBMPC_850, IBMPC_860, IBMPC_86, IBMPC_863, IBMPC_865, SYSTEM, and SYMBOL.
CharFontPitch	Specify the character font pitch using the <code>com.sun.star.awt.FontPitch</code> constant group. The values are self-explanatory: DONTKNOW, FIXED, and VARIABLE.
CharColor	Specify the text color as a Long Integer.
CharEscapement	Specify the Short Integer representing the percentage of raising or lowering for superscript/subscript characters. Negative values lower the characters.
CharHeight	Specify the character height in points as a decimal number.
CharUnderline	Specify the character underline type using the <code>com.sun.star.awt.FontUnderline</code> constant group. - NONE = 0 – No underlining. - SINGLE = 1 – Single line. - DOUBLE = 2 – Double line. - DOTTED = 3 – Dotted line. - DONTKNOW = 4 – Unknown underlining. - DASH = 5 – Dashed line.

Property	Description
	• LONGDASH = 6 – Long dashes. • DASHDOT = 7 – Dash and dot sequence. • DASHDOTDOT = 8 – Dash, dot, dot sequence. • SMALLWAVE = 9 – Small wave. • WAVE = 10 – Wave. • DOUBLEWAVE = 11 – Double wave. • BOLD = 12 – Bold line. • BOLDDOTTED = 13 – Bold dots. • BOLDDASH = 14 – Bold dashes. • BOLDLONGDASH = 15 – Long bold dashes. • BOLDDASHDOT = 16 – Dash and dot sequence in bold. • BOLDDASHDOTDOT = 17 – Dash, dot, dot sequence in bold. • BOLDWAVE = 18 – Bold wave.
CharWeight	Specify the font weight using the com.sun.star.awt.FontWeight constant group. • DONTKNOW = 0.000 – Not specified/unknown. • THIN = 50.00 – 50% font weight. • ULTRALIGHT = 60.00 – 60% font weight. • LIGHT = 75.00 – 75% font weight. • SEMILIGHT = 90.00 – 90% font weight. • NORMAL = 100.00 – normal font weight (100%). • SEMIBOLD = 110.00 – 110% font weight. • BOLD = 150.00 – 150% font weight. • ULTRABOLD = 175.00 – 175% font weight. • BLACK = 200.00 – 200% font weight.
CharPosture	Specify the character posture using the com.sun.star.awt.FontSlant enumeration with values: • NONE – No slant, regular text. • OBLIQUE – Oblique font (slant not designed into the font). • ITALIC – Italic font (slant designed into the font). • DONTKNOW – Unknown slant. • REVERSE_OBLIQUE – Reverse oblique (slant not designed into the font). • REVERSE_ITALIC – Reverse italic font (slant designed into the font).
CharAutoKerning	Set to True to use the kerning tables for the current font. Automatic kerning adjusts the spacing between certain pairs of characters to improve readability.
CharBackColor	Specify the text background color as a Long Integer.
CharBackTransparent	If True, the text background color is transparent.
CharCaseMap	Specify how characters should be displayed using the com.sun.star.style.CaseMap constant group. This does not change the actual text—only the way it is displayed. • NONE = 0 – No case mapping is performed; this is the most commonly used value. • UPPERCASE = 1 – All characters are displayed in uppercase. • LOWERCASE = 2 – All characters are displayed in lowercase. • TITLE = 3 – The first character of each word is displayed in uppercase.

Property	Description
	SMALLCAPS = 4 – All characters are displayed in uppercase, but with a smaller font.
CharCrossedOut	If True, the characters have a line through them.
CharFlash	If True, the characters are displayed flashing.
CharStrikeout	Specify character strikeout using the com.sun.star.awt.FontStrikeout constant group: - NONE = 0 – Do not strike out characters. - SINGLE = 1 – Strike out the characters with a single line. - DOUBLE = 2 – Strike out the characters with a double line. - DONTKNOW = 3 – The strikeout mode is not specified. - BOLD = 4 – Strike out the characters with a bold line. - SLASH = 5 – Strike out the characters with slashes. - X = 6 – Strike out the characters with X's.
CharWordMode	If True, white spaces (spaces and tabs) ignore the CharStrikeout and CharUnderline properties.
CharKerning	Specify the character kerning value as a Short Integer.
CharLocale	Specify the character locale as a com.star.lang.Locale structure.
CharKeepTogether	If True, OOo tries to keep the character range on the same line. If a break must occur, it occurs before the characters.
CharNoLineBreak	If True, OOo ignores a line break in the character range. If a break must occur, it occurs after the characters so it is possible that they will cross a border.
CharShadowed	If True, the characters are formatted and displayed with a shadow effect.
CharFontType	Specify the fundamental technology of the font using the com.sun.star.awt.FontType constant group. - DONTKNOW = 0 – The type of the font is not known. - RASTER = 1 – The font is a raster (bitmapped) font. - DEVICE = 2 – The font is output-device specific, such as a printer font. - SCALABLE = 3 – The font is scalable.
CharStyleName	Specify the name of the font style as a string.
CharContoured	If True, characters are formatted and displayed with a contour (3-D outline) effect.
CharCombineIsOn	If True, text is formatted and displayed using two lines. The CharCombinePrefix string precedes the text in full size, and the CharCombineSuffix follows the text in full size.
CharCombinePrefix	Specify the prefix (usually parentheses) used with the CharCombineIsOn property.
CharCombineSuffix	Specify the suffix (usually parentheses) used with the CharCombineIsOn property.
CharEmphasize	Specify the type and position of emphasis marks in Asian texts using the com.sun.star.text.FontEmphasis constant group: - NONE = 0 – No emphasis mark is used. - DOT_ABOVE = 1 – A dot is set above (or right from vertical text) the text. - CIRCLE_ABOVE = 2 – A circle is set above (or right from vertical text) the text. - DISK_ABOVE = 3 – A disk is set above (or right from vertical text) the text. - ACCENT_ABOVE = 4 – An accent is set above (or right from vertical text) the text.

Property	Description
	• DOT_BELOW = 11 – A dot is set below (or left from vertical text) the text. • CIRCLE_BELOW = 12 – A circle is set below (or left from vertical text) the text. • DISK_BELOW = 13 – A disk is set below (or left from vertical text) the text. • ACCENT_BELOW = 14 – An accent is set below (or left from vertical text) the text.
CharRelief	Specify the relief value from the com.sun.star.text.FontRelief constant group: • NONE = 0 – No relief is used; normal text. • EMBOSSED = 1 – Characters look embossed (raised). • ENGRAVED = 2 – Characters look engraved (lowered).
RubyText	Specify the text that is set as ruby. “Ruby Text” acts as an annotation and is associated with a “Ruby Base.” This is typically used in Asian writing systems, providing a helper for uncommonly used writing characters that are not easily recognizable, especially by children. For example, in Japanese writing, the phonetic Hiragana alphabet is used to pair phonetic “helper” readings (called Furigana or Yomigana in Japanese) with the Chinese character counterpart.
RubyAdjust	Specify the ruby text adjustment using the com.sun.star.text.RubyAdjust enumeration: • LEFT – Adjust to the left. • CENTER – Adjust to the center. • RIGHT – Adjust to the right. • BLOCK – Adjust to both borders (stretched). • INDENT_BLOCK – Adjust to both borders with a small indent on both sides.
RubyCharStyleName	Specify the name of the character style that is applied to RubyText.
RubyIsAbove	If True, the Ruby is printed above the text (right if the text is vertical).
CharRotation	Specify the rotation of a character in degrees as a Short Integer. Not all implementations support all values.
CharRotationIsFitToLine	If True, OOo tries to fit the rotated text to the surrounding line height.
CharScaleWidth	Specify the scaling for superscript and subscript as a percentage using a Short Integer.
HyperLinkURL	Specify the URL of a hyperlink (if set) as a String.
HyperLinkTarget	Specify the name of the target for a hyperlink (if set) as a String.
HyperLinkName	Specify the name of the hyperlink (if set) as a String.
VisitedCharStyleName	Specify the character style for visited hyperlinks as a String.
UnvisitedCharStyleName	Specify the character style name for unvisited hyperlinks as a String.
CharEscapementHeight	Specify the additional height used for subscript or superscript characters as an Integer percent. For subscript characters the value is negative.
CharNoHyphenation	If True, the word cannot be hyphenated at the character.
CharUnderlineColor	Specify the color of the underline as a Long Integer.
CharUnderlineHasColor	If True, the CharUnderlineColor is used for an underline.
CharStyleNames	An array of character style names applied to the text. The order is not necessarily relevant.

TIP

When a property supports a DONTKNOW value, the property is usually used as a hint to perform certain operations more efficiently or to find a close replacement value if the requested value is not available. For example, if a particular font is not available, you can use the CharFontFamily to choose a font of the correct type.

The code in Listing 341 demonstrates modifying the character properties by modifying the FontRelief property and then changing it back.

Listing 341. *Set then restore the font relief.*

```
Sub ViewFontRelief
    Dim oEnum 'com.sun.star.container.XEnumerationAccess
    Dim oPar 'Paragraph of some sort
    Dim i% 'General Counting variable
    Dim s$

    oEnum = ThisComponent.Text.createEnumeration()
    Do While oEnum.hasMoreElements()
        oPar = oEnum.nextElement()

        REM The returned paragraph will be a paragraph or a text table
        If oPar.supportsService("com.sun.star.text.Paragraph") Then
            i = i + 1
            oPar.CharRelief = i MOD 3
        End If
    Loop

    MsgBox "The document now uses NONE, EMBOSSED, and ENGRAVED character relief"

    oEnum = ThisComponent.Text.createEnumeration()
    Do While oEnum.hasMoreElements()
        oPar = oEnum.nextElement()

        REM The returned paragraph will be a paragraph or a text table
        If oPar.supportsService("com.sun.star.text.Paragraph") Then
            i = i + 1
            oPar.CharRelief = com.sun.star.text.FontRelief.NONE
        End If
    Loop
End Sub
```

14.2.3. Enumerating text sections (paragraph portions)

It is not unusual for a paragraph to contain text with dissimilar formatting—for example, a single word may be displayed in **bold** in the middle of a sentence that is displayed in the font's normal state. Just as you can enumerate the paragraphs in a document, you can enumerate the text sections in a paragraph. Text within each enumerated portion uses the same properties and is of the same type. Table 129 lists the properties directly supported by the TextPortion service. The

TextPortion service exports the TextRange service so it also supports the paragraph properties in Table 127 and the character properties in Table 128.

TIP An object that supports paragraph or character properties typically provides a way to enclose a range of text. If a specific property changes value in the text range, it usually isn't available to be set. For example, a text range can contain more than one paragraph. If all of the contained paragraphs in the text range do not support the same paragraph style, the paragraph style property is not available to the text range. If, however, the text range is reduced to contain only paragraphs that support a single paragraph style, the paragraph style property will be available to that text range.

Table 129. Properties supported by the *com.sun.text.TextPortion* service.

Property	Description
TextPortionType	String containing the type of the text portion. Valid content type names are: - Text – String content. - TextField – TextField content. - TextContent – Indicates that text content is anchored as or to a character that is not really part of the paragraph—for example, a text frame or a graphic object. As of OOo 1.1.0 and OOo 1.1.1, the type “Frame” is returned rather than “TextContent”. The OOo team refers to this as issue #24444. - Frame – This is not a documented return value, but it is returned rather than the type “TextContent”. - Footnote – Footnote or endnote. - ControlCharacter – Control character. - ReferenceMark – Reference mark. - DocumentIndexMark – Document index mark. - Bookmark – Bookmark. - Redline – Redline portion, which is a result of the change-tracking feature. - Ruby – Ruby attribute (used in Asian text).
ControlCharacter	Short Integer containing the control character if the text portion contains a ControlCharacter.
Bookmark	If the text content is a bookmark, this is a reference to the bookmark. The property implements the <i>com.sun.star.text.XTextContent</i> interface.
IsCollapsed	If True, the text portion is a point.
IsStart	If True, the text portion is a start portion if two portions are needed to include an object. For example, a DocumentIndexMark has a start and end text portion surrounding the text to be indexed.

The macro in Listing 342 demonstrates enumerating the text content inside a paragraph. It displays the paragraph number and the included text portion types in a dialog. The paragraph number is calculated and is not a property of the paragraph.

Listing 342. See text portion types.

```
Sub EnumerateTextSections
    Dim oParEnum          'Paragraph enumerator
    Dim oSecEnum          'Text section enumerator
    Dim oPar              'Current paragraph
```

```

Dim oParSection          'Current section
Dim nPars As Integer     'Number of paragraphs
Dim s$

oParEnum = ThisComponent.Text.createEnumeration()
Do While oParEnum.hasMoreElements()
    oPar = oParEnum.nextElement()

    If oPar.supportsService("com.sun.star.text.Paragraph") Then
        nPars = nPars + 1
        oSecEnum = oPar.createEnumeration()
        s = s & nPars & ":"
        Do While oSecEnum.hasMoreElements()
            oParSection = oSecEnum.nextElement()
            s = s & oParSection.TextPortionType & ":"
        Loop
        s = s & CHR$(10)
        If nPars MOD 10 = 0 Then
            MsgBox s, 0, "Paragraph Text Sections"
            s = ""
        End If
    End If
Loop
MsgBox s, 0, "Paragraph Text Sections"
End Sub

```

14.3. Graphics

The following macro inserts a text graphic object as a link in to the current document. This is a text graphics object, which is inserted at a cursor position. Although you must set the image size, you do not need to set the position.

Listing 343. *Insert a Graphics as a link at the start of the document.*

```

Sub InsertGraphicObject(oDoc, sURL$)
    Dim oCursor
    Dim oGraph
    Dim oText

    oText = oDoc.getText()
    oCursor = oText.createTextCursor()
    oCursor.goToStart(FALSE)
    oGraph = oDoc.createInstance("com.sun.star.text.GraphicObject")

    With oGraph
        .GraphicURL = sURL
        .AnchorType = com.sun.star.text.TextContentAnchorType.AS_CHARACTER
        .Width = 6000
        .Height = 8000
    End With

    'now insert the image into the text document
    oText.insertTextContent( oCursor, oGraph, False )
End Sub

```

You can also insert a graphics object shape, which is inserted into the draw page rather than at a cursor location, but you must use a `com.sun.star.drawing.GraphicObjectShape`, and then set both the location and the size.

Sometimes, you must guess the image size because it is not available. The following method assumes that the argument is a service of type `com.sun.star.graphic.GraphicDescriptor`, which optionally provides the image size in 100th mm and in pixels. The method returns a value in 100th mm, which is what is required for the internal display.

Integers are not used because the intermediate values may be large.

If the size is available in 100th mm, then this is used. Next, the size is checked in Pixels. An image has both a size in pixels, and an expected pixel density (Dots Per Inch). We may have the number of pixels, but we do not have the DPI. I guess the pixel density as the pixel density of the computer display. In other words, if the expected size is not available, then assume that this was created for display on the current monitor.

Listing 344. Guess image size.

```
Function RecommendGraphSize(oGraph)
    Dim oSize
    Dim lMaxW As Double ' Maximum width in 100th mm
    Dim lMaxH As Double ' Maximum height in 100th mm

    lMaxW = 6.75 * 2540 ' 6.75 inches
    lMaxH = 9.5 * 2540 ' 9.5 inches

    If IsNull(oGraph) OR IsEmpty(oGraph) Then
        Exit Function
    End If
    oSize = oGraph.Size100thMM
    If oSize.Height = 0 OR oSize.Width = 0 Then
        ' 2540 is 25.40 mm in an inch, but I need 100th mm.
        ' There are 1440 twips in an inch
        oSize.Height = oGraph.SizePixel.Height * 2540.0 * TwipsPerPixelY() / 1440
        oSize.Width = oGraph.SizePixel.Width * 2540.0 * TwipsPerPixelX() / 1440
    End If
    If oSize.Height = 0 OR oSize.Width = 0 Then
        'oSize.Height = 2540
        'oSize.Width = 2540
        Exit Function
    End If
    If oSize.Width > lMaxW Then
        oSize.Height = oSize.Height * lMaxW / oSize.Width
        oSize.Width = lMaxW
    End If
    If oSize.Height > lMaxH Then
        oSize.Width = oSize.Width * lMaxH / oSize.Height
        oSize.Height = lMaxH
    End If
    RecommendGraphSize = oSize
End Function
```

End Function

An image is embedded as follows:

1. A shape is created and added to the draw page.
2. The graphic provider service is used to obtain image descriptor from disk before it is loaded.
3. The image descriptor is used to guess the image size. The image size is guessed, because we only know what is in the descriptor, and the descriptor may not really know.
4. The shape is set to image as provided by the graphic provider service. At this point, the image is loaded and known by OOo.
5. A newly created graph object sets its URL to the URL used by the shape object. As such, the graphic and the shape should reference the same image.
6. The graph is anchored as a character and then inserted into the document at the cursor.
7. The shape is no longer required, so it is removed.

For reasons I do not understand, all images inserted as a very small images (less than 1 cm). I use the guessed image size to set the graphic size.

Listing 345. Embed an image in a document.

```
' oDoc - document to contain the image.
' oCurs - Cursor where the image is added
' sURL - URL of the image to insert.
' sParStyle - set the paragraph style to this.
Sub EmbedGraphic(oDoc, oCurs, sURL$, sParStyle$)
    Dim oShape
    Dim oGraph      'The graphic object is text content.
    Dim oProvider   'GraphicProvider service.
    Dim oText

    oShape = oDoc.CreateInstance("com.sun.star.drawing.GraphicObjectShape")
    oGraph = oDoc.CreateInstance("com.sun.star.text.GraphicObject")

    oDoc.getDrawPage().add(oShape)

    oProvider = createUnoService("com.sun.star.graphic.GraphicProvider")

    Dim oProps(0) as new com.sun.star.beans.PropertyValue
    oProps(0).Name = "URL"
    oProps(0).Value = sURL

    REM Save the original size.
    Dim oSize100thMM
    Dim lHeight As Long
    Dim lWidth As Long
    oSize100thMM = RecommendGraphSize(oProvider.queryGraphicDescriptor(oProps))
    If NOT IsNull(oSize100thMM) AND NOT IsEmpty(oSize100thMM) Then
```

```

    lHeight = oSize100thMM.Height
    lWidth = oSize100thMM.Width
End If

oShape.Graphic = oProvider.queryGraphic(oProps())
oGraph.graphicurl = oShape.graphicurl
oGraph.AnchorType = com.sun.star.text.TextContentAnchorType.AS_CHARACTER
oText= oCurs.getText()
oText.insertTextContent(oCurs, oGraph, false)
oDoc.getDrawPage().remove(oShape)

If lHeight > 0 AND lWidth > 0 Then
    Dim oSize
    oSize = oGraph.Size
    oSize.Height = lHeight
    oSize.Width = lWidth
    oGraph.Size = oSize
End If

' Set the paragraph style if it is in the document.
Dim oStyles
oStyles = oDoc.StyleFamilies.getByName("ParagraphStyles")
If oStyles.hasByName(sParStyle) Then
    oCurs.ParaStyleName = sParStyle
End If
End Sub

```

14.4. Paste HTML then embed linked graphics

After copying a web page and pasting it into a text document, all of the images are embedded. Graphics in a document contain the GraphicURL property. The URL for a graphic contained in the document begins with the “vnd.sun.star.GraphicObject:”. Graphic inserted using the API are inserted as links – they are not embedded into the document without some work.

The following macro brings many concepts together to find all linked images and convert them into embedded images contained in the document.

Listing 346. Convert all linked images to embedded images.

```

Sub ConvertAllLinkedGraphics(Optional aDoc)
    Dim oDoc          ' Working document
    Dim oDP           ' Draw page
    Dim i%            ' Index counter
    Dim oGraph        ' Graph object in the draw page
    Dim iLinked%      ' Number of linked images
    Dim iEmbedded%    ' Number of embedded images
    Dim iConverted%   ' Linked images converted to embedded
    Dim s1$           ' Graphic service name
    Dim s2$           ' Graphic service name

    REM Only know how to convert these types
    s1 = "com.sun.star.drawing.GraphicObjectShape"
    s2 = "com.sun.star.text.TextGraphicObject"

```

```

If IsMissing(aDoc) OR IsNull(aDoc) OR IsEmpty(aDoc) Then
    oDoc = ThisComponent
Else
    oDoc = aDoc
End If

REM Get the document draw page and then enumerate the images.
oDP = oDoc.getDrawPage()
For i=0 To oDP.getCount()-1
    oGraph = oDP.getByIndex(i)
    If oGraph.supportsService(s1) OR oGraph.supportsService(s2) Then
        If InStr(oGraph.GraphicURL, "vnd.sun") <> 0 Then
            iEmbedded = iEmbedded + 1
        Else
            iLinked = iLinked + 1
            If EmbedLinkedGraphic(oGraph, oDoc) Then
                iConverted = iConverted + 1
            End If
        End If
    End If
Next
Print "Found " & iLinked & " linked and " & iEmbedded & _
      " embedded images and converted " & iConverted
End Sub

Function EmbedLinkedGraphic(oGraph, oDoc) As Boolean
    REM Author: Andrew Pitonyak
    Dim sGraphURL$ ' External URL of the image.
    Dim oGraph_2 ' Created image.
    Dim oCurs ' Cursor where the image is located.
    Dim oText ' Text object containing image.
    Dim oAnchor ' Anchor point of the image
    Dim s1$ ' Graphic service name
    Dim s2$ ' Graphic service name

    EmbedLinkedGraphic = False
    If InStr(oGraph.GraphicURL, "vnd.sun") <> 0 Then
        REM Ignore an image that is already embedded
        Exit Function
    End If
    s1 = "com.sun.star.drawing.GraphicObjectShape"
    s2 = "com.sun.star.text.TextGraphicObject"
    If oGraph.supportsService(s1) Then

        REM Convert a GraphicObjectShape.
        oAnchor = oGraph.getAnchor()
        oText = oAnchor.getText()

        oGraph_2 = ThisComponent.createInstance(s)
        oGraph_2.GraphicObjectFillBitmap = oGraph.GraphicObjectFillBitmap
        oGraph_2.Size = oGraph.Size
        oGraph_2.Position = oGraph.Position
    End If
End Function

```

```

oText.insertTextContent(oAnchor, oGraph_2, False)
oText.removeTextContent(oGraph)
EmbedLinkedGraphic = True

ElseIf oGraph.supportsService(s2) Then

    REM Convert a TextGraphicObject.
    Dim oBitmaps
    Dim sNewURL$
    Dim sName$

    sName$ = oGraph.LinkDisplayName
    oBitmaps = oDoc.createInstance( "com.sun.star.drawing.BitmapTable" )
    If oBitmaps.hasByName(sName) Then
        Print "Link display name " & sName & " already exists"
        Exit Function
    End If
    'Print "Ready to insert " & sName
    oBitmaps.insertByName( sName, oGraph.GraphicURL )
    sNewURL$ = oBitmaps.getByName( sName )
    'Print "inserted URL " & sNewURL
    oGraph.GraphicURL = sNewURL
    EmbedLinkedGraphic = True
End If
End Function

```

14.5. Cursors

The capability to enumerate the entire text content is primarily used for tasks such as exporting a document, because exporting requires all of the content to be accessed in sequential order. A more typical method for accessing and manipulating a document involves the use of cursors. A `TextCursor` is a `TextRange`, which can be moved within a `Text` object. In other words, a text cursor not only can specify a single point in the text, but it also can encompass a range of text. You can use the cursor movement methods to reposition the cursor and to expand the section of selected text. Two primary types of text cursors are available in OOO: cursors that are view cursors (see Table 130) and cursors that are not (see Table 132).

14.5.1. View cursors

As its name implies, the view cursor deals with the visible cursor. In a single document window, you see one view at a time. Analogously, you can have one view cursor at a time. View cursors support commands that are directly related to viewing. To move a cursor one line at a time, or one screen at a time, requires a view cursor. The view cursor knows how the text is displayed (see Table 131).

Table 130. In general, view cursors are not related to text ranges or *XTextCursor*.

Cursor	Description
com.sun.star.view.XViewCursor	Simple cursor with basic movement methods that work in both text and tables.
com.sun.star.text.XTextViewCursor	Derived from <i>XTextCursor</i> , this describes a cursor in a text document's view. It supports only very simple movements.
com.sun.star.view.XLineCursor	Defines line-related methods; this interface is not derived from a text range.
com.sun.star.text.XPageCursor	Defines page-related methods; this interface is not derived from a text range.
com.sun.star.view.XScreenCursor	Defines methods to move up and down one screen at a time.

Most of the cursor movement methods accept a Boolean argument that determines whether the text range of the cursor is expanded (True) or whether the cursor is simply moved to the new position (False). In other words, if the Boolean expression is False, the cursor is moved to the new position and no text is selected by the cursor. The description of the Boolean variable is assumed and not explicitly stated for each of the cursor movement methods in Table 131. Another commonality with the movement methods is that they return a Boolean value. A True value means that the movement worked, and a False value means that the movement failed. A movement request fails if it cannot be completed successfully. It is not possible, for example, to move the cursor down if it is already at the end of the document. The screen cursor is obtained from the document's current controller (see Listing 347).

Table 131. Object methods related to view cursors.

Defined	Method	Description
XViewCursor	goDown(n, Boolean)	Move the cursor down by n lines.
XViewCursor	goUp(n, Boolean)	Move the cursor up by n lines.
XViewCursor	goLeft(n, Boolean)	Move the cursor left by n characters.
XViewCursor	goRight(n, Boolean)	Move the cursor right by n characters.
XTextViewCursor	isVisible()	Return True if the cursor is visible.
XTextViewCursor	setVisible(Boolean)	Show or hide the cursor.
XTextViewCursor	getPosition()	Return a com.sun.star.awt.Point structure specifying the cursor's coordinates relative to the top-left position of the first page of the document.
XLineCursor	isAtStartOfLine()	Return True if the cursor is at the start of the line.
XLineCursor	isAtEndOfLine()	Return True if the cursor is at the end of the line.
XLineCursor	gotoEndOfLine(Boolean)	Move the cursor to the end of the current line.
XLineCursor	gotoStartOfLine(Boolean)	Move the cursor to the start of the current line.
XPageCursor	jumpToFirstPage()	Move the cursor to the first page.
XPageCursor	jumpToLastPage()	Move the cursor to the last page.

Defined	Method	Description
XPageCursor	jumpToPage(n)	Move the cursor to the specified page.
XPageCursor	getPage()	Return the current page as a Short Integer.
XPageCursor	jumpToNextPage()	Move the cursor to the next page.
XPageCursor	jumpToPreviousPage()	Move the cursor to the previous page.
XPageCursor	jumpToEndOfPage()	Move the cursor to the end of the current page.
XPageCursor	jumpToStartOfPage()	Move the cursor to the start of the current page.
XScreenCursor	screenDown()	Scroll the view forward by one visible page.
XScreenCursor	screenUp()	Scroll the view backward by one visible page.

Listing 347. *ScrollDownOneScreen* is found in the *Writer* module in this chapter’s source code files as *SC13.sxw*.

```
Sub ScrollDownOneScreen
    REM Get the view cursor from the current controller
    ThisComponent.currentController.getViewCursor().screenDown()
End Sub
```

A more typical use of the view cursor is to insert some special text content at the current cursor position. The macro in Listing 348 inserts the character with Unicode value 257 (“a” with a bar over it) at the current cursor position. This type of macro is typically associated with a keystroke to insert special characters that are not present on the keyboard. The macro in Listing 348 is short and simple, yet very useful.

Listing 348. *Insert the character with Unicode value 257 at the current cursor.*

```
Sub InsertControlCharacterAtCurrentCursor
    Dim oViewCursor As Object
    oViewCursor = ThisComponent.CurrentController.getViewCursor()
    oViewCursor.getText.insertString(oViewCursor.getStart(), CHR$(257), False)
End Sub
```

14.5.2. Text (non-view) cursors

The view cursor knows how the data is displayed, but doesn’t know about the data itself. Text cursors (non-view cursors), however, know a lot about the data but very little about how it is displayed. For example, view cursors do not know about words or paragraphs, and text cursors do not know about lines, screens, or pages (see Table 132).

Table 132. *Text cursor interfaces all implement the XTextCursor interface.*

Cursor	Description
com.sun.star.text.XTextCursor	The primary text cursor that defines simple movement methods.
com.sun.star.text.XWordCursor	Provides word-related movement and testing methods.
com.sun.star.text.XSentenceCursor	Provides sentence-related movement and testing methods.
com.sun.star.text.XParagraphCursor	Provides paragraph-related movement and testing methods.
com.sun.star.text.XTextViewCursor	Derived from XTextCursor, this describes a cursor in a text

Cursor	Description
	document's view.

TIP Text cursors and view cursors have some overlap. The `XTextViewCursor` is derived from the `XTextCursor` so it supports the `XTextCursor` methods. This does not provide functionality with respect to the underlying data such as word-related or paragraph-related methods (see Table 133).

The word cursor, sentence cursor, and paragraph cursor all define essentially identical object methods (see Table 133). The `XTextViewCursor` interface is listed in Table 131 so it is omitted from Table 133.

Table 133. Object methods related to text cursors.

Defined	Method	Description
XTextCursor	<code>collapseToStart()</code>	Set the end position to the start position.
XTextCursor	<code>collapseToEnd()</code>	Set the start position to the end position.
XTextCursor	<code>isCollapsed()</code>	Return True if the start and end positions are the same.
XTextCursor	<code>goLeft(n, Boolean)</code>	Move the cursor left by n characters.
XTextCursor	<code>goRight(n, Boolean)</code>	Move the cursor right by n characters.
XTextCursor	<code>gotoStart(Boolean)</code>	Move the cursor to the start of the text.
XTextCursor	<code>gotoEnd(Boolean)</code>	Move the cursor to the end of the text.
XTextCursor	<code>gotoRange(XTextRange, Boolean)</code>	Move or expand the cursor to the TextRange.
XWordCursor	<code>isStartOfWord()</code>	Return True if at the start of a word.
XWordCursor	<code>isEndOfWord()</code>	Return True if at the end of a word.
XWordCursor	<code>gotoNextWord(Boolean)</code>	Move to the start of the next word.
XWordCursor	<code>gotoPreviousWord(Boolean)</code>	Move to the end of the previous word.
XWordCursor	<code>gotoEndOfWord(Boolean)</code>	Move to the end of the current word.
XWordCursor	<code>gotoStartOfWord(Boolean)</code>	Move to the start of the current word.
XSentenceCursor	<code>isStartOfSentence()</code>	Return True if at the start of a sentence.
XSentenceCursor	<code>isEndOfSentence()</code>	Return True if at the end of a sentence.
XSentenceCursor	<code>gotoNextSentence(Boolean)</code>	Move to the start of the next sentence.
XSentenceCursor	<code>gotoPreviousSentence(Boolean)</code>	Move to the end of the previous sentence.
XSentenceCursor	<code>gotoEndOfSentence(Boolean)</code>	Move to the end of the current sentence.
XSentenceCursor	<code>gotoStartOfSentence(Boolean)</code>	Move to the start of the current sentence.
XParagraphCursor	<code>isStartOfParagraph()</code>	True if at the start of a paragraph.
XParagraphCursor	<code>isEndOfParagraph()</code>	True if at the end of a paragraph.
XParagraphCursor	<code>gotoNextParagraph(Boolean)</code>	Move to the start of the next paragraph.
XParagraphCursor	<code>gotoPreviousParagraph(Boolean)</code>	Move to the end of the previous paragraph.

Defined	Method	Description
XParagraphCursor	gotoEndOfParagraph(Boolean)	Move to the end of the current paragraph.
XParagraphCursor	gotoStartOfParagraph(Boolean)	Move to the start of the current paragraph.

14.5.3. Using cursors to traverse text

Although traversing text using a cursor is not inherently difficult, I struggled with cursors for a long time before I finally realized that I had a very basic and yet simple misunderstanding. A very common, and yet subtly incorrect, method of traversing text content using cursors is shown in Listing 349. This macro attempts to move the cursor from one paragraph to the next, selecting one paragraph at a time. Typically, something would be done to manipulate or modify the paragraph, such as setting the paragraph style.

***Listing 349.** Example of incorrect use of cursors: This code misses the last paragraph in the document.*

```
Dim oCursor
REM Create a text cursor
oCursor = ThisComponent.Text.createTextCursor()
REM Start at the beginning of the document.
REM This is the same as the start of the first document.
oCursor.gotoStart(False)
REM And this is where things go wrong!
REM The cursor now spans from the start of the
REM first paragraph to the start of the second paragraph.
Do While oCursor.gotoNextParagraph(True)
    REM Process the paragraph here!
    REM Now, deselect all of the text, leaving the cursor at the
    REM start of the next paragraph.
    oCursor.goRight(0, False)
Loop
```

TIP

I produced incorrect code as shown in Listing 349 before I understood cursors.

The problem in Listing 349 is that the method `gotoNextParagraph(True)` causes the cursor to extend its selection from the start of one paragraph to the start of the next. The first problem with selecting from the start of one paragraph to the start of the next is that more than just one paragraph is selected. If the two different paragraphs do not have the same paragraph style, the `ParaStyleName` property will return an empty string rather than a value. The second problem is that when the cursor (as shown in Listing 349) is positioned at the start of the last paragraph, it is not possible to go to the next paragraph because the next paragraph does not exist. Therefore, the statement “`gotoNextParagraph(True)`” returns `False` and the last paragraph is never processed. The code in Listing 350 demonstrates one correct method of traversing all of the paragraphs using a cursor.

***Listing 350.** The correct way to use cursors.*

```
Dim oCursor
```

```

REM Create a text cursor.
oCursor = ThisComponent.Text.createTextCursor()
REM Start at the beginning of the document.
REM This is the same as the start of the first document.
oCursor.gotoStart(False)
Do
    REM The cursor is already positioned at the start
    REM of the current paragraph, so select the entire paragraph.
    oCursor.gotoEndOfParagraph(True)
    REM Process the paragraph here!
    REM The Loop statement moves to the next paragraph and
    REM it also deselects the text in the cursor.
Loop While oCursor.gotoNextParagraph(False)

```

The point is to align the cursor over the current paragraph, and then iterate aligned with the paragraphs, rather than ever looking ahead to the next paragraph while in the midst of dealing with the cursor.

TIP You can use cursors and enumeration to traverse an entire document and obtain all of the paragraphs. Using a cursor is five times faster than using an enumerator.

It is very simple to write an iterator that traverses the text while counting the words, sentences, and paragraphs (see Listing 351).

Listing 351. Count paragraphs, sentences, and words.

```

Sub CountWordSentPar
    Dim oCursor
    Dim nPars As Long
    Dim nSentences As Long
    Dim nWords As Long

    REM Create a text cursor.
    oCursor = ThisComponent.Text.createTextCursor()
    oCursor.gotoStart(False)
    Do
        nPars = nPars + 1
    Loop While oCursor.gotoNextParagraph(False)
    oCursor.gotoStart(False)
    Do
        nSentences = nSentences + 1
    Loop While oCursor.gotoNextSentence(False)

    oCursor.gotoStart(False)
    Do
        nWords = nWords + 1
    Loop While oCursor.gotoNextWord(False)

    MsgBox "Paragraphs: " & nPars & CHR$(10) & _
        "Sentences: " & nSentences & CHR$(10) & _
        "Words: " & nWords & CHR$(10), 0, "Doc Statistics"
End Sub

```

TIP

With OOo 1.1, I found gotoNextSentence() and gotoNextWord() to be unreliable, but the paragraph cursor worked well. To count words, therefore, I used the word count macros from Andrew Browns: http://www.darwinwars.com/lunatic/bugs/oo_macros.html.

Keep the view cursor and text cursor in sync

A paragraph is broken into lines based on the page margins. Adjust the page margins and the line breaks change. Also, text copied from writer as text do not contain the page breaks. For the real world example in Listing 352 adds real line breaks into the text based on how the lines break on screen.

Both a view cursor and a non-view cursor are required to move through the text. The view cursor knows where a line ends, but it does not know where the paragraph starts or ends. A non-view cursor knows about paragraphs, but the view cursor does not.

Listing 352 moves the view cursor around on the screen, demonstrating how to change its location.

Listing 352. Add line breaks in a paragraph.

```
Sub LineBreaksInParagraph
    Dim oText          'Save typing ThisComponent.Text
    Dim oViewCursor     'Save typing
    ThisComponent.CurrentController.getViewCursor()
    Dim oTextCursor     'Created text cursor
    Dim oSaveCursor     'In case I want to restore the view cursor

    oText = ThisComponent.Text

    REM You require a view cursor
    REM because only the view knows where a line ends.
    oViewCursor = ThisComponent.CurrentController.getViewCursor()

    REM You require a text cursor so that you know where the paragraph ends.
    REM Too bad the view cursor is not a paragraph cursor.
    oTextCursor = oText.createTextCursorByRange(oViewCursor)

    REM You only need this if you want to restore the view cursor
    oSaveCursor = oText.createTextCursorByRange(oViewCursor)

    REM Move the cursor to the start of the current paragraph
    REM so that the entire paragraph can be processed.
    If NOT oTextCursor.isStartOfParagraph() Then
        oTextCursor.gotoStartOfParagraph(False)
        oViewCursor.gotoRange(oTextCursor, False)
    End If
    REM Now visit each line of the paragraph.
    Do While True
        REM Only the view cursor knows where the end of the line is
        REM because this is a formatting issue and not determined by
```

```

REM punctuation.
oViewCursor.gotoEndOfLine(false)

REM Move with the view cursor to the end of the current line
REM and then see if you're at the end of the paragraph.
oTextCursor.gotoRange(oViewCursor, False)

REM Check for end of paragraph BEFORE inserting the line break.
If oTextCursor.isEndOfParagraph() Then Exit Do

REM Insert a line break at the current view cursor.
oText.insertControlCharacter(oViewCursor, _
    com.sun.star.text.ControlCharacter.LINE_BREAK, false)
Loop
REM If you only want to move the cursor away from the end of the
REM current paragraph, then this will suffice.
oViewCursor.goRight(1, False)
REM I want to restore the view cursor location, however.
REM oViewCursor.gotoRange(oSaveCursor, False)
End Sub

```

14.5.4. Accessing content using cursors

I have a habit of inspecting the objects returned to me by OOo. While inspecting a view cursor, I noticed that the view cursor contained useful undocumented properties. According to the developers at Sun, these will eventually be documented. Some of the properties that I noticed include: Cell, DocumentIndex, DocumentIndexMark, Endnote, Footnote, ReferenceMark, Text, TextField, TextFrame, TextSection, and TextTable. If a cursor is in a text table, the cursor's TextTable property is not empty. If a cursor is on a text field, the cursor's TextField property is not empty. These special properties are empty if they are not relevant. I first used these undocumented properties when I was asked how to determine if the view cursor was positioned in a text table, and if so, what cell contains the view cursor. See Listing 353.

Listing 353. Testing view cursor properties.

```

If NOT IsEmpty(oViewCursor.TextTable) Then

```

Cursors provide the ability to quickly find objects in close proximity to something else. For example, I was recently asked how to find a text field contained in the current paragraph—the current paragraph is defined as the paragraph containing the view cursor. It's easy to obtain the view cursor, and you can use a paragraph cursor to move around the current paragraph.

My first attempt called the createContentEnumeration("com.sun.star.text.TextContent") object method on the cursor. This creates an enumeration of text content, enumerating objects such as inserted buttons. I had mistakenly assumed that this would include text fields in the enumeration. My second attempt to find a text field, which was successful, uses the createEnumeration() object method. The createEnumeration() method returns an enumeration of the paragraphs contained in the cursor. Enumerating the content contained in a paragraph provides access to the text field. My final attempt, which was also successful, moves the cursor to the start of the document and then

moves through the paragraph one location at a time looking for a text field. The macro in Listing 354 demonstrates all of the methods that I used to try to find a text field in the current paragraph.

Listing 354. Check current paragraph for a text field.

```
Sub TextFieldInCurrentParagraph
    Dim oEnum          'Cursor enumerator
    Dim oSection        'Current section
    Dim oViewCursor     'Current view cursor
    Dim oTextCursor     'Created text cursor
    Dim oText           'Text object in current document
    Dim s$
    Dim sTextContent$ 'Service name for text content.
    sTextContent = "com.sun.star.text.TextContent"

    oText = ThisComponent.Text
    oViewCursor = ThisComponent.CurrentController.getViewCursor()

    REM Obtain the view cursor, and then select the paragraph
    REM containing the view cursor.
    oTextCursor = oText.createTextCursorByRange(oViewCursor)

    REM Move to the start of the paragraph as a single point.
    oTextCursor.gotoStartOfParagraph(False)
    REM Move to the end of the current paragraph and expand the
    REM selection so that the entire paragraph is selected.
    oTextCursor.gotoEndOfParagraph(True)

    REM I want to enumerate the text content contained by this text cursor.
    REM Although this will find inserted drawing objects, such as the shape
    REM used by a button, it will not find a text field!
    oEnum = oTextCursor.createContentEnumeration(sTextContent)
    Do While oEnum.hasMoreElements()
        oSection = oEnum.nextElement()
        Print "Enumerating TextContent: " & oSection.ImplementationName
    Loop

    REM And this enumerates the paragraphs that are contained in
    REM the text cursor.
    oEnum = oTextCursor.createEnumeration()
    Dim v
    Do While oEnum.hasMoreElements()
        v = oEnum.nextElement()
        Dim oSubSection
        Dim oSecEnum

        REM Now create an enumeration of the paragraphs.
        REM We can enumerate the sections of the paragraph to
        REM obtain the text field and other paragraph content.
        oSecEnum = v.createEnumeration()
        s = "Enumerating section type: " & v.ImplementationName
        Do While oSecEnum.hasMoreElements()
            oSubSection = oSecEnum.nextElement()
```



```

s = s & CHR$(10) & oSubSection.TextPortionType
If oSubSection.TextPortionType = "TextField" Then
    s = s & " <== Type " & oSubSection.TextField.ImplementationName
End If
Loop
MsgBox s, 0, "Enumerate Single Paragraph"
Loop

REM And this is yet another way to find the text field.
REM Start at the beginning of the paragraph and then move the cursor
REM through it, looking for text fields.
oTextCursor.gotoStartOfParagraph(False)
Do While oTextCursor.goRight(1, False) AND _
    NOT oTextCursor.isEndOfParagraph()
    If NOT IsEmpty(oTextCursor.TextField) Then
        Print "It is NOT empty, you can use the text field"
    End If
Loop
End Sub

```

TIP

As unintuitive as it sounds, it is not only possible—but common—for the end of a text range to come before the start of a text range. The order of the start and end is primarily an issue when dealing with text selected by a human user but may also occur due to the movement of a text cursor while expanding the text range.

As the object methods `getStart()` and `getEnd()` imply, it's possible for a text range to represent a single point. It's also possible that the start position comes after the end position. Unexpected start and end positions are typically a problem while dealing with selected text. When you select text using a mouse or your keyboard, the initial selection point is generally the start location. Moving the final selection point toward the start of the document causes the end position to occur before the start position in the text range. The same behavior may occur while manually moving and expanding a cursor. This behavior is not documented, but it has been observed in OOo 1.1.0, and like all undocumented behavior, it may change at any time. The text object can compare two ranges (see Table 134), but the text object must contain both text ranges—a text range has the `getText()` object method to return the text object that contains the text range.

Table 134. *Methods defined by the `com.sun.star.text.XTextRangeCompare` interface.*

Method	Description
<code>compareRegionStarts(XTextRange, XTextRange)</code>	- Return 1 if the first range starts before the second. - Return 0 if the first range starts at the same position as the second. - Return -1 if the first range starts after the second.
<code>compareRegionEnds(XTextRange, XTextRange)</code>	- Return 1 if the first range ends before the second. - Return 0 if the first range ends at the same position as the second. - Return -1 if the first range ends after the second.

14.6. Selected text

Selected text refers to the text that has been selected by a user, probably using the keyboard or the mouse. Selected text is represented as nothing more than a text range. After you find a text selection, it's possible to get the text [getString()] and set the text [setString()]. Although strings are limited to 64KB in size, selections are not. There are some instances, therefore, when you can't use the getString() and setString() methods. Therefore, it's probably better to use a cursor to traverse the selected text and then use the cursor's text object to insert text content. Most problems using selected text look the same at an abstract level.

```
If nothing is selected then
  do work on entire document
else
  for each selected area
    do work on selected area
```

The difficult part that changes each time is writing a macro that iterates over a selection or between two text ranges.

14.6.1. Is text selected?

Text documents support the XTextSectionsSupplier interface (see Table 123), which defines the single method getCurrentSelection(). If there is no current controller (which means that you're an advanced user running OpenOffice.org as a server with no user interface and you won't be looking for selected text anyway), getCurrentSelection() returns a null rather than any selected text.

If the selection count is zero, nothing is selected. I have never seen a selection count of zero, but I check for it anyway. If no text is selected, there is one selection of zero length—the start and end positions are the same. I have seen examples, which I consider unsafe, where a zero-length selection is determined as follows:

```
If Len(oSel.getString()) = 0 Then nothing is selected
```

It is possible that selected text will contain more than 64KB characters, and a string cannot contain more than 64KB characters. Therefore, don't check the length of the selected string to see if it is zero; this is not safe. The better solution is to create a text cursor from the selected range and then check to see if the start and end points are the same.

```
oCursor = oDoc.Text.CreateTextCursorByRange(oSel)
If oCursor.IsCollapsed() Then nothing is selected
```

The macro function in Listing 355 performs the entire check, returning True if something is selected, and False if nothing is selected.

Listing 355. Determine if anything is selected.

```
Function IsAnythingSelected(oDoc As Object) As Boolean
  Dim oSelections 'Contains all of the selections
  Dim oSel        'Contains one specific selection
  Dim oCursor      'Text cursor to check for a collapsed range
  REM Assume nothing is selected
```

```

IsAnythingSelected = False
If IsNull(oDoc) Then Exit Function
' The current selection in the current controller.
'If there is no current controller, it returns NULL.
oSelections = oDoc.GetCurrentSelection()
If IsNull(oSelections) Then Exit Function
If oSelections.getCount() = 0 Then Exit Function
If oSelections.getCount() > 1 Then
    REM There is more than one selection so return True
    IsAnythingSelected = True
Else
    REM There is only one selection so obtain the first selection
    oSel = oSelections.getByIndex(0)

    REM Create a text cursor that covers the range and then see if it is
    REM collapsed.
    oCursor = oDoc.Text.CreateTextCursorByRange(oSel)
    If Not oCursor.IsCollapsed() Then IsAnythingSelected = True

    REM You can also compare these to see if the selection starts and ends at
    REM the same location.
    REM If oDoc.Text.compareRegionStarts(oSel.getStart(), _
    REM     oSel.getEnd()) <> 0 Then
    REM     IsAnythingSelected = True
    REM End If
End If
End Function

```

Obtaining a selection is complicated because it's possible to have multiple non-contiguous selections. Some selections are empty and some are not. If you write code to handle text selection, it should handle all of these cases because they occur frequently. The example in Listing 356 iterates through all selected sections and displays them in a message box.

Listing 356. Display selected text.

```

Sub PrintMultipleTextSelection
    Dim oSelections           'Contains all of the selections
    Dim oSel                  'Contains one specific selection
    Dim lWhichSelection As Long 'Which selection to print

    If NOT IsAnythingSelected(ThisComponent) Then
        Print "Nothing is selected"
    Else
        oSelections = ThisComponent.GetCurrentSelection()
        For lWhichSelection = 0 To oSelections.getCount() - 1
            oSel = oSelections.getByIndex(lWhichSelection)
            MsgBox oSel.getString(), 0, "Selection " & lWhichSelection
        Next
    End If
End Sub

```

14.6.2. Selected text: Which end is which?

Selections are text ranges with a start and an end. Although selections have both a start and an end, which side of the text is which is determined by the selection method. For example, position the cursor in the middle of a line, and then select text by moving the cursor either right or left. In both cases, the start position is the same. In one of these cases, the start position is after the end position. The text object provides methods to compare starting and ending positions of text ranges (see Table 134). I use the two methods in Listing 357 to find the leftmost and rightmost cursor position of selected text.

Listing 357. *Get left and right cursor.*

```
'oSel is a text selection or cursor range
'oText is the text object
Function GetLeftMostCursor(oSel, oText)
    Dim oRange
    Dim oCursor

    If oText.compareRegionStarts(oSel.getEnd(), oSel) >= 0 Then
        oRange = oSel.getEnd()
    Else
        oRange = oSel.getStart()
    End If
    oCursor = oText.CreateTextCursorByRange(oRange)
    oCursor.goRight(0, False)
    GetLeftMostCursor = oCursor
End Function

'oSel is a text selection or cursor range
'oText is the text object
Function GetRightMostCursor(oSel, oText)
    Dim oRange
    Dim oCursor

    If oText.compareRegionStarts(oSel.getEnd(), oSel) >= 0 Then
        oRange = oSel.getStart()
    Else
        oRange = oSel.getEnd()
    End If
    oCursor = oText.CreateTextCursorByRange(oRange)
    oCursor.goLeft(0, False)
    GetRightMostCursor = oCursor
End Function
```

While using text cursors to move through a document, I noticed that cursors remember the direction in which they are traveling. The cursors returned by the macros in Listing 357 are oriented to travel into the text selection by moving the cursor left or right zero characters. This is also an issue while moving a cursor to the right and then turning it around to move left. I always start by moving the cursor zero characters in the desired direction before actually moving the cursor. Then my macro can use these cursors to traverse the selected text from the start (moving right) or the end (moving left).

14.6.3. Selected text framework

While dealing with selected text, I use a framework that returns a two-dimensional array of start and end cursors over which to iterate. Using a framework allows me to use a very minimal code base to iterate over selected text or the entire document. If no text is selected, the framework asks if the macro should use the entire document. If the answer is yes, a cursor is created at the start and the end of the document. If text is selected, each selection is retrieved, and a cursor is obtained at the start and end of each selection. See Listing 358.

Listing 358. *Create cursors with selected areas.*

```
'sPrompt : How to ask if should iterate over the entire text
'oCursors() : Has the return cursors
'Returns True if should iterate and False if should not
Function CreateSelectedTextIterator(oDoc, sPrompt$, oCursors()) As Boolean
    Dim oSelections                'Contains all of the selections
    Dim oSel                       'Contains one specific selection
    Dim oText                      'Document text object
    Dim lSelCount As Long          'Number of selections
    Dim lWhichSelection As Long    'Current selection
    Dim oLCursor, oRCursor         'Temporary cursors
    CreateSelectedTextIterator = True
    oText = oDoc.Text
    If Not IsAnythingSelected(oDoc) Then
        Dim i%
        i% = MsgBox("No text selected!" + CHR$(13) + sPrompt, _
            1 OR 32 OR 256, "Warning")
        If i% = 1 Then
            oLCursor = oText.createTextCursorByRange(oText.getStart())
            oRCursor = oText.createTextCursorByRange(oText.getEnd())
            oCursors = DimArray(0, 1) 'Two-Dimensional array with one row
            oCursors(0, 0) = oLCursor
            oCursors(0, 1) = oRCursor
        Else
            oCursors = DimArray()      'Return an empty array
            CreateSelectedTextIterator = False
        End If
    Else
        oSelections = oDoc.getCurrentSelection()
        lSelCount = oSelections.getCount()
        oCursors = DimArray(lSelCount - 1, 1)
        For lWhichSelection = 0 To lSelCount - 1
            oSel = oSelections.getByIndex(lWhichSelection)
            oLCursor = GetLeftMostCursor(oSel, oText)
            oRCursor = GetRightMostCursor(oSel, oText)
            oCursors(lWhichSelection, 0) = oLCursor
            oCursors(lWhichSelection, 1) = oRCursor
        Next
    End If
End Function
```

TIP

The argument oCursors() is an array that is set in the macro in Listing 358.

The macro in Listing 359 uses the selected text framework to print the Unicode values of the selected text.

Listing 359. *Display Unicode of the selected text.*

```
Sub PrintUnicodeExamples
    Dim oCursors() As Integer
    If Not CreateSelectedTextIterator(ThisComponent, _
        "Print Unicode for the entire document?", oCursors()) Then Exit Sub
    For i% = LBound(oCursors()) To UBound(oCursors())
        PrintUnicode_worker(oCursors(i%), 0), oCursors(i%), 1, ThisComponent.Text
    Next i%
End Sub

Sub PrintUnicode_worker(oLCursor As Integer, oRCursor As Integer, oText As String)
    Dim s As String 'contains the primary message string
    Dim ss As String 'used as a temporary string

    If IsNull(oLCursor) Or IsNull(oRCursor) Or IsNull(oText) Then Exit Sub
    If oText.CompareRegionEnds(oLCursor, oRCursor) <= 0 Then Exit Sub

    REM Start the cursor in the correct direction with no text selected
    oLCursor.GoRight(0, False)
    Do While oLCursor.GoRight(1, True) _
        AND oText.CompareRegionEnds(oLCursor, oRCursor) >= 0
        ss = oLCursor.GetString()
        REM The string may be empty
        If Len(ss) > 0 Then
            s = s & oLCursor.GetString() & "=" & ASC(oLCursor.GetString()) & " "
        End If
        oLCursor.GoRight(0, False)
    Loop
    MsgBox s, 0, "Unicode Values"
End Sub
```

14.6.4. Remove empty spaces and lines: A larger example

A common request is for a macro that removes extra blank spaces. To remove all empty paragraphs, it's better to use the Remove Blank Paragraphs option from the AutoCorrect dialog (**Tools | AutoCorrect Options | Options**). To remove only selected paragraphs or runs of blank space, a macro is required.

This section presents a set of macros that replaces all runs of white-space characters with a single white-space character. You can easily modify this macro to delete different types of white space. The different types of space are ordered by importance, so if you have a regular space followed by a new paragraph, the new paragraph stays and the single space is removed. The end effect is that leading and trailing white space is removed from each line.

What is white space?

The term “white space” typically refers to any character that is displayed as a blank space. This includes tabs (ASCII value 9), regular spaces (ASCII value 32), non-breaking spaces (ASCII value 160), new paragraphs (ASCII value 13), and new lines (ASCII value 10). By encapsulating the definition of white space into a function (see Listing 360), you can trivially change the definition of white space to ignore certain characters.

Listing 360. Determine if a character is white space.

```
Function IsWhiteSpace(iChar As Integer) As Boolean
    Select Case iChar
        Case 9, 10, 13, 32, 160
            IsWhiteSpace = True
        Case Else
            IsWhiteSpace = False
    End Select
End Function
```

If time is a serious issue, it is faster to create an array that is indexed by the Unicode value that contains true or false to indicate if a character is white space. The problem is that the array must be set before the call.

```
For i = LBound(isWhiteArray) To UBound(isWhiteArray)
    isWhiteArray(i) = False
Next
isWhiteArray(9) = True
isWhiteArray(10) = True
isWhiteArray(13) = True
isWhiteArray(32) = True
isWhiteArray(160) = True
```

The function to test is as follows:

```
Function IsWhiteSpace2(iChar As Integer) As Boolean
    If iChar > UBound(isWhiteArray) Then
        IsWhiteSpace2 = False
    Else
        IsWhiteSpace2 = isWhiteArray(iChar)
    End If
End Function
```

Rank characters for deletion

While removing runs of white space, each character is compared to the character before it. If both characters are white space, the less important character is deleted. For example, if there is both a space and a new paragraph, the space is deleted. The RankChar() function (see Listing 361) accepts two characters: the previous character and the current character. The returned integer indicates which, if any, character should be deleted.

Listing 361. Rank characters for deletion.

```
'-1 means delete the previous character
' 0 means ignore this character
```

```

' 1 means delete this character
' If an input character is 0, this is the start of a line.
' Rank from highest to lowest is: 0, 13, 10, 9, 160, 32
Function RankChar(iPrevChar, iCurChar) As Integer
    If Not IsWhiteSpace(iCurChar) Then 'Current not white space, ignore it
        RankChar = 0
    ElseIf iPrevChar = 0 Then 'Line start, current is white space
        RankChar = 1 ' delete the current character.
    ElseIf Not IsWhiteSpace(iPrevChar) Then 'Current is space but not previous
        RankChar = 0 ' ignore the current character.

    REM At this point, both characters are white space
    ElseIf iPrevChar = 13 Then 'Previous is highest ranked space
        RankChar = 1 ' delete the current character.
    ElseIf iCurChar = 13 Then 'Current is highest ranked space
        RankChar = -1 ' delete the previous character.

    REM Neither character is a new paragraph, the highest ranked
    ElseIf iPrevChar = 10 Then 'Previous is new line
        RankChar = 1 ' delete the current character.
    ElseIf iCurChar = 10 Then 'Current is new line
        RankChar = -1 ' delete the previous character.

    REM At this point, the highest ranking possible is a tab
    ElseIf iPrevChar = 9 Then 'Previous char is tab
        RankChar = 1 ' delete the current character.
    ElseIf iCurChar = 9 Then 'Current char is tab
        RankChar = -1 ' delete the previous character.
    ElseIf iPrevChar = 160 Then 'Previous char is a hard space
        RankChar = 1 ' delete the current character.
    ElseIf iCurChar = 160 Then 'Current char is a hard space
        RankChar = -1 ' delete the previous character.
    ElseIf iPrevChar = 32 Then 'Previous char is a regular space
        RankChar = 1 ' delete the current character.

    REM Probably should never get here... both characters are white space
    REM and the previous is not any known white space character.
    ElseIf iCurChar = 32 Then 'Current char is a regular space
        RankChar = -1 ' delete the previous character.
    Else 'Should probably not get here
        RankChar = 0 'so simply ignore it!
    End If
End Function

```

Use the standard framework

The standard framework is used to remove the empty spaces. The primary routine is simple enough that it barely warrants mentioning.

Listing 362. Remove empty space.

```

Sub RemoveEmptySpace
    Dim oCursors(), i%
    If Not CreateSelectedTextIterator(ThisComponent, _

```



```

    "ALL empty space will be removed from the ENTIRE document?", oCursors()) Then Exit Sub
For i% = LBOUND(oCursors()) To UBOUND(oCursors())
    RemoveEmptySpaceWorker (oCursors(i%, 0), oCursors(i%, 1), ThisComponent.Text)
Next i%
End Sub

```

The worker macro

The macro in Listing 363 represents the interesting part of this problem; it decides what is deleted and what is left untouched. Some interesting points should be noted:

Because a text cursor is used, the formatting is left unchanged.

A text range (cursor) may contain text content that returns a zero-length string. This includes, for example, buttons and graphic images contained in the document. Handling exceptional cases adds complexity to the macro. Many tasks are very simple if you ignore the exceptional cases, such as inserted graphics. If you know that your macro will run with simple controlled data, you may choose to sacrifice robustness to reduce complexity. Listing 363 handles the exceptional cases.

If the selected text starts or ends with white space, it will be removed even if it does not start or end the document.

Listing 363. RemoveEmptySpaceWorker.

```

Sub RemoveEmptySpaceWorker(oLCursor, oRCursor, oText)
    Dim s As String          'Temporary text string
    Dim i As Integer         'Temporary integer used for comparing text ranges
    Dim iLastChar As Integer 'Unicode of last character
    Dim iThisChar As Integer 'Unicode of the current character
    Dim iRank As Integer     'Integer ranking that decides what to delete

    REM If something is null, then do nothing
    If IsNull(oLCursor) Or IsNull(oRCursor) Or IsNull(oText) Then Exit Sub

    REM Ignore any collapsed ranges
    If oText.compareRegionEnds(oLCursor, oRCursor) <= 0 Then Exit Sub

    REM Default the first and last character to indicate start of new line
    iLastChar = 0
    iThisChar = 0

    REM Start the leftmost cursor moving toward the end of the document
    REM and make certain that no text is currently selected.
    oLCursor.goRight(0, False)

    REM At the end of the document, the cursor can no longer move right
    Do While oLCursor.goRight(1, True)

        REM It is possible that the string is zero length.
        REM This can happen when stepping over certain objects anchored into
        REM the text that contain no text. Extra care must be taken because
        REM this routine can delete these items because the cursor steps over
        REM them but they have no text length. I arbitrarily call this a regular
        REM ASCII character without obtaining the string.
    
```

```

s = oLCursor.getString()
If Len(s) = 0 Then
    oLCursor.goRight(0, False)
    iThisChar = 65
Else
    iThisChar = Asc(oLCursor.getString())
End If

REM If at the last character Then always remove white space
i = oText.compareRegionEnds(oLCursor, oRCursor)
If i = 0 Then
    If IsWhiteSpace(iThisChar) Then oLCursor.setString("")
    Exit Do
End If

REM If went past the end then get out
If i < 0 Then Exit Do

iRank = RankChar(iLastChar, iThisChar)
If iRank = 1 Then
    REM Ready to delete the current character.
    REM The iLastChar is not changed.
    REM Deleting the current character by setting the text to the
    REM empty string causes no text to be selected.
    'Print "Deleting Current with " + iLastChar + " and " + iThisChar
    oLCursor.setString("")
ElseIf iRank = -1 Then
    REM Ready to delete the previous character. One character is already
    REM selected. It was selected by moving right so moving left two
    REM deselects the currently selected character and selects the
    REM character to the left.
    oLCursor.goLeft(2, True)
    'Print "Deleting to the left with " + iLastChar + " and " + iThisChar
    oLCursor.setString("")
    REM Now the cursor is moved over the current character again but
    REM this time it is not selected.
    oLCursor.goRight(1, False)
    REM Set the previous character to the current character.
    iLastChar = iThisChar
Else
    REM Instructed to ignore the current character so deselect any text
    REM and then set the last character to the current character.
    oLCursor.goRight(0, False)
    iLastChar = iThisChar
End If
Loop
End Sub

```

14.6.5. Selected text, closing thoughts

Anyone who has studied algorithms will tell you that a better algorithm is almost always better than a faster computer. An early problem that I solved was counting words in selected text. I created three solutions with varying degrees of success.

My first solution converted the selected text to Basic strings and then manipulated the strings. This solution was fast, counting 8000 words in 2.7 seconds. This solution failed when text strings exceeded 64KB in size, rendering it useless for large documents.

My second solution used a cursor as it walked through the text one character at a time. This solution, although able to handle any length of text, required 47 seconds to count the same 8000 words. In other words, the users found the solution unusably slow.

My final solution used a word cursor, which counted the words in 1.7 seconds. Unfortunately, sometimes the word cursor is sometimes unreliable.

TIP

To count words correctly, visit Andrew Brown's useful macro Web site:
http://www.darwinwars.com/lunatic/bugs/oo_macros.html.

14.7. Search and replace

The search process is directed by a search descriptor, which is able to search only the object that created it. In other words, you cannot use the same search descriptor to search multiple documents. The search descriptor specifies the search text and how the text is searched (see Table 135). The search descriptor is the most complicated component of searching.

Table 135. *Properties of the com.sun.star.util.SearchDescriptor service.*

Property	Description
SearchBackwards	If True, search the document backwards.
SearchCaseSensitive	If True, the case of the letters affects the search.
SearchWords	If True, only complete words are found.
SearchRegularExpression	If True, the search string is treated as a regular expression.
SearchStyles	If True, text is found based on applied style names—not on the text content.
SearchSimilarity	If True, a “similarity search” is performed.
SearchSimilarityRelax	If True, the properties SearchSimilarityRelax, SearchSimilarityRemove, SearchSimilarityAdd, and SearchSimilarityExchange are all used.
SearchSimilarityRemove	Short Integer specifying how many characters may be ignored in a match.
SearchSimilarityAdd	Short Integer specifying how many characters may be added in a match.
SearchSimilarityExchange	Short Integer specifying how many characters may be replaced in a match.

Although not included in Table 135, a search descriptor supports the string property SearchString, which represents the string to search. The XSearchDescriptor interface defines the methods getSearchString() and setSearchString() to get and set the property if you prefer to use a method

rather than directly setting the property. The XSearchable interface defines the methods used for searching and creating the search descriptor (see Table 136).

Table 136. *Methods defined by the com.sun.star.util.XSearchable interface.*

Method	Description
createSearchDescriptor()	Create a new SearchDescriptor.
findAll(XSearchDescriptor)	Return an XindexAccess containing all occurrences.
findFirst(XSearchDescriptor)	Starting from the beginning of the searchable object, return a text range containing the first found text.
findNext(XTextRange, XSearchDescriptor)	Starting from the provided text range, return a text range containing the first found text.

The macro in Listing 364 is very simple; it sets the CharWeight character property of all occurrences of the text “hello” to com.sun.star.awt.FontWeight.BOLD—text ranges support character and paragraph properties.

Listing 364. *Set all occurrences of the word “hello” to bold text.*

```
Sub SetHelloToBold
    Dim oDescriptor 'The search descriptor
    Dim oFound      'The found range

    oDescriptor = ThisComponent.createSearchDescriptor()
    With oDescriptor
        .SearchString = "hello"
        .SearchWords = true           'The attributes default to False
        .SearchCaseSensitive = False 'So setting one to False is redundant
    End With

    ' Find the first one
    oFound = ThisComponent.findFirst(oDescriptor)
    Do While Not IsNull(oFound)
        Print oFound.getString()
        oFound.CharWeight = com.sun.star.awt.FontWeight.BOLD
        oFound = ThisComponent.findNext(oFound.End, oDescriptor)
    Loop
End Sub
```

14.7.1. Searching selected text or a specified range

The trick to searching a specified range of text is to notice that you can use any text range, including a text cursor, in the findNext routine. After each call to findNext(), check the end points of the find to see if the search went too far. You may, therefore, constrain a search to any text range. The primary purpose of the findFirst method is to obtain the initial text range for the findNext routine. You can use the selected text framework very easily to search a range of text.

Listing 365. *Iterate through all occurrences of text between two cursors.*

```
Sub SearchSelectedWorker(oLCursor, oRCursor, oText, oDescriptor)
    If oText.compareRegionEnds(oLCursor, oRCursor) <= 0 Then Exit Sub
```

```

oLCursor.goRight(0, False)
Dim oFound
REM There is no reason to perform a findFirst.
oFound = oDoc.findNext(oLCursor, oDescriptor)
Do While Not IsNull(oFound)
    REM See if we searched past the end
    If -1 = oText.compareRegionEnds(oFound, oRCursor) Then Exit Do
    Print oFound.getString()
    oFound = ThisComponent.findNext(oFound.End, oDescriptor)
Loop
End Sub

```

The text object cannot compare two regions unless they both belong to that text object. Text that resides in a different frame, section, or even a text table, uses a different text object than the main document text object. As an exercise, investigate what happens if the found text is in a different text object than the text object that contains oRCursor in Listing 365. Is the code in Listing 365 robust?

Searching for all occurrences

It is significantly faster to search for all occurrences of the text at one time using the findAll() object method than to repeatedly call findNext(). Care must be used, however, when using all occurrences of the specified text. The macro in Listing 366 is an extreme example of code gone bad on purpose.

Listing 366. Find and replace all occurrences of the word “helloxyzyzy”.

```

Sub SimpleSearchHelloXyzyzy
    Dim oDescriptor 'The search descriptor
    Dim oFound      'The found range
    Dim oFoundAll   'The found range
    Dim n%          'General index variable
    oDescriptor = ThisComponent.createSearchDescriptor()
    oDescriptor.SearchString = "helloxyzyzy"
    oFoundAll = ThisComponent.findAll(oDescriptor)
    For n% = 0 to oFoundAll.getCount()-1
        oFound = oFoundAll.getByIndex(n%)
        'Print oFound.getString()
        oFound.setString("hello" & n%)
    Next
End Sub

```

The macro in Listing 366 obtains a list of text ranges that encapsulate the text “helloxyzyzy”. This text is then replaced with a shorter piece of text. In a perfect world, the occurrences of “helloxyzyzy” would be replaced with “hello0”, “hello1”, “hello2”, ... When each instance is replaced with the shorter text, the total length of the document changes. Remember that the text-range objects were all obtained before the first instance of text is modified. Although the text-range interface is clearly defined, the internal workings are not. I purposely created this example because I expected it to fail, and with no defined behavior my only option was to create a test. Experimentally, I observed that if multiple occurrences of “helloxyzyzy” are contained in the same

word, poor behavior results. I also observed that if all occurrences of “helloxyzy” are contained in separate words, everything works great, and only occurrences of “helloxyzy” are changed—leaving the surrounding text intact. I can but nod my head in approval at the brilliance of the programmers that allow this behavior, while remaining cautiously paranoid, expecting that code relying on this behavior will fail in the future.

Investigating the behavior of the macro in Listing 366 is more than simple academics. I use a similar macro regularly in my own writing. For various reasons, the numbering used during the creation of this book was done manually rather than using the built-in numbering capabilities of OOo—in other words, I should have read Chapter 5 of *Taming OpenOffice.org Writer 1.1* by Jean Hollis Weber. Manually inserting numbering is a problem when a table, figure, or code listing is deleted, inserted, or moved. In other words, if I delete the first table in my document, all of my tables will be numbered incorrectly. I created a macro that verifies that the items are sequentially numbered starting with 1. Consider tables, for example. The table captions use the paragraph style “_table caption.” Tables are numbered using the form “Table 123.” The macro in **Listing 390** verifies that the captions are sequentially numbered, and it renumbers them if required. When the text “Table #” is found in a paragraph using the “_table caption” paragraph style, it is assumed to be identifying a table. The macro in Listing 390 renumbers these based on their occurrence in the document. Each occurrence of “Table #” that is not in the “_table caption” paragraph style is assumed to be a reference to a table.

14.7.2. Searching and replacing

You can perform simple searching and replacing by searching and manually replacing each found occurrence with the replacement text. OOo also defines the XReplaceable interface, which adds the ability to replace all occurrences using one method. You must use an XReplaceDescriptor rather than an XSearchDescriptor, however. Replacing all occurrences of text is very simple (see Listing 367).

TIP	The XReplaceable interface is derived from the XSearchable interface, and the XReplaceDescriptor is derived from the XSearchDescriptor interface.

Listing 367. Replace “hello you” with “hello me”.

```
oDescriptor = oDoc.createReplaceDescriptor()  
With oDescriptor  
    .SearchString = "hello you"  
    .ReplaceString = "hello me"  
End With  
oDoc.ReplaceAll(oDescriptor)
```

14.7.3. Advanced search and replace

While using the OOo GUI for searching and replacing, it is possible to search for and replace attributes as well as text. An inspection of the search-and-replace descriptors reveals the object methods setSearchAttributes() and setReplaceAttributes(). I discovered how to use these object

methods when I found some code written by Alex Savitsky, whom I do not know, and Laurent Godard, whom I do.

The macro in Listing 368 finds all text that is in bold type, converts the text to regular type, and then surrounds the text with two sets of curly brackets. The conversion of attributes to text tags is frequently done while converting formatted text to regular ASCII text with no special formatting capabilities. While reading Listing 368, look for the following interesting techniques:

To search for all text that is bold regardless of the content, you must use a regular expression. In OOo, the period matches any single character and the asterisk means “find zero or more occurrences of the previous character.” Placed together, the regular expression “.*” matches any text. Regular expressions are required to find “any text” that is bold.

While searching regular expressions, the ampersand character is replaced by the found text. In Listing 368, the replacement text “{{ & }}" causes the found text “hello” to become “{{ hello }}”.

Text that is set to bold using an applied style is found only while searching character attributes if SearchStyles is set to True. If the SearchStyles attribute is set to False, only text that has been directly set to bold will be found.

To search for text with specific attributes, create an array of structures of type PropertyValue. There should be one entry in the array for each attribute you want to search. Set the property name to the name of the attribute to search, and the property value to the value for which to search. Although this is complicated to describe using words, it is clearly shown in Listing 368.

You can set attribute values by specifying the replacement attributes in the same way that you set the search attributes.

Listing 368. Replace bold text.

```
Sub ReplaceFormatting
    REM original code : Alex Savitsky
    REM modified by : Laurent Godard
    REM modified by : Andrew Pitonyak
    REM The purpose of this macro is to surround all BOLD elements with {{ }}
    REM and change the Bold attribute to NORMAL by using a regular expression.

    Dim oReplace
    Dim SrchAttributes(0) as new com.sun.star.beans.PropertyValue
    Dim ReplAttributes(0) as new com.sun.star.beans.PropertyValue

    oReplace = ThisComponent.createReplaceDescriptor()

    oReplace.SearchString = ".*"           'Regular expression. Match any text
    oReplace.ReplaceString = "{{ & }}"     'Note the & places the found text back
    oReplace.SearchRegularExpression=True 'Use regular expressions
    oReplace.searchStyles=True             'We want to search styles
    oReplace.searchAll=True                'Do the entire document

    REM This is the attribute to find
    SrchAttributes(0).Name = "CharWeight"
    SrchAttributes(0).Value =com.sun.star.awt.FontWeight.BOLD

    REM This is the attribute to replace it with
```

```

ReplAttributes(0).Name = "CharWeight"
ReplAttributes(0).Value =com.sun.star.awt.FontWeight.NORMAL

REM Set the attributes in the replace descriptor
oReplace.SetSearchAttributes(SrchAttributes())
oReplace.SetReplaceAttributes(ReplAttributes())

REM Now do the work!
ThisComponent.replaceAll(oReplace)
End Sub

```

Table 137 lists the supported regular expression characters.

Table 137. Supported regular expression characters.

Character	Description
.	A period represents any single character. The search term "sh.rt" finds both "shirt" and "short".
*	An asterisk represents any number of characters. The search term "sh*rt" finds "shrt", "shirt", "shiirt", "shioibaldawpclasdfa asdf asdftr" and "short"—to name a few things that it can find.
^	A caret represents the beginning of a paragraph. The search term "^Bob" only finds the word "Bob" if it is at the beginning of a paragraph. The search term "^." finds the first character in a paragraph.
\$	A dollar sign represents the end of a paragraph. The search term "Bob\$" only finds the word "Bob" if it is at the end of a paragraph.
^\$	Search for an empty paragraph. This is listed here only because it is used so frequently.
+	A plus sign indicates that the preceding character must appear at least once. The plus sign also works with the wildcard character ".". For example, "t.+s" finds a section of text that starts with a "t" and ends with an "s".
?	The longest possible text within the paragraph is always found. In other words, multiple words may be found, but the found text will always reside in the same paragraph.
?	A question mark marks the previous character as optional. For example, you could find words that include the characters that come before the character that is in front of the "\". For example, "birds?" finds both "bird" and "birds".
\n	The text "\n" has two uses. When searching, this finds a hard row break inserted with Shift+Enter. In the replace field, this represents a paragraph break. You can, therefore, replace all hard breaks with a paragraph break.
\t	The text "\t" is used to find a tab. In the replace field, this adds a tab.
\>	Using the text ">" indicates that the preceding text must end a word. For example, "book>" finds "checkbook" but not "bookmark".
\<	Using the text "<" indicates that the following text must start a word. For example, "\<book" finds "bookmark" but not "checkbook".
\xXXXX	A backslash followed by a lowercase x followed by a four-digit hexadecimal number (XXXX) finds the character whose Unicode (ASCII) value is the same as the four-digit hexadecimal number.

Character	Description
\	The backslash character followed by anything other than “n”, “t”, “>”, “<”, or “x” is used to specify the character that follows. For example, “\M” finds “M”. The primary purpose is to allow special wild characters to be found. For example, assume that I wanted to find any character preceded by a “+”. Well, the “+” is a special character so I need to precede it with a “\”. Use “\.” to find any character preceding a “+” character.
&	The ampersand is used in the replace text to add the found characters. In Listing 368, the ampersand is used to surround all bold text with “{“ and “}”.
[abc123]	Match any character that is between square brackets. For example, “[ex]+t” finds “text”, “teet”, and “txeeet”; to name a few examples of what it finds.
[a-e]	The minus sign is used to define a range when used inside of square brackets. For example, “[a-e]” matches characters between “a” and “e” and “[a-ex-z]” matches characters between “a” and “e” or “x” and “z”.
[^a-e]	Placing a caret symbol inside square brackets will find anything but the specified characters. For example, “[^a-e]” finds any character that is not between “a” and “e”.
	Placing a vertical bar between two search strings will match what is before and also match what is after. For example, “bob jean” matches the string “bob” and also matches the string “jean”.
{2}	Placing a number between curly brackets finds that many occurrences of the previous character. For example, “me{2}t” matches “meet”, and “[0-9]{3}” matches any three-digit number. Note that “[0-9]{3}” will also find the first three digits of a number with more than three digits, unless “find whole words” is also specified.
{1,2}	Placing two numbers separated by a comma between curly brackets finds the preceding character a variable number of times. For example, “[0-9]{1,4}” finds any number that contains between one and four digits.
()	Text placed within parentheses is treated as a reference. The text “\1” finds the first reference, “\2” finds the second reference, and so on. For example, “([0-9]{3})-[0-9]{2}-\1” finds “123-45-123” but not “123-45-678”. Parentheses can also be used for grouping. For example, “(he she me)\$” finds any paragraph that ends with “he”, “she”, or “me”.
[:digit:]	Finds a single-digit number. For example, “[:digit:]?” finds a single-digit number and “[:digit:]+” finds any number with one or more digits.
[:space:]	Finds any white space, such as spaces and tabs.
[:print:]	Finds any printable characters.
[:cntrl:]	Finds any non-printing characters.
[:alnum:]	Finds any alphanumeric characters (numbers and text characters).
[:alpha:]	Finds any alphabetic characters, both uppercase and lowercase.
[:lower:]	Finds any lowercase characters if “Match case” is selected in the Options area.
[:upper:]	Finds any uppercase characters if “Match case” is selected in the Options area.

14.8. Text content

The primary purpose of a Writer document is to contain simple text. The text is stored and enumerated by paragraphs. When a paragraph enumerates content, each enumerated section uses the same set of properties. In general, text content must be created by the document that will contain the content. After creation, the text content is inserted into the document at a specified location. Paragraphs, however, are not specifically created and then inserted (see Listing 333 near the beginning of this chapter). Paragraph text is inserted as a string and new paragraphs are inserted as control characters (see Table 124). More complicated text content is typically added using the `insertTextContent()` object method (see Listing 334). There are other, less-used methods to insert text content—for example, pasting content from the clipboard (see Listing 381 later in this chapter) and inserting an entire document (see Listing 369).

Listing 369. Insert a document at a text cursor.

```
oCursor.insertDocumentFromURL(sFileURL, Array())
```

Most text content is named and accessible in a similar way (see **Table 138**). The most popular content type in Table 138 is undoubtedly text tables.

Table 138. Content contained in a text document.

Content Type	Mechanism	Access Method
Footnotes	Index Access	<code>getFootnotes()</code>
Endnotes	Index Access	<code>getEndnotes()</code>
Reference marks	Named Access	<code>getReferenceMarks()</code>
Graphic objects	Named Access	<code>getGraphicObjects()</code>
Embedded objects	Named Access	<code>getEmbeddedObjects()</code>
Text tables	Named Access	<code>getTables()</code>
Bookmarks	Named Access	<code>getBookmarks()</code>
Style families	Named Access	<code>getStyleFamilies()</code>
Document indexes	Index Access	<code>getDocumentIndexes()</code>
Text fields	Enumeration Access	<code>getTextFields()</code>
Text field masters	Named Access	<code>getTextFieldMasters()</code>
Text frames	Named Access	<code>getTextFrames()</code>
Text sections	Named Access	<code>getTextSections()</code>

TIP Content accessible using named access also provides indexed access.

14.9. Text tables

Writer documents act as a text-tables supplier, so you can directly retrieve text tables from Writer documents. Although text tables are enumerated as text content along with paragraphs (see

Listing 336), tables are more typically obtained by name or by index (see **Listing 370** and **Figure 93**).

Listing 370. *Demonstrate enumerating text tables.*

```
Sub EnumrateAllTextTables
    Dim oTables 'All of the text tables
    Dim s$      'Work string
    Dim i%      'Index variable

    oTables = ThisComponent.TextTables REM First, access the tables based on index.
    s = "Tables By Index" & CHR$(10)
    For i = 0 To oTables.getCount() - 1
        s = s & "Table " & (i+1) & " = " & oTables(i).Name & CHR$(10)
    Next

    s = s & CHR$(10) & CHR$(10) & "Text Tables By Name" & CHR$(10)
    s = s & Join(oTables.getElementNames(), CHR$(10))
    MsgBox s, 0, "Tables"
End Sub
```

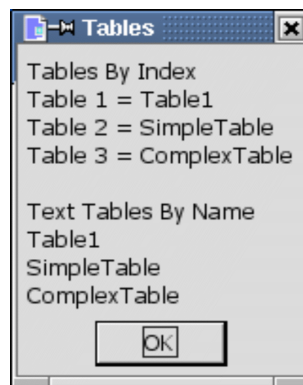


Figure 93. *Tables contained in a document, by index and named access.*

TIP

Most named text content is retrieved, created, inserted, and disposed of in the same way. Learn to do this using tables and you'll be ready to use other named text content—bookmarks, for example.

As with most text content, tables must be created by the document before they are inserted into the document. The macro in Listing 371 inserts a table named “SampleTable” if it does not exist, and removes it if it does.

Listing 371. *Insert then delete a text table.*

```
Sub InsertDeleteTable
    Dim oTable 'Newly created table to insert
    Dim oTables 'All of the text tables
    Dim oInsertPoint 'Where the table will be inserted
    Dim sTableName as String

    sTableName = "SampleTable"
    oTables = ThisComponent.TextTables
```

```

If oTables.hasByName(sTableName) Then
    oTable = oTables.getByName(sTableName)

    REM Although this seems like the correct way to remove text content,
    REM what if the table is not inserted into the primary document's
    REM text object? oTable.dispose() may be safer to use.
    ThisComponent.Text.removeTextContent(oTable)
Else

    REM Let the document create the text table.
    oTable = ThisComponent.CreateInstance( "com.sun.star.text.TextTable" )
    oTable.initialize(2, 3) 'Two rows, three columns

    REM If there is a bookmark named "InsertTableHere", then insert
    REM the table at that point. If this bookmark does not exist,
    REM then simply choose the very end of the document.
    If ThisComponent.getBookmarks().hasByName("InsertTableHere") Then
        oInsertPoint = _
            ThisComponent.getBookmarks().getByName("InsertTableHere").getAnchor()
    Else
        oInsertPoint = ThisComponent.Text.getEnd()
    End If

    REM Now insert the text table at the end of the document.
    REM Note that the text object from the oInsertPoint text
    REM range is used rather than the document text object.
    oInsertPoint.getText().insertTextContent(oInsertPoint , oTable, False)

    REM The setData() object method works ONLY with numerical data.
    REM The setDataArray() object method, however, also allows strings.
    oTable.setDataArray(Array(Array(0, "One", 2), Array(3, "Four", 5)))
    oTable.setName(sTableName)
End If
End Sub

```

TIP

In general, it is better to set table properties before inserting the table into the document. This prevents screen flicker as the object is modified and then redrawn on the screen. Due to a bug in OOo 1.1.0, if Listing 371 is modified to set the data before inserting the table, the table name is modified and it contains an extra garbage character at the end. Be warned, however, that some text content, such as text field masters, cannot change their names after they have been inserted into a document.

The macro in Listing 371 demonstrates many useful techniques:

- A named text content is found and obtained. Notice that finding the table and the bookmark are very similar processes.
- A bookmark is used.
- A text table is created, initialized, and inserted at a location marked by a bookmark.
- Text content is deleted.
- A table is initialized with data.
- The table name is set.

14.9.1. Using the correct text object

It is very important that you use the correct text object. It is possible that the text object in a text section or table cell is not the same as the text object returned by the document. Each text range is associated with a specific text object. Attempting to use object methods from one text object to operate on a text range associated with a different text object causes an error. In Listing 371, a table is removed, with the single line shown in **Listing 372**.

Listing 372. *What if the table is not contained in the document's text object?*

```
ThisComponent.Text.removeTextContent(oTable)
```

The code in Listing 372 assumes that the table is contained in the document's text object. If the table is not contained in the primary document's text object, the code in Listing 372 will fail. Although Listing 372 will rarely fail, it will undoubtedly fail at the worst possible time. The macro works because the sample document was designed so that the table is inserted into the primary document's text object. Either solution shown in Listing 373 may be a better solution for deleting the table.

Listing 373. *Two safe methods to delete the table.*

```
oTable.getAnchor().getText().removeTextContent(oTable)
oTable.dispose()
```

The second time that a text object is used in Listing 371, it is obtained from the anchor returned from a bookmark—this is safe.

Listing 374. *A safe way to get a text object.*

```
oInsertPoint.getText().insertTextContent(oInsertPoint, oTable, False)
```

If the text range `oInsertPoint` is not contained in the document's text object, attempting to insert the table using the document's text object will fail. Only you can decide how careful you need to be when accessing text objects. Consider the selected text framework. What text object is used to create text cursors and to compare text cursors? Can you make the code more robust?

14.9.2. Methods and properties

The methods supported by text tables are very similar to the methods supported by spreadsheets contained in Calc documents (see Chapter 15, “Calc Documents”). Table 139 summarizes the object methods supported by text tables.

Table 139. *Object methods supported by text tables.*

Method	Description
autoFormat(name)	Apply the specified auto-format name to the table.
createCursorByCellName(name)	XTextTableCursor positioned at the specified cell.
createSortDescriptor()	Array of PropertyValue that specify the sort criteria.

Method	Description
dispose()	Destroy a text object, which also removes it from the document.
getAnchor()	Return a text range identifying where the table is anchored. This method allows text content to be added easily before or after a text table.
getCellByName(name)	Return an XCell based on the cell name, such as "B3".
getCellByPosition(col, row)	Numbering starts at zero. This has difficulties with complex tables.
getCellNames()	String array of cell names contained in the table.
getCellRangeByName(name)	XCellRange based on cell name, such as A1:B4. Fails if the name identifies a cell that has been split.
getCellRangeByPosition(left, top, right, bottom)	XCellRange based on numeric range.
getColumnDescriptions()	Array of strings describing the columns. Fails for complex tables.
getColumns()	XTableColumns object enumerates columns by index. Also supports insertByIndex(idx, count) and removeByIndex(idx, count).
getData()	Get numerical data as a nested sequence of values (arrays in an array). Fails for complex tables.
getDataArray()	Same as getData() but may contain String or Double.
getName()	Get the table name as a string.
getRowDescriptions()	Array of strings describing the rows. Fails for complex tables.
getRows()	XTableRows object enumerates rows by index. Also supports insertByIndex(idx, count) and removeByIndex(idx, count).
initialize(rows, cols)	Set the numbers of rows and columns. Must be done before the table is inserted (see Listing 334).
setColumnDescriptions(string())	Set the column descriptions from an array of strings.
setData(Double())	Set numerical data as a nested sequence of values. Fails for complex tables.
setDataArray(array())	Same as setData() but may contain String or Double.
setName(name)	Set the table name.
setRowDescriptions(string())	Set the row descriptions from an array of strings.
sort(array())	Sort the table based on a sort descriptor.

Text table objects also support a variety of properties (see Table 140). Text tables support many of the same properties that are supported by paragraphs (see Table 127).

Table 140. Properties supported by the *com.sun.star.text.TextTable* service.

Property	Description
BreakType	Specify the type of break that is applied at the start of the table (see the BreakType attribute in Table 127).
LeftMargin	Specify the left table margin in 0.01 mm as a Long Integer. Set the HoriOrient property to something other than FULL.
RightMargin	Specify the right table margin in 0.01 mm as a Long Integer. Set the HoriOrient property to something other than FULL.
HoriOrient	Specify the horizontal orientation using the <i>com.sun.star.text.HoriOrientation</i> constants.

Property	Description
	The default value is com.sun.star.text.HoriOrientation.FULL. - NONE = 0 – No alignment is applied. - RIGHT = 1 – The object is aligned at the right side. - CENTER = 2 – The object is aligned at the middle. - LEFT = 3 – The object is aligned at the left side. - INSIDE = 4 – (Not yet supported) - OUTSIDE = 5 – (Not yet supported) - FULL = 6 – The object uses the full space (for text tables only). - LEFT_AND_WIDTH = 7 – The left offset and the width of the object are defined.
KeepTogether	If True, prevents page or column breaks between this table and the following paragraph or text table.
Split	If False, the table will not split across two pages.
PageDescName	If this string is set, a page break occurs before the paragraph, and the new page uses the given page style name (see PageDescName in Table 127).
PageNumberOffset	If a page break occurs, specify a new page number (see PageNumberOffset in Table 127).
RelativeWidth	Specify the width of the table relative to its environment as a Short Integer.
IsWidthRelative	If True, the relative width is valid.
RepeatHeadline	If True, the first row of the table is repeated on every new page.
ShadowFormat	Specify the type, color, and size of the shadow (see ParaShadowFormat in Table 127).
TopMargin	Specify the top table margin in 0.01 mm as a Long Integer.
BottomMargin	Specify the bottom table margin in 0.01 mm as a Long Integer.
BackTransparent	If True, the background color is transparent.
Width	Specify the absolute table width as a Long Integer—this is a read-only property.
ChartRowAsLabel	If True, the first row is treated as axis labels if a chart is created.
ChartColumnAsLabel	If True, the first column is treated as axis labels if a chart is created.
TableBorder	Specify the table borders in a com.sun.star.table.TableBorder structure. The structure contains numerous complicated properties: - The properties TopLine, BottomLine, LeftLine, RightLine, HorizontalLine, and VerticalLine are all structures of type BorderLine as described by the LeftBorder property in Table 127. - The Distance property contains the distance between the lines and other contents. - You can turn each border property on or off by setting one of the following properties to True or False: IsTopLineValid, IsBottomLineValid, IsLeftLineValid, IsRightLineValid, IsHorizontalLineValid, IsVerticalLineValid, and IsDistanceValid.
TableColumnSeparators	Specify the width of each column with an array of table column separators. Each separator is a com.sun.star.text.TableColumnSeparator structure. - Position is a Short Integer that defines the position of a cell separator. - IsVisible determines if the separator is visible. The width of a cell is defined by the position of the separator between adjacent cells. When two cells are merged, the separator is hidden, not removed. The Position values are relative to the text table TableColumnRelativeSum property. This property is valid for a table only if every row has the same structure. If they do not, obtain

Property	Description
	the separators from the individual row objects.
TableColumnRelativeSum	Specify the sum of the column-width values used in TableColumnSeparators as a Short Integer.
BackColor	Specify the paragraph background color as a Long Integer.
BackGraphicURL	Specify the URL of the paragraph background graphic.
BackGraphicFilter	Specify the name of the graphic filter for the paragraph background graphic.
BackGraphicLocation	Specify the position of a background graphic (see ParaBackGraphicLocation in Table 127).

14.9.3. Simple and complex tables

Simply speaking, a text table is a set of rows and columns of text. All of the tables in this book are represented using simple text tables. OOo supports both simple and complex tables. A simple table contains no merged or split cells; the table cells are laid out in a simple grid (see **Table 141**). Each column is labeled alphabetically starting with the letter A, and each row is labeled numerically starting with the number 1. The object method `getCellByName()` uses this name to return the specified cell. A similar object method, `getCellByPosition()`, returns the cell based on the column and row number. The column and row number are zero-based numbers, so requesting (1, 2) returns the cell named “B3”.

Table 141. Simple table with the cell names labeled.

A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3
A4	B4	C4	D4

Use `getRows()` to obtain the rows from the text table. The returned rows object is useful to determine how many rows are present, retrieving, inserting, and deleting rows.

Table 142. Main access methods supported by a table rows object.

Method	Description
<code>getByIndex</code>	Retrieve a specific row.
<code>getCount</code>	Number of rows in the table.
<code>hasElements</code>	Determine if there are any rows in the table.
<code>insertByIndex</code>	Add rows to the table.
<code>removeByIndex</code>	Remove rows from the table.

Although it is easy to enumerate individual rows, a row is not useful for obtaining the cells that it contains. An individual row object is primarily used as follows:

- Set the row height using the Height attribute.
- Set IsAutoHeight to true so that the rows height is automatically adjusted.
- Set IsSplitAllowed to false so that a row can not be split at a page boundary.
- Modify the TableColumnSeparators to change column widths.

Use `getColumns()` to obtain the columns from the text table. The columns object supports the same methods as the rows object. It is not possible, however, to obtain a specific column from a column object; the method `getByIndex()` exists, but it returns null. Although I expected to set a column width by obtaining a specific column and setting the width, you need to modify the table column separators available from the row object.

A complex text table is a text table that is not simple. More accurately, a complex text table contains cells that have been split, or merged. To demonstrate a complex text table, start with *Table 141* and perform the following tasks:

- 1) Right click in cell A2 and choose Cell > Split > Horizontal. Cell A2 becomes two cells. From a cell naming perspective, a new third row is inserted that contains one column. At this point, the API indicates that there are five rows and four columns.

Table 143. Split cell A2 horizontally.

A1	B1	C1	D1
A2	B2	C2	D2
(A3)			
A3=>A4	B3=>B4	C3=>C4	D3=>D4
A4=>A5	B4=>B5	C4=>C5	D4=>D5

- 2) Right click in cell B2 and choose Cell > Split > Vertical. Cell B2 becomes two distinct cells, B2 and C2; a new column appears to have been inserted at cell B2. The API still indicates that there are five rows and four columns.

Table 144. Split cell B2 vertically..

A1	B1	C1	D1
A2	B2	(C2)	C2=>D2
A3			D2=>E2
A4	B4	C4	D4
A5	B5	C5	D5

- 3) Select cells A4 and B4, right click and choose Cell > Merge.

Table 145. Complex table after merging cells in the same row.

A1	B1	C1	D1

A2	B2	C2	D2	E2
A3				
A4, (B4) => A4			C4=>B4	D4=>C4
A5	B5		C5	D5

- 4) Select cells B4 and C5, right click and choose Cell > Merge. Cell C5 just goes away.

Table 146. Split cell B2 vertically..

A1	B1		C1	D1
A2	B2	C2	D2	E2
A3				
A4			B4, (C5)=>B4	C4
A5	B5			D5

TIP Not all object methods work with complex tables. For example, the object methods `getData()` and `setData()` cause an exception for complex tables, which makes sense.

Although the object method `getCellByName()` works as expected for complex tables, `getCellByPosition()` is not able to return all of the cells because it allows only a column and a row number. Use the `getCellNames()` object method to return the names of the cells in a table (see Listing 375); you can then use the cell names to individually obtain each cell in the table.

Listing 375. Join the array of strings to print the cell names in a table.

```
MsgBox Join(oTable.getCellNames(), "|")
```

Place the text cursor in a text table and run the following macro to set the text in each cell to the text name; this modifies the text table and erases all existing text.

Listing 376. Set the text of each cell to contain the cell's name.

```
Sub SetCellNames
    Dim oTable
    Dim sNames
    Dim i As Integer

    oTable = ThisComponent.currentController.ViewCursor.TextTable
    If IsNull(oTable) Then
        Exit Sub
    End If

    sNames = oTable.getCellNames()
    For i = LBound(sNames) To UBound(sNames)
        oTable.getCellByName(sNames(i)).setString(sNames(i))
    Next
End Sub
```

14.9.4. Tables contain cells

The cells in a text table are very versatile objects capable of holding all types of data. The cell objects implement both the XText interface (see Table 124 near the beginning of this chapter) as well as the XCell interface (see Table 147).

TIP Tables are able to create a text table cursor with methods and properties specifically designed to traverse and select cells, and each individual cell is able to produce a text cursor that is local to the cell text object.

Table 147. Methods defined by the `com.sun.star.table.XCell` interface.

Method	Description
<code>getFormula()</code>	The original string typed into the cell, even if it is not a formula.
<code>setFormula(String)</code>	Set the cell's formula. Use <code>setString()</code> from the XText interface to set text.
<code>getValue()</code>	Floating-point (Double) value of the cell.
<code>setValue(Double)</code>	Set the floating-point value of the cell.
<code>getType()</code>	Return a <code>com.sun.star.table.CellContentType</code> enumeration with valid values of EMPTY, VALUE, TEXT, and FORMULA.
<code>getError()</code>	Long Integer error value. If the cell is not a formula, the error value is zero.

Each cell object possesses numerous properties. These properties are generally familiar because they are used in other objects. For example, the `BackColor`, `BackGraphicFilter`, `BackGraphicLocation`, `BackGraphicURL`, `BackTransparent`, `BorderDistance`, `BottomBorder`, `BottomBorderDistance`, `LeftBorder`, `LeftBorderDistance`, `RightBorder`, `RightBorderDistance`, `TopBorder`, and `TopBorderDistance` properties are defined for text tables in Table 140 and/or paragraph properties in Table 127. One of the more useful properties that is available only in the cell object, however, is `CellName`. This is useful to determine the location of the current cursor. The macro in Listing 377 demonstrates a few new manipulations for tables.

The text tables, rows, columns, and cells all support the `BackColor` property. Listing 377 sets the background color of the first row to a light gray, which is commonly used to mark headings.

The `insertByIndex(index, num)` object method is used to insert new rows at the end of a table. Rows can also be inserted into the middle or at the start of a table.

Individual cells are retrieved by name; both numeric values and strings are set. Notice that strings are set using `setString()` rather than `setFormula()`.

TIP Although the Web-based OOo documentation makes no distinction between cells contained in text tables and cells contained in spreadsheets, the two cell types do not support the same property set. For example, the `CellStyle`, `CellBackColor`, and `RotateAngle` properties are not supported in a text document.

Listing 377. Simple text table manipulations.

```
Sub SimpleTableManipulations
    Dim oTable          'Newly created table to insert
    Dim oTables         'All of the text tables
    Dim oInsertPoint    'Where the table will be inserted
    Dim sTableName as String

    sTableName = "SampleTable"
    oTables = ThisComponent.TextTables

    REM Remove the table if it exists!
    If oTables.hasByName(sTableName) Then
        ThisComponent.Text.removeTextContent(oTables.getByName(sTableName))
    End If
    REM Let the document create the text table.
    oTable = ThisComponent.CreateInstance( "com.sun.star.text.TextTable" )
    oTable.initialize(4, 3) 'Two rows, three columns

    REM If there is a bookmark named "InsertTableHere", then insert
    REM the table at that point. If this bookmark does not exist,
    REM then simply choose the very end of the document.
    If ThisComponent.getBookmarks().hasByName("InsertTableHere") Then
        oInsertPoint = _
            ThisComponent.getBookmarks().getByName("InsertTableHere").getAnchor()
    Else
        oInsertPoint = ThisComponent.Text.getEnd()
    End If

    oInsertPoint.getText().insertTextContent(oInsertPoint , oTable, False)

    oTable.setDataArray(Array(Array("Name", "Score", "Test"), _
        Array("Bob", 80, "CCW"), Array("Andy", 80, "CCW"), _
        Array("Jean", 100, "CCI")))
    oTable.setName(sTableName)
    REM Set the first row to have a gray background.
    oTable.getRows().getByIndex(0).BackColor = RGB(235, 235, 235)

    REM removeByIndex uses the same arguments as insertByIndex, namely
    REM the index at which to insert or remove followed by the number
    REM of rows to insert or remove. The following line inserts
    REM one row at index 4.
    oTable.getRows().insertByIndex(4, 1)

    REM Obtain the individual cells and set the values.
    oTable.getCellByName("A5").setString("Whil")
    oTable.getCellByName("B5").setValue(100)
    oTable.getCellByName("C5").setString("Advanced")
End Sub
```

14.9.5. Using a table cursor

Although text table cursors implement methods specific to traversing text tables, they are not significantly different from their text cursor counterparts in general functionality. You can select and manipulate ranges of cells, and set cell properties.

TIP You cannot obtain a text table from the document and then simply insert it again at another location. So, how do you copy a text table? See Listing 381.

Like text cursors, text table cursor movement methods accept a Boolean argument that indicates if the current selection should be expanded (True) or if the cursor should simply be moved (False). The movement methods also return a Boolean value indicating if the movement was successful. Table 148 contains the methods defined by the `XTextTableCursor` interface.

Table 148. *Methods defined by the `com.sun.star.text.XTextTableCursor` interface.*

Method	Description
<code>getRangeName()</code>	Return the cell range selected by this cursor as a string. For example, "B3:D5".
<code>gotoCellByName(String, boolean)</code>	Move the cursor to the cell with the specified name; return Boolean.
<code>goLeft(n, boolean)</code>	Move the cursor left n cells; return Boolean.
<code>goRight(n, boolean)</code>	Move the cursor right n cells; return Boolean.
<code>goUp(n, boolean)</code>	Move the cursor up n cells; return Boolean.
<code>goDown(n, boolean)</code>	Move the cursor down n cells; return Boolean.
<code>gotoStart(boolean)</code>	Move the cursor to the top left cell.
<code>gotoEnd(boolean)</code>	Move the cursor to the bottom right cell.
<code>mergeRange()</code>	Merge the selected range of cells; return True for success.
<code>splitRange(n, boolean)</code>	Create n (an integer) new cells in each cell selected by the cursor. For the Boolean, specify True to split horizontally, False for vertically. Returns True on success.

Text table cursors are used to split and merge table cells. In general, I consider this to be the primary use of a text table cursor. You can use text table cursors to move around the table by using the methods in Table 148. The macro in Listing 378 obtains the table cell names, creates a cell cursor that contains the first table cell, and then moves the cursor to the last cell in the table. A cell range is created based on the range name, and then the entire table is selected by the current controller.

Listing 378. *Select an entire table using a cursor. This may fail for a complex table.*

```
oCellNames = oTable.getCellNames()  
oCursor = oTable.createCursorByCellName(oCellNames(0))  
oCursor.gotoCellByName(oCellNames(UBound(oCellNames())), True)  
oRange = oTable.getCellRangeByName(oCursor.getRangeName()) 'This may fail!  
Thiscomponent.getCurrentController.select(oRange)
```

Listing 378 demonstrates how to select all cells in a table by using a table cell cursor. You can then manipulate the entire table using the cursor. It may fail, however, in selecting the entire table in

the current view. Table cell cursors have no problems with complex tables. The object methods supported by tables, however, do not all support complex tables. A notable example is the object method `getCellRangeByName()`, as used in Listing 378. This is very unfortunate because the view cursor is able to select text based on a cell range, but the table cannot return a cell range that has a split cell as one of the endpoints. For example, the cell range A1.2.1:C4 fails.

There is no easy method to duplicate an entire text table. The clipboard has always been the solution of choice to copy general text (see Listing 381), but, if available, use transferable content instead (see Listing 382). First, use the view cursor or current controller to select the object that you want to copy. Then use a dispatcher to copy the object to the clipboard, move the view cursor where the object should be placed, and then use a dispatcher to paste the object from the clipboard.

As you may have guessed, the difficult part in this process is selecting the table with the view cursor. Although numerous people have tried and failed to solve this problem, a brilliant solution was provided by Paolo Mantovani, a contributor on the OOo mailing lists. Paolo starts by noting that selecting an entire table with the current controller places the view cursor at the start of the first cell (see Listing 379).

Listing 379. *Place the cursor at the start of the first cell in the table.*

```
ThisComponent.CurrentController.select(oTable)
```

Although Listing 379 does not entirely solve the problem, it does provide a good start, because the view cursor is in the table at a known position. Paolo then provides a very succinct method to select the entire table (see Listing 380).

Listing 380. *Select the entire table in the current view.*

```
ThisComponent.CurrentController.select(oTable)
oVCursor.gotoEnd(True) 'Move to the end of the current cell
oVCursor.gotoEnd(True) 'Move to the end of the table
```

TIP

Remember to carefully test all code dealing with tables. A different solution proposed by Paolo—which failed—was to use `goRight()` and then `goDown()` based on the number of rows and columns.

The macro in Listing 381 selects a table by name, copies the table to the clipboard, and then pastes it at the end of the document.

Listing 381. *Copy a text table using the clipboard.*

```
Sub CopyNamedTableToEndWithClipboard(sName As String)
    Dim oTable          'Table to copy
    Dim oText            'Document's text object
    Dim oFrame          'Current frame to use with the dispatcher
    Dim oVCursor        'Current view cursor
    Dim oDispatcher     'Dispatcher for clipboard commands

    oVCursor = ThisComponent.CurrentController.getViewCursor()
    oText = ThisComponent.getText()
    oFrame = ThisComponent.CurrentController.Frame
```

```

oDispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

If NOT ThisComponent.getTextTables().hasByName(sName) Then
    MsgBox "Sorry, the document does not contain table " & sName
    Exit Sub
End If

oTable = ThisComponent.getTextTables().getByName(sName)

REM Place the cursor in the start of the first cell.
REM This is very easy!
ThisComponent.CurrentController.select(oTable)
oVCursor.gotoEnd(True) 'Move to the end of the current cell.
oVCursor.gotoEnd(True) 'Move to the end of the table.

REM Copy the table to the clipboard.
oDispatcher.executeDispatch(oFrame, ".uno:Copy", "", 0, Array())

REM Move the cursor to the end of the document and then paste the table.
oVCursor.gotoRange(oText.getEnd(), False)
oDispatcher.executeDispatch(oFrame, ".uno:Paste", "", 0, Array())
End Sub

```

The clipboard is used by all applications so another application may modify the clipboard while the problem is running. The current controller provides access to the transferable content without using the clipboard.

Listing 382. Copy a text table using transferable content.

```

Sub CopyNamedTableToEndUsingTransferable(sName As String)
    Dim oTable          'Table to copy
    Dim oText            'Document's text object
    Dim oVCursor         'Current view cursor
    Dim o                'Transferable content

    oVCursor = ThisComponent.CurrentController.getViewCursor()
    oText = ThisComponent.getText()

    If NOT ThisComponent.getTextTables().hasByName(sName) Then
        MsgBox "Sorry, the document does not contain table " & sName
        Exit Sub
    End If

    oTable = ThisComponent.getTextTables().getByName(sName)

    REM Place the cursor in the start of the first cell.
    REM This is very easy!
    ThisComponent.CurrentController.select(oTable)
    oVCursor.gotoEnd(True) 'Move to the end of the current cell.
    oVCursor.gotoEnd(True) 'Move to the end of the table.
    o = ThisComponent.CurrentController.getTransferable()

    REM Move the cursor to the end of the document and then paste the table.
    oVCursor.gotoRange(oText.getEnd(), False)

```

```

    ThisComponent.CurrentController.insertTransferable(o)
End Sub

```

14.9.6. Formatting a text table

I format text tables using paragraph styles, alternating the background color for each row, and by changing the cell borders.

Listing 383. Format a text table.

```

Sub FormatTable(Optional oUseTable)
    Dim oTable
    Dim oCell
    Dim nRow As Long
    Dim nCol As Long

    If IsMissing(oUseTable) Then
        oTable = ThisComponent.CurrentController.getViewCursor().TextTable
    Else
        oTable = oUseTable
    End If
    If IsNull(oTable) OR IsEmpty(oTable) Then
        Print "FormatTable: No table specified"
        Exit Sub
    End If

    Dim v
    Dim x
    v = oTable.TableBorder
    x = v.TopLine : x.OuterLineWidth = 2
    v.TopLine = x

    x = v.LeftLine
    x.OuterLineWidth = 2
    v.LeftLine = x

    x = v.RightLine
    x.OuterLineWidth = 2
    v.RightLine = x

    x = v.TopLine
    x.OuterLineWidth = 2
    v.TopLine = x

    x = v.VerticalLine
    x.OuterLineWidth = 2
    v.VerticalLine = x

    x = v.HorizontalLine
    x.OuterLineWidth = 0
    v.HorizontalLine = x

    x = v.BottomLine
    x.OuterLineWidth = 2

```



```

v.BottomLine = x

oTable.TableBorder = v

For nRow = 0 To oTable.getRows().getCount() - 1
    For nCol = 0 To oTable.getColumns().getCount() - 1
        oCell = oTable.getCellByPosition(nCol, nRow)
        If nRow = 0 Then
            oCell.BackColor = 128
            SetParStyle(oCell.getText(), "OOoTableHeader")
        Else
            SetParStyle(oCell.getText(), "OOoTableText")
            If nRow MOD 2 = 1 Then
                oCell.BackColor = -1
            Else
                REM color is (230, 230, 230)
                oCell.BackColor = 15132390
            End If
        End If
    Next
Next
End Sub

```

Each cell has its own text object. The following macro sets every paragraph in a text object to use the same paragraph style.

Listing 384. *Set the paragraph style for every paragraph in a text object.*

```

Sub SetParStyle(oText, sParStyle As String)
    Dim oEnum
    Dim oPar
    oEnum = oText.createEnumeration()
    Do While oEnum.hasMoreElements()
        oPar = oEnum.nextElement()
        If oPar.supportsService("com.sun.star.text.Paragraph") Then
            'oPar.ParaConditionalStyleName = sParStyle
            oPar.ParaStyleName = sParStyle
        End If
    Loop
End Sub

```

14.10. Text fields

A text field is text content that is usually seamlessly inserted into the existing text, but the actual content comes from elsewhere—for example, the total number of pages or a database field. Table 149 lists the standard field types.

Table 149. *Text field services starting with com.sun.star.text.TextField.*

Field Type	Description
Annotation	Inserted note with string properties Author and Content. The property Date, of type com.sun.star.util.Date, contains the date that the note was created.

Field Type	Description
Author	Displays the document's author. The following optional fields may be present: • IsFixed – If False, the author is modified every time the document is saved. • Content – String content of the text field. • AuthorFormat – Constants from the com.sun.star.text.AuthorDisplayFormat constant group; these have the values: FULL (0), LAST_NAME (1), FIRST_NAME (2), or INITIALS (3). • CurrentPresentation – String containing the current text of the field. • FullName – If False, the initials are displayed rather than the full name.
Bibliography	Contains a property named Fields, which is an array of type PropertyValue. This field is dependent on a Bibliography text field master.
Chapter	Chapter information. The Level property is a Byte integer. The ChapterFormat property is a constant group of type com.sun.star.text.ChapterFormat with the following valid values: NAME (0), NUMBER (1), NAME_NUMBER (2), NO_PREFIX_SUFFIX (3), or DIGIT (4).
CharacterCount	Indicates the number of characters in the document. This contains one property, NumberingType, from the constant group com.sun.star.style.NumberingType; valid values are shown in Table 150.
CombinedCharacters	Displays one to six characters and treats them as one character.
ConditionalText	Displays text that changes based on a condition in the text field. • TrueContent – String to use if the condition is True. • FalseContent – String to use if the condition is False. • Condition – String condition to evaluate. • IsConditionTrue – Boolean result of the evaluation (read-only value).
DDE	Displays the result from a DDE connection. Uses a DDE text field master.
Database	Database text field used as a mail-merge field. This field depends on a text field master and contains the following properties: • Content – Merged database content as a String. • CurrentPresentation – Displays content as a String. • DataBaseFormat – If True, the database number display format is used. • NumberFormat – com.sun.star.util.NumberFormatter that formats the field.
DatabaseName	Display the database name when performing database operations (depends on a text field master) with these properties: • DataBaseName – String containing the database name. • DataCommandType – Constant group com.sun.star.sdb.CommandType specifies what DataTableName supports: TABLE (0), QUERY(1), or COMMAND (2). • DataTableName – String containing the table name, query, or statement.
DatabaseNextSet	Increment a selection (depends on a text field master) with these properties: • DataBaseName – String name of the database. • DataCommandType – Constant group com.sun.star.sdb.CommandType specifies what DataTableName supports: TABLE (0), QUERY(1), or COMMAND (2). • DataTableName – String containing the table name, query, or statement. • Condition – String that determines if the selection is advanced to the next position.

Field Type	Description
DatabaseNumberOfSet	Display the current database set number (depends on a text field master) with these properties: • DataBaseName – String name of the database. • DataCommandType – Constant group com.sun.star.sdb.CommandType specifies what DataTableName supports: TABLE (0), QUERY(1), or COMMAND (2). • DataTableName – String containing the table name, query, or statement. • NumberingType – See Table 150 property for valid values. • SetNumber – Long Integer database set.
DatabaseSetNumber	Set the database cursor selection (depends on a text field master) with these properties: • DataBaseName – String name of the database. • DataCommandType – Constant group com.sun.star.sdb.CommandType specifies what DataTableName supports: TABLE (0), QUERY(1), or COMMAND (2). • DataTableName – String containing the table name, query, or statement. • Condition – String condition that determines if the SetNumber is applied. • SetNumber – Long Integer set number to be applied.
DateTime	Display a date or time with the following optional properties: • IsFixed – If False, the current date or time is displayed. • IsDate – If False, this is only a time. If True, this is a date with an optional time. • DateTimeValue – com.sun.star.util.DateTime object with the actual content. • NumberFormat – com.sun.star.util.NumberFormatter that formats the field. • Adjust – Long Integer offset to the date or time in minutes. • IsFixedLanguage – If False, setting the adjacent text language may change the field display.
DropDown	Display a drop-down field with the following properties: • Name – Field name. • Items – Array of strings with the drop-down value. • SelectedItem – The selected item or an empty string if nothing is selected.
EmbeddedObjectCount	Display the number of objects embedded in the document. Contains the NumberingType property; see Table 150 property for valid values.
ExtendedUser	Display information for the user data (under Tools Options OpenOffice.org User Data) such as name, address, or phone number. • Content – String content. • CurrentPresentation – String containing the current text of the field. • IsFixed – If False, the content is updated. • UserDataPart – Specify what to display from the com.sun.star.text.UserDataPart constant group: COMPANY, FIRSTNAME, NAME, SHORTCUT, STREET, COUNTRY, ZIP, CITY, TITLE, POSITION, PHONE_PRIVATE, PHONE_COMPANY, FAX, EMAIL, STATE.
FileName	Display the document file name (URL). Contains the following properties: • CurrentPresentation – String containing the current text of the field. • FileFormat – File name format com.sun.star.text.FilenameDisplayFormat constants with the following values: FULL, PATH, NAME, and NAME_AND_EXT. • IsFixed – If False, the content is updated.

Field Type	Description
GetExpression	Display the result from a “SetExpression” text field. (use Insert > Fields > Other, to open the Fields dialog, then navigate to the Cross-references tab). - Content – String content. - CurrentPresentation – String containing the current text of the field. - NumberFormat – com.sun.star.util.NumberFormatter that formats the field. - NumberingType – See Table 150 property for valid values. - IsShowFormula – If True, the formula is displayed rather than the content. - SubType – Variable type from the com.sun.star.text.SetVariableType constants with the following values: VAR, SEQUENCE, FORMULA, and STRING. - Value – Numerical (Double) value of the field. - IsFixedLanguage – If False, setting the adjacent text language may change the field display. <i>A GetExpression field references a SetExpression field. If you delete the corresponding SetExpression field, the CurrentPresentation says something like “Invalid reference”, but I know of no other way to determine that the reference is no longer valid.</i>
GetReference	Reference field with these properties: - CurrentPresentation – String containing the current text of the field. - ReferenceFieldSource – com.sun.star.text.ReferenceFieldSource constant with the following values: REFERENCE_MARK, SEQUENCE_FIELD, BOOKMARK, FOOTNOTE, or ENDNOTE. - SourceName – String reference name such as a bookmark name. - ReferenceFieldPart – com.sun.star.text.ReferenceFieldPart constant with the following values: PAGE, CHAPTER, TEXT, UP_DOWN, PAGE_DESC, CATEGORY_AND_NUMBER, ONLY_CAPTION, and ONLY_SEQUENCE_NUMBER. - SequenceNumber – Short integer sequence number used as sequence field or ReferenceId property of a footnote or endnote.
GraphicObjectCount	Display the number of graphic objects embedded in the document. Contains the NumberingType property; see Table 150 property for valid values.
HiddenParagraph	Allow a paragraph to be hidden. Used, for example, to create a test with the questions and answers all in one document. Setting the answers to hidden allows the test questions to be printed for the students. - Condition – String condition to evaluate. - IsHidden – Boolean result of the last evaluation of the condition.
HiddenText	A field with hidden text. Differs from a hidden paragraph in that only the text in the field is hidden, rather than the entire containing paragraph. - Content – String text content of the hidden text field. - Condition – String condition. - IsHidden – Boolean result of the last evaluation of the condition.
Input	Text input field. - Content – String text content of the field. - Hint – String hint text.
InputUser	User-defined text field that depends on a field master. - Content – String text content of the field. - Hint – String hint text.

Field Type	Description
JumpEdit	Placeholder text field. - Hint – String hint text. - Placeholder – String text of the placeholder. - PlaceholderType – com.sun.star.text.PlaceholderType constant with the following valid values: TEXT, TABLE, TEXTFRAME, GRAPHIC, or OBJECT.
Macro	Macro text field. - Hint – String hint text. - MacroName – String macro name to run. - MacroLibrary – String library name that contains the macro.
PageCount	Display the number of pages in the document. Contains the NumberingType property; see Table 150 property for valid values.
PageNumber	Display a page number. - Offset – Short Integer offset to show a different page number. - SubType – Which page is displayed from the com.sun.star.text.PageNumberType enumeration. Valid values: PREV, CURRENT, or NEXT. - UserText – String that is displayed when the NumberingType is CHAR_SPECIAL. - NumberingType – See Table 150 property for valid values.
ParagraphCount	Display the number of paragraphs in the document. Contains the NumberingType property; see Table 150 property for valid values.
ReferencePageGet	Display the page number of a reference point. Contains the NumberingType property; see Table 150 property for valid values.
ReferencePageSet	Insert additional page numbers. Contains these properties: - Offset – Short Integer that changes the displayed value of a ReferencePageGet field. - NameOn – If True, the ReferencePageGet text fields are displayed.
Script	Display text obtained by running a script. Contains these properties: - Content – Script text or URL of the script as a string. - ScriptType – String script type, such as JavaScript. - URLContent – If True, Content is a URL. If False, Content is the script text.

Field Type	Description
SetExpression	An expression text field (use Insert > Fields > Other, to open the Fields dialog, then navigate to the Variables tab). Contains these properties: • Content – String content. • CurrentPresentation – String containing the current text of the field. • NumberFormat – com.sun.star.util.NumberFormatter that formats the field. • NumberingType – See Table 150 property for valid values. • IsShowFormula – If True, the formula is displayed rather than the content. • Hint – String hint used if this is an input field. • IsInput – If True, the field is an input field. • IsVisible – If True, the field is visible. • SequenceValue – Sequence value when this field is used as sequence field. • SubType – Variable type from the com.sun.star.text.SetVariableType constants with the following values: VAR, SEQUENCE, FORMULA, and STRING. • Value – Numerical (Double) value of the field. • VariableName – Name of the associated set expression field master. • IsFixedLanguage – If False, setting the adjacent text language may change the field display.
TableCount	Display the number of tables in the document. Contains the NumberingType property; see Table 150 property for valid values.
TemplateName	Display the name of the template used to create the document. Supports the FileFormat property as supported by the Filename property.
URL	Display a URL. Contains these properties: • Format – Short Integer specifying the URL output format. • URL – String containing the unparsed original URL. • Representation – Display string shown to the user. • TargetFrame – String frame name where the URL will be opened.
User	Display a user-defined field with a field master. Contains these properties: • IsShowFormula – If True, the formula is displayed rather than the content. • IsVisible – If True, the field is visible. • NumberFormat – com.sun.star.util.NumberFormatter that formats the field. • IsFixedLanguage – If False, setting the adjacent text language may change the field display.
WordCount	Display the number of words in the document. Contains the NumberingType property; see Table 150 property for valid values.
docinfo.ChangeAuthor	Display the name of the last author to modify the document. • Author – String containing the name of the author. • CurrentPresentation – Current content of the text field as a String. • IsFixed – If False, the content is updated when the document is saved.

Field Type	Description
docinfo.ChangeDateTime	Display the date and time the document was last changed. Contains these properties: • CurrentPresentation – Current content of the text field as a String. • IsFixed – If False, the current date or time is displayed. • IsDate – If False, this is only a time. If True, this is a date with an optional time. • DateTimeValue – com.sun.star.util.DateTime object with the actual content. • NumberFormat – com.sun.star.util.NumberFormatter that formats the field. • IsFixedLanguage – If False, setting the adjacent text language may change the field display.
docinfo.CreateAuthor	Display the name of the author who created the document (see docinfo.ChangeAuthor for supported properties).
docinfo.CreateDateTime	Display the date and time the document was created (see docinfo.ChangeDateTime for supported properties).
docinfo.Custom	Display the user-defined field in the document information; used to be docinfo.Info0.
docinfo.Description	Display the document description as contained in the document properties (File Properties). • Content – String content. • CurrentPresentation – String containing the current text of the field. • IsFixed – If False, the content is updated when the document information is changed.
docinfo.EditTime	Display the duration the document has been edited. In other words, how long did it take to write? • CurrentPresentation – String containing the current text of the field. • IsFixed – If False, the date or time is always displayed as the current date or time. • DateTimeValue – Date and time as a Double. • NumberFormat – com.sun.star.util.NumberFormatter that formats the field. • IsFixedLanguage – If False, setting the adjacent text language may change the field display.
docinfo.Info0	Deprecated, use docinfo.Custom instead.
docinfo.Info1	Deprecated, use docinfo.Custom instead.
docinfo.Info2	Deprecated, use docinfo.Custom instead.
docinfo.Info3	Deprecated, use docinfo.Custom instead.
docinfo.Keywords	Display the document info keywords (see docinfo.Description).
docinfo.PrintAuthor	Display the name of the author who printed the document (see docinfo.ChangeAuthor for supported properties).
docinfo.PrintDateTime	Display the time the document was last printed (see docinfo.ChangeDateTime for supported properties).
docinfo.Revision	Display the current document revision (see docinfo.Description).
docinfo.Subject	Display the document subject specified in the document information (see docinfo.Description).
docinfo.Title	Display the document title specified in the document information (see docinfo.Description).

TIP

The Annotation field is a service of type `com.sun.star.text.TextField.Annotation`. Some sources of documentation show the text “textfield” in all lowercase letters; this is incorrect. The code in Listing 385 shows this correctly.

Table 150 contains the valid values for the `CharacterCount` property from Table 149. Most API listings stop at `CIRCLE_NUMBER=14`, I do not know why.

Table 150. Constants defined by `com.sun.star.style.NumberingType`.

Constant	Description
<code>CHARS_UPPER_LETTER</code>	Numbering is in uppercase letters as “A, B, C, D, ...”.
<code>CHARS_LOWER_LETTER</code>	Numbering is in lowercase letters as “a, b, c, d,...”.
<code>ROMAN_UPPER</code>	Numbering is in Roman numbers with uppercase letters as “I, II, III, IV, ...”.
<code>ROMAN_LOWER</code>	Numbering is in Roman numbers with lowercase letters as “i, ii, iii, iv, ...”.
<code>ARABIC</code>	Numbering is in Arabic numbers as “1, 2, 3, 4, ...”.
<code>NUMBER_NONE</code>	Numbering is invisible.
<code>CHAR_SPECIAL</code>	Use a character from a specified font.
<code>PAGE_DESCRIPTOR</code>	Numbering is specified in the page style.
<code>BITMAP</code>	Numbering is displayed as a bitmap graphic.
<code>CHARS_UPPER_LETTER_N</code>	Numbering is in uppercase letters as “A, B, ..., Y, Z, AA, BB, CC, ... AAA, ...”.
<code>CHARS_LOWER_LETTER_N</code>	Numbering is in lowercase letters as “a, b, ..., y, z, aa, bb, cc, ... aaa, ...”.
<code>TRANSLITERATION</code>	A transliteration module is used to produce numbers in Chinese, Japanese, etc.
<code>NATIVE_NUMBERING</code>	The native-number-supplier service is called to produce numbers in native languages.
<code>FULLWIDTH_ARABIC</code>	Numbering for full-width Arabic number.
<code>CIRCLE_NUMBER</code>	Bullet for Circle Number.
<code>NUMBER_LOWER_ZH</code>	Numbering for Chinese lowercase numbers.
<code>NUMBER_UPPER_ZH</code>	Numbering for Chinese uppercase numbers.
<code>NUMBER_UPPER_ZH_TW</code>	Numbering for Traditional Chinese uppercase numbers.
<code>TIAN_GAN_ZH</code>	Bullet for Chinese Tian Gan.
<code>DI_ZI_ZH</code>	Bullet for Chinese Di Zi.
<code>NUMBER_TRADITIONAL_JA</code>	Numbering for Japanese traditional numbers.
<code>AIU_FULLWIDTH_JA</code>	Bullet for Japanese AIU full width.
<code>AIU_HALFWIDTH_JA</code>	Bullet for Japanese AIU half width.
<code>IROHA_FULLWIDTH_JA</code>	Bullet for Japanese IROHA full width.
<code>IROHA_HALFWIDTH_JA</code>	Bullet for Japanese IROHA half width.
<code>NUMBER_UPPER_KO</code>	Numbering for Korean uppercase numbers.
<code>NUMBER_HANGUL_KO</code>	Numbering for Korean hangul numbers.

Constant	Description
HANGUL_JAMO_KO	Bullet for Korean Hangul Jamo.
HANGUL_SYLLABLE_KO	Bullet for Korean Hangul Syllable.
HANGUL_CIRCLED_JAMO_KO	Bullet for Korean Hangul Circled Jamo.
HANGUL_CIRCLED_SYLLABLE_KO	Bullet for Korean Hangul Circled Syllable.
CHARS_ARABIC	Numbering in Arabic alphabet letters.
CHARS_THAI	Numbering in Thai alphabet letters.

The text fields contained in the document are available using the `getTextFields()` object method (see Table 138). Every text field object supports the object method `getPresentation(boolean)`, which returns a string representing either the field type (True) or the displayed text (False). See **Listing 385** and **Figure 94**.

Listing 385. Display Fields.

```
Sub DisplayFields
    oEnum = ThisComponent.getTextFields().createEnumeration()
    Do While oEnum.hasMoreElements()
        oField = oEnum.nextElement()
        s = s & oField.getPresentation(True) & " = " & 'Field type
        If oField.supportsService("com.sun.star.text.TextField.Annotation") Then
            REM More cryptic, I could use If oField.getPresentation(True) = "Note"...
            REM A "Note" has no displayed content so calling getPresentation(False)
            REM returns an empty string. Instead, obtain the author and the content.
            s = s & oField.Author & " says " & oField.Content
        Else
            s = s & oField.getPresentation(False) & 'String content
        End If
        s = s & CHR$(13)
    Loop
    MsgBox s, 0, "Text Fields"
End Sub
```

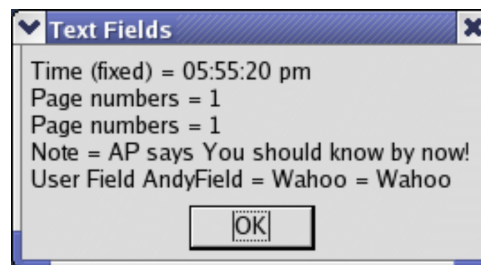


Figure 94. Text fields in the current document.

The source document used to create Figure 94 contains a `DateTime` field, `PageNumber` fields, `Annotation` field, and a `User` field. The code in Listing 385 provides special treatment for note fields to display both the author and the note. The field is checked to see if it supports the `Annotation` service by using the `supportsServiceObject()` method.

The primary method of finding a specific text field is by enumerating the text fields as shown in Listing 385. If the document is large and contains many text fields, it may take a long time to find a specific text field. If you know where the text field is located in the document, Listing 354 demonstrates how to find a text field by enumerating text content in a paragraph.

TIP Text fields implement the object method `update()`. The `update()` method causes a text field to update itself with the most current information if it is applicable. For example, date/time, file name, and the document information text fields all update to the most current information.

14.10.1. Text master fields

Some text fields contain their own content, and others rely on an external source to obtain the displayed information. The external source is called a master field. Table 151 lists the text field types that require a master field. Besides the properties shown in Table 151, every text field master also supports the properties shown in Table 152.

TIP The object method `getTextFieldMaster()` returns a text field’s master field. Unfortunately, every field, even fields that do not have master fields, implements this method and returns a non-null master field. OOo may crash if you obtain and manipulate a master field from a field that does not contain one.

Although text fields are accessible only by enumeration, master fields are accessible by name and enumeration (see Table 138). The name used for a master field is obtained by appending the field name to the field master type. For example, the User field “AndyField”, as shown in Figure 94, has a master field named `com.sun.star.text.FieldMaster.User.AndyField`. Database master fields are named differently than all of the other master fields; they append the `DatabaseName`, `DatatableName`, and `DataColumnName` to the service name. Listing 386 demonstrates how to obtain the text field masters in a document. Figure 95 shows the results.

Table 151. Text field services starting with `com.sun.star.text.FieldMaster`.

Field Type	Description
Bibliography	Field master to a Bibliography text field. Contains these properties: • <code>IsNumberEntries</code> – If True, the fields are numbered; otherwise, the short entry name is used. • <code>IsSortByPosition</code> – If True, the bibliography index is sorted by the document position (see <code>SortKeys</code>). • <code>BracketBefore</code> – The opening bracket displayed in the Bibliography text field. • <code>BracketAfter</code> – The closing bracket displayed in the Bibliography text field. • <code>SortKeys</code> – This array of <code>PropertyValues</code> is used if <code>IsSortByPosition</code> is False. The properties are a sequence of the property <code>SortKey</code> (<code>com.sun.star.text.BibliographyDataField</code> constant identifying the field to sort) and <code>IsSortAscending</code> (Boolean). • <code>Locale</code> – <code>com.sun.star.lang.Locale</code> of the field master. • <code>SortAlgorithm</code> – String containing the name of the sort algorithm used to sort the text fields.

Field Type	Description
DDE	Field master to a DDE text field. Contains these properties: • DDECommandElement – DDE command as a string. • DDECommandFile – File string of the DDE command. • DDECommandType – DDE command type as a string. • IsAutomaticUpdate – If True, the DDE link is automatically updated.
Database	Field master to a Database text field. Contains these properties: • DataBaseName – String name of the data source. • CommandType – Long Integer CommandType (0 = table, 1 = query, 2 = statement). • DataTableName – Command string type is determined by the CommandType property. • DataColumnName – Database column name as a String.
SetExpression	Field master to a “set expression” text field. Contains these properties: • ChapterNumberingLevel – Chapter number as a byte, if this is a number sequence. • NumberingSeparator – Numbering separator string, used if this is a number sequence. • SubType – Variable type from the com.sun.star.text.SetVariableType constants with the following values: VAR, SEQUENCE, FORMULA, and STRING.
User	Field master to a user text field. Contains the properties: • IsExpression – If True, the field contains an expression. • Value – Double value. • Content – Field content as a string.

Table 152. Properties defined by the service *com.sun.star.text.FieldMaster*.

Property	Description
Name	Optional string with the field name; this must be set before the field is added to the document.
DependentTextFields	Array of text fields that use this master field.
InstanceName	String instance name as it is used in the XTextFieldsSupplier.

Listing 386. Show text field masters.

```

Sub ShowFieldMasters
    Dim oMasters          'All of the text field masters
    Dim oMasterNames      'Array of the text field master names
    Dim i%, j%            'Index variables
    Dim sMasterName$      'Full name of the master field
    Dim s$                'Utility string
    Dim oMaster           'Master field

    REM Obtain the text field masters object.
    oMasters = ThisComponent.getTextFieldMasters()

    REM Obtain ALL of the text master field names.
    REM This is an array of strings.
    oMasterNames = oMasters.getElementNames()
    For i = LBound(oMasterNames) to UBound(oMasterNames)

```

```

REM For a given name, obtain the master field,
REM then look at the DependentTextFields property,
REM which is an array of text fields that depend
REM on this master field.
sMasterName = oMasterNames(i)
oMaster = oMasters.GetByName(sMasterName)
s = s & "****" & sMasterName & "****" & CHR$(10)
s = s & oMaster.Name & " Contains " & _
    CStr(UBound(oMaster.DependentTextFields) + 1) & _
    " dependent fields" & CHR$(10)
s = s & CHR$(13)
Next I
REM Directly obtain a master field based on the name.
REM This is a user field that I added to the source code file.
If oMasters.HasByName("com.sun.star.text.FieldMaster.User.AndyField") Then
    oMaster=oMasters.GetByName("com.sun.star.text.FieldMaster.User.AndyField")
    s = s & "Directly obtained the field master " & oMaster.Name & CHR$(10) & _
        "The field contains the text " & oMaster.Content
End If
MsgBox s, 0, "Text Field Masters"
End Sub

```

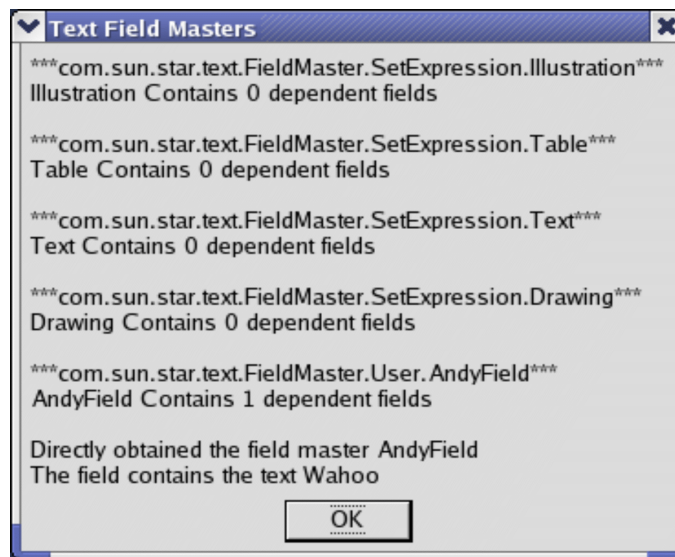


Figure 95. Text master fields in the current document.

14.10.2. Creating and adding text fields

Text fields must be created by the document that will contain them. It is also the document that will destroy them if you choose to remove them from the document. Listing 387 demonstrates the creation, configuration, and insertion of both a DateTime text field and an Annotation text field. The text fields are appended to the end of the document. The DateTime text field is formatted in an unusual manner, so a new number format style is created if it does not yet exist.

TIP

The very popular routine FindCreateNumberFormatStyle (see Listing 391) is used by Listing 387.

What is required to properly insert an annotation field has changed over time and may differ between AOO and LO. When OOo, inserting an annotation set the insertion date to the current time. My latest testing with LO 4.0.1.2 does not set a date. In LO, there is a Date property, which seems to be ignored. To set the date, ou must set the DateTimeValue property instead. The insertion date/time must be set before the text field is inserted, the date/time cannot be changed after. I have not tested this particular macro with AOO.

Listing 387. Insert text fields.

```
Sub InsertFields
    Dim oText    'Text object for the current document
    Dim oField   'Field to insert
    Dim oDoc     'oDoc is fewer characters than ThisComponent

    oDoc = ThisComponent
    oText = oDoc.Text

    REM Start by inserting a date time text field at the end of the
    REM document. Format the date time as "DD. MMM YYYY"
    REM Insert some explanatory text before the newly inserted field.
    oText.insertString(oText.getEnd(), "Today is ", FALSE)

    REM Create a date time field.
    oField = oDoc.createInstance("com.sun.star.text.TextField.DateTime")
    oField.IsFixed = TRUE
    oField.NumberFormat = FindCreateNumberFormatStyle("DD. MMMM YYYY", oDoc)
    oText.insertTextContent(oText.getEnd(), oField, False)

    REM Now, insert an Annotation after the inserted text field.
    REM Lie about the date and say that I did it a little while ago!
    Dim oDate As New com.sun.star.util.Date
    Dim oRightNow
    oRightNow = Now
    With oDate
        .Day    = Day(oRightNow - 10)
        .Month  = Month(oRightNow - 10)
        .Year   = Year(oRightNow - 10)
    End With

    Dim oDT As New com.sun.star.util.DateTime
    With oDT
        .Day    = Day(oRightNow - 10)
        .Month  = Month(oRightNow - 10)
        .Year   = Year(oRightNow - 10)
        .Hours  = 4
        .Minutes = 5
        .Seconds = 6
        .HundredthSeconds = 0
    End With

    REM Like most text content, the field must be created by the document
    REM that will contain it.
```

```

REM In LO, setting the Date field does nothing.
oField = oDoc.CreateInstance("com.sun.star.text.TextField.Annotation")
With oField
    .Author = "AP"
    .Content = "This note is next to a date field that I just added"
    .Date = oDate
    .DateTimeValue = oDT
End With
oText.InsertTextContent(oText.getEnd(), oField, False)
MsgBox "Two fields inserted at the end of the document"
End Sub

```

Inserting a field that requires a text field master is slightly more difficult than inserting a regular text field. Both the master field and the dependent text field must be created by the document using the object method `CreateInstance()`. The master field must be named before it is used; after inserting a field into a document, you cannot change the name. The dependent field is attached to the master field, which provides the content to the dependent field. The dependent field, not the master field, is inserted as text content into the document. The dependent field can be removed from the document by using the `removeTextContent()` object method. To remove the master field, use the `dispose()` method of the master field.

Listing 388 demonstrates the use of master fields by performing various operations on a master field named “TestField”. Three specific states are checked and appropriate behavior is taken as follows:

If the master field does not exist, the master field and a dependent field are created and inserted into the document. The field is now visible by opening the Field dialog using `Insert | Fields | Other` and choosing the `Variables` tab.

If the master field exists with a corresponding dependent field, the dependent field is removed from the document. The master field still exists, but no dependent field is inserted in the document. You can view the master field by using the Field dialog.

If the master field exists and has no corresponding dependent field, the master field is removed by using the `dispose()` object method. The Field dialog no longer shows the master field.

Listing 388. Insert field master.

```

Sub InsertFieldMaster
    Dim oMasters 'All of the text field masters
    Dim oText    'Text object for the current document
    Dim oUField  'User field to insert
    Dim oMField  'The master field for the user field
    Dim oDoc     'oDoc is fewer characters than ThisComponent
    Dim sLead$   'Leading field name
    Dim sName$   'Name of the field to remove or insert
    Dim sTotName$ 'The entire name

    REM Set the names.
    sName = "TestField"
    sLead = "com.sun.star.text.FieldMaster.User"
    sTotName = sLead & "." & sName

```

```

REM Initialize some values.
oDoc = ThisComponent
oText = oDoc.Text
oMasters = ThisComponent.getTextFieldMasters()

REM If the master field already exists, then perform special handling.
REM Special handling is for illustrative purposes only, not that it
REM solves any particularly fun and exciting problem.
If oMasters.hasByName(sTotName) Then
    REM Obtain the master field and the fields dependent on this field.
    oMField = oMasters.getByName(sTotName)

    REM If there are fields dependent on this field then
    REM the array of dependent fields has values!
    If UBound(oMField.DependentTextFields) >= 0 Then
        REM Remove the text content and it disappears from the
        REM document. The master field still exists, however!
        oUField = oMField.DependentTextFields(0)
        oText.removeTextContent(oUField)
        MsgBox "Removed one instance from the document"
    Else
        REM I arbitrarily decided that I would destroy the master field
        REM but I could just as easily create a new user field and
        REM attach it to the existing master field and then insert
        REM the new field into the document.
        MsgBox "No instances in the document, disposing master field"
        oMField.content=""
        oMField.dispose()
    End If
Else
    REM Create a User text field that requires a master field.
    oUField = oDoc.createInstance("com.sun.star.text.TextField.User")

    REM Now create the master field.
    Dim oMasterField
    oMasterField = oDoc.createInstance(sLead)

    REM You CANNOT change the name of a master field AFTER it is inserted
    REM into a document so you must set it now.
    oMasterField.Name = sName

    REM This is the data that will be displayed. Remember that the
    REM user field displays what the master tells it to display.
    oMasterField.Content = "Hello"

    REM A user field must be attached to a master field.
    REM The user field is now a "DependentTextField".
    oUField.attachTextFieldMaster(oMasterField)

    REM Insert the user field into the document.
    oText.insertTextContent(oText.getEnd(), oUField, False)
    MsgBox "One field inserted at the end of the document"

```

```
End If
End Sub
```

14.11. Bookmarks

A bookmark is text content that is accessible based on its name. A bookmark may encompass a text range or a single point. Listing 371 inserts text content at the point where a bookmark is anchored. Use the `getString()` object method to obtain the string contained in the bookmark. Use `setString()` to set the string contained in the bookmark. If the bookmark is merely a point, the text is merely inserted before the bookmark. When a created bookmark is inserted into the text, the insertion point determines the bookmark's anchor position.

Listing 389. *Demonstrate how to add a bookmark.*

```
Sub AddBookmark
    Dim oBookmark 'Created bookmark to add
    Dim oCurs      'Text cursor

    REM Create a text cursor that contains the last four characters
    REM in the document.
    oCurs = ThisComponent.Text.createTextCursor()
    oCurs.gotoEnd(False)
    oCurs.goLeft(4, True)

    REM Create the bookmark.
    oBookmark = ThisComponent.CreateInstance("com.sun.star.text.Bookmark")

    REM If the bookmark is not given a name, OOo will create a name.
    REM If the name already exists, a number is appended to the name.
    oBookmark.setName("Robert")

    REM Because True is used, the bookmark contains the last four characters
    REM in the document. If False is used instead, then the bookmark
    REM contains no characters and it is positioned before the fourth character
    REM from the end of the document.
    ThisComponent.Text.insertTextContent(oCurs, oBookmark, False)
End Sub
```

14.12. Sequence fields, references, and formatting

Consider the caption for Listing 389. Ignoring formatting, enter the caption as follows:

1. Enter the text “Listing”.
2. Use **Insert > Fields > Other** to open the Fields dialog.
3. Select the Variables tab.
4. Select the Number Range type .
5. Enter *Listing* for name and Listing + 1 for the value.
6. Click Insert.

7. Enter the rest of the caption.

Sequence fields (number ranges) work well because they automatically renumber as captions are added and removed. A reference to a sequence field renumbers if the field itself changes value because a field is added or removed.

I had a collection of documents containing hundreds of captions and references. Unfortunately, all captions and references were text as opposed to fields. I needed a macro to convert the text captions and references to use sequence fields and linked cross references.

14.12.1. Formatting numbers and dates

Sequence fields can be formatted in many ways. Ultimately, the formatting is associated to a defined format string. OOo is configured to know many common number and date formats, and you can create new formats as you desire. Each number format is assigned a numeric ID. Things that format numbers and dates – such as fields and table cells – contain the numeric key for the formatting string used to display the contained value.

List formats known to the current document

When you create a new number format, the format is stored in the current document. The supported numeric formats and the ID for a specific format will differ from document to document.

Listing 390. List the formats in the current document.

```
Sub ListFormatsInCurrentDocument()  
    Dim oDoc                ' Document created to hold the format strings.  
    Dim oFormats             ' Formats in the current document.  
    Dim oFormat              ' Current format object.  
    Dim oData                ' Keys queried from the formats.  
    Dim i%                   ' General index variable.  
    Dim sFormat$             ' Current format string.  
    Dim sPrevFormat$         ' Previous format string.  
    Dim aLocale as new com.sun.star.lang.Locale  
  
    oFormats = ThisComponent.getNumberFormats()  
  
    ' Create an output document.  
    oDoc = StarDesktop.loadComponentFromURL( "private:factory/swriter", "_blank", 0, Array() )  
    oData = oFormats.queryKeys(com.sun.star.util.NumberFormat.ALL, aLocale, False)  
    For i = LBound(oData) To UBound(oData)  
        oFormat=oFormats.getbykey(oData(i))  
        sFormat=oFormat.FormatString  
        If sFormat<>sPrevFormat Then  
            sPrevFormat=sFormat  
            oDoc.getText().insertString(oDoc.getText().End, _  
                CStr(oData(i)) & CHR$(9) & sFormat & CHR$(10), False)  
        End If  
    Next  
End Sub
```

Find or create a numeric format.

When a number format is added, it may be modified and stored in a slightly different format. QueryKey searches the internal modified query strings without first modifying the input string. The end result is that after storing a key queryKey will claim that the format string is not present. If you then dutifully, add the format string, an exception will be thrown because it already exists.

1. Adding "#,##0.00_);[Red](#,##0.000)" stores "#,##0.00_);[RED](#,##0.000)"; red is converted to upper case (see https://issues.apache.org/ooo/show_bug.cgi?id=72380).
2. While storing, "##0,0#\h" is changed to "##,00#\h".

The usual advice is to use a number format that does not change.

Listing 391 returns the ID that identifies the specified format. The returned ID can be used to format a value. If you add a format and then the format is not available, make note of the format ID and use Listing 390 to display the formats in the modified form.

Listing 391. Find or create a numeric format.

```
'sFormat - Format to find / create.
'oDoc     - Document to use. If omitted, the current document is used.
'locale   - Locale to use. If omitted, the default locale is used.
Function FindCreateNumberFormatStyle (sFormat$, Optional oDoc, Optional locale)
    Dim oDocument As Object
    Dim aLocale as new com.sun.star.lang.Locale
    Dim oFormats As Object
    Dim formatNum As Long
    oDocument = IIf(IsMissing(oDoc), ThisComponent, oDoc)
    oFormats = oDocument.getNumberFormats()
    'I could set the locale from values stored at
    'http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt
    'http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html
    'I use a NULL locale and let it use what ever it likes.
    'First, see if the number format exists
    If ( Not IsMissing(locale)) Then
        aLocale = locale
    End If
    formatNum = oFormats.queryKey (sFormat, aLocale, TRUE)
    'MsgBox "Current Format number is" & formatNum
    'If the number format does not exist then add it
    If (formatNum = -1) Then
        formatNum = oFormats.addNew(sFormat, aLocale)
        If (formatNum = -1) Then formatNum = 0
        'MsgBox "new Format number is " & formatNum
    End If
    FindCreateNumberFormatStyle = formatNum
End Function
```

Default formats

It is possible to get formats of a specific type without manually specifying the format using the getStandardFormat method on the Formats object. Specify the desired format using the

NumberFormat constant group. Consider this snippet to set cells in Calc to the local currency format:

```
oFormats = oDoc.NumberFormats
Dim aLocale As New com.sun.star.lang.Locale
oRange = oSheet.getCellRangeByName("D2:F19")
oRange.NumberFormat = oFormats.getStandardFormat(_
    com.sun.star.util.NumberFormat.CURRENCY, aLocale)
```

Table 153. Constants for the *com.sun.star.util.NumberFormat* constant group.

Value	Name	Description
0	ALL	selects all number formats.
1	DEFINED	selects only user-defined number formats.
2	DATE	selects date formats.
4	TIME	selects time formats.
8	CURRENCY	selects currency formats.
16	NUMBER	selects decimal number formats.
32	SCIENTIFIC	selects scientific number formats.
64	FRACTION	selects number formats for fractions.
128	PERCENT	selects percentage number formats.
256	TEXT	selects text number formats.
6	DATETIME	selects number formats which contain date and time.
1024	LOGICAL	selects boolean number formats.
2048	UNDEFINED	is used as a return value if no format exists.

14.12.2. Create a field master

A *Listing* sequence field is associated to the `com.sun.star.text.FieldMaster.SetExpression.Listing` master field, which must exist before the sequence field can be used. Listing 392 Demonstrates how to create a master field if it does not exist, and how to get the master field if it does.

Listing 392. *Creating a master field.*

```
oMasters = oDoc.getTextFieldMasters
If NOT oMasters.hasByName("com.sun.star.text.FieldMaster.SetExpression.Listing") Then
    oMasterField = oDoc.createInstance("com.sun.star.text.FieldMaster.SetExpression")
    oMasterField.Name = "Listing"
    oMasterField.SubType = com.sun.star.text.SetVariableType.SEQUENCE
Else
    oMasterField = oMasters.getByNam("com.sun.star.text.FieldMaster.SetExpression.Listing")
End If
```

The master field sub-type determines the field type (see Table 154).

Table 154. Constants for *com.sun.star.text.SetVariableType*.

Value	Constant	Description
0	<code>com.sun.star.text.SetVariableType.VAR</code>	Simple variable.
1	<code>com.sun.star.text.SetVariableType.SEQUENCE</code>	Number sequence field.
2	<code>com.sun.star.text.SetVariableType.FORMULA</code>	Formula field.
3	<code>com.sun.star.text.SetVariableType.STRING</code>	String field.

14.12.3. Insert a sequence field

The sequence field is created by the document, then the numbering type is set to arabic. The number format is set, the master field is associated to the field (the master field is a sequence field), and then the field content is set to increment the value (Listing + 1). Note that when the field is inserted, the field may not be immediately updated. Either wait for the field to update, or force an update using **Tools > Update > Fields**.

Listing 393. Creating a set expression sequence field.

```
oField = oDoc.CreateInstance("com.sun.star.text.TextField.SetExpression")
oField.NumberingType = com.sun.star.style.NumberingType.ARABIC
oField.NumberFormat = FindCreateNumberFormatStyle("###0", oDoc)
oField.attachTextFieldMaster(oMasterField)
oField.Content = name & " + 1"
```

Constants used to set the numbering type are set as shown in the Table 150.

14.12.4. Replace text with a sequence field

Finally, the main working macro to find the text and replace it with a sequence field. The location to insert the sequence field is found by searching for a regular expression. The regular expression "`^Listing [:digit:]+\.[[:space:]]+`" searches for a line that begins with the text “Listing”, followed by a space, one or more numeric digits, a period, and one or more spaces; for example, “Listing 12.”.

Listing 394. Replace text with sequence fields.

```
Function FindReplaceTextFields(oDoc, name$) As Integer
    Dim oDescriptor ' Search descriptor used to find the captions.
    Dim oFound      ' Text matching the search descriptor.
    Dim oMasters    ' Master fields in this document.
    Dim oMasterField ' The associated master field.
    Dim oFrame      ' Documents main frame, used for dispatches.
    Dim oDispatcher ' Dispatcher object.
    Dim oCurs       ' Text cursor used to insert text content.
    Dim oField      ' Sequence field to insert.
    Dim formatNum&  ' ID (Key) for the format for the field.
    Dim i As Integer ' General index variable.

    'Assume that no text fields are inserted.
    FindReplaceTextFields = 0

    'What number format to use for the field.
```

```

formatNum = FindCreateNumberFormatStyle("###0", oDoc)

' Create the master field if needed.
oMasters = oDoc.getTextFieldMasters
If NOT oMasters.hasByName("com.sun.star.text.FieldMaster.SetExpression." & name) Then
    oMasterField = oDoc.createInstance("com.sun.star.text.FieldMaster.SetExpression")
    oMasterField.Name = name
    oMasterField.SubType = com.sun.star.text.SetVariableType.SEQUENCE
Else
    oMasterField = oMasters.getByByName("com.sun.star.text.FieldMaster.SetExpression." & name)
End If

'Search for a line beginning with the name, a space, a number, a period, and a space.
oDescriptor = oDoc.createSearchDescriptor()
With oDescriptor
    .SearchString = "^" & name & " [:digit:]+\.[[:space:]]+"
    .SearchRegularExpression = True
End With

'Prepare to issue a dispatch to clear formatting.
oFrame = oDoc.CurrentController.Frame
oDispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

'Begin the find process.
oFound = oDoc.findFirst(oDescriptor)
Do While Not IsNull(oFound)
    ' Use the strong emphasis character style for "Listing 7."
    oFound.CharStyleName = "OOoStrongEmphasis"

    ' Create and configure the sequence field.
    oField = oDoc.createInstance("com.sun.star.text.TextField.SetExpression")
    oField.NumberingType = com.sun.star.style.NumberingType.ARABIC
    oField.NumberFormat = formatNum
    oField.attachTextFieldMaster(oMasterField)
    oField.Content = name & " + 1"

    ' Create a text cursor with the found text.
    oCurs = oFound.getText().createTextCursorByRange(oFound)

    ' Insert the text such as "Listing <sequence variable>."
    ' This inserted text replaces the found text.
    oFound.getText().insertString(oCurs, name & " ", True)
    oCurs.collapseToEnd()
    oFound.getText().insertTextContent(oCurs, oField, True)
    oCurs.collapseToEnd()
    oFound.getText().insertString(oCurs, ".", True)
    oCurs.collapseToEnd()
    oFound.getText().insertString(oCurs, " ", True)
    oCurs.collapseToEnd()

    ' The inserted text uses strong emphasis.
    ' Move the cursor back one so that it is before the space.
    ' Jump to the end of the paragraph and reset the attributes.

```

```

oCurs.goLeft(1, False)
oCurs.gotoEndofParagraph(True)
oDoc.CurrentController.Select(oCurs)
oDispatcher.executeDispatch(oFrame, ".uno:ResetAttributes", "", 0, Array())

' Find the next occurrence.
oFound = oDoc.findNext(oFound.End, oDescriptor)
i = i + 1
Loop
FindReplaceTextFields = i
End Function

```

The above method contains extra code that clears formatting after the main sequence variables.

14.12.5. Create a GetReference field

An inserted sequence field is directly associated to a field master, which may be referenced by many different fields; every listing in this book references a single field master. A GetReference field, however, does not reference a field as easily. When a field is inserted, OOo automatically assigns a sequence value. The GetReference field has a SequenceNumber attribute that takes the numeric SequenceValue from the referenced field (see Listing 395).

The reference field source indicates the type of field that is referenced. The supported types are shown in Table 156. Much of this is poorly documented, and I speculate that the sequence value may be unique relative to field source.

The ReferenceFieldPart indicates how the reference is displayed. Valid values are shown in Table 155. The example in Listing 395 uses the category and the number. In a caption, all text before the number is the category.

Listing 395. Inserting a GetReference field.

```

oField = oDoc.createInstance("com.sun.star.text.textfield.GetReference")
oField.ReferenceFieldPart = com.sun.star.text.ReferenceFieldPart.CATEGORY_AND_NUMBER
oField.ReferenceFieldSource = com.sun.star.text.ReferenceFieldSource.SEQUENCE_FIELD
oField.SequenceNumber = oReferencedField.SequenceValue
oField.SourceName = sSeqName
oText.insertTextContent(oCurs, oField, True)

```

Table 155. Constants for *com.sun.star.text.ReferenceFieldPart*.

Value	Constant	Description
0	PAGE	The page number is displayed using Arabic numbers.
1	CHAPTER	The chapter number is displayed.
2	TEXT	The reference text is displayed.
3	UP_DOWN	Use localized words for "above" or "below".
4	PAGE_DESC	The page number is displayed using the numbering type defined in the page style of the reference position.
5	CATEGORY_AND_NUMBER	The category and the number of a caption is displayed; for example, Listing 7.
6	ONLY_CAPTION	The caption text of a caption is displayed.
7	ONLY_SEQUENCE_NUMBER	The number of a sequence field is displayed.
8	NUMBER	The numbering label and depending of the reference field context numbering labels of superior list levels of the reference are displayed.
9	NUMBER_NO_CONTEXT	The numbering label of the reference is displayed.
10	NUMBER_FULL_CONTEXT	The numbering label and numbering labels of superior list levels of the reference are displayed.

Table 156. Constants for *com.sun.star.text.ReferenceFieldSource*.

Value	Constant	Description
0	REFERENCE_MARK	The source is a reference mark.
1	SEQUENCE_FIELD	The source is a number sequence field.
2	BOOKMARK	The source is a bookmark.
3	FOOTNOTE	The source is a footnote.
4	ENDNOTE	The source is an endnote.

14.12.6. Replace text with a **GetReference** field

Listing 396 accepts an array of strings and a string to find in the array. The function returns the index of the string in the array. The need for this macro is explained below.

Listing 396. Find a string in an array.

```
Function FindStringInArray(oData, s) As Integer
    Dim i As Integer
    FindStringInArray = -1
    For i = LBound(oData) To UBound(oData)
        If oData(i) = s Then
            FindStringInArray = i
            Exit Function
        End If
    Next
End Function
```

The inner workings of the main macro are described in the comments of the macro itself (see Listing 397).

Listing 397. Replace text with a GetReference field.

```
Function referenceSequenceVariables(oDoc, sSeqName) As Integer
    Dim oEnum          ' Enumeration of text sections.
    Dim oField         ' Enumerated text field.
    Dim s$             ' Generic string variable.
    Dim oFrame         ' Documents main frame, used for dispatches.
    Dim oDispatcher    ' Dispatcher object.
    Dim oFields()      ' Set expressions that may be referenced.
    Dim sPresentations() ' Text of what is displayed; for example, "Listing 7"
    Dim i As Integer   ' Location of a reference (such as "Listing 7") in sPresentations.
    Dim n As Integer   ' Count the fields.
    Dim oCurs          ' Cursor used to clear formatting.
    Dim oDescriptor     ' The search descriptor.
    Dim oFound         ' The found range.

    ' Default to no references created.
    referenceSequenceVariables = 0

    ' Setup to perform a dispatch.
    oFrame = oDoc.CurrentController.Frame
    oDispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

    ' Enumerate the text fields and identify the appropriate set field expressions.
    ' The field presentation is the number as it is displayed.
    ' When this section is finished:
    ' oFields() - All the set expression fields for the specified name.
    ' sPresentations() - Contains the full reference such as "Listing 7" for the listing
    '                  sequence with the value 7.
    oEnum = oDoc.getTextFields().createEnumeration()
    If Not IsNull(oEnum) Then
        Do While oEnum.hasMoreElements()
            oField = oEnum.nextElement()
            If oField.supportsService("com.sun.star.text.TextField.SetExpression") Then
                If oField.VariableName = sSeqName Then
                    ReDim Preserve oFields(0 To n)
                    ReDim Preserve sPresentations(0 To n)
                    oFields(n) = oField
                    sPresentations(n) = sSeqName & " " & oField.CurrentPresentation
                    n = n + 1
                End If
            End If
        Loop
    End If

    ' Create the search descriptor to find all instances of things such as "Listing 23".
    ' Values in fields are not found, because search does NOT search fields.
    oDescriptor = oDoc.createSearchDescriptor()
    With oDescriptor
        .SearchString = sSeqName & " [:digit:]+"
    End With
End Function
```



```

        .SearchRegularExpression = True
End With

n = 0
oFound = oDoc.findFirst(oDescriptor)
Do While Not IsNull(oFound)

    ' Look for a field with the text such as "Listing 7"
    i = FindStringInArray(sPresentations, oFound.getString())
    If i >= 0 Then
        ' Create and configure the GetReference field
        oField = oDoc.createInstance("com.sun.star.text.textfield.GetReference")
        oField.ReferenceFieldPart = com.sun.star.text.ReferenceFieldPart.CATEGORY_AND_NUMBER
        oField.ReferenceFieldSource = com.sun.star.text.ReferenceFieldSource.SEQUENCE_FIELD
        oField.SequenceNumber = oFields(i).SequenceValue
        oField.SourceName = sSeqName

        ' Create a text cursor where the text was found.
        ' Clear the text, then insert the GetReference field at that location.
        oCurs = oFound.getText().createTextCursorByRange(oFound)
        oCurs.setString("")
        oFound.getText().insertTextContent(oCurs, oField, True)

        ' Sometimes, special formatting was used for the reference text.
        ' A cursor cannot move into a field, so collapsing the cursor then selecting
        ' one character to the left will select the GetReference field.
        oCurs.collapseToEnd()
        oCurs.goLeft(1, True)

        ' Select the GetReference field on the display, then use a
        ' dispatch to clear all attributes.
        oDoc.CurrentController.select(oCurs)
        oDispatcher.executeDispatch(oFrame, ".uno:ResetAttributes", "", 0, Array())
        n = n + 1
    Else
        s = s & "Failed to find (" & oFound.getString() & ")" & CHR$(10)
    End If
    oFound = oDoc.findNext(oFound.End, oDescriptor)
Loop
referenceSequenceVariables = n
If s <> "" Then
    MsgBox s
End If
End Function

```

14.12.7. The worker that ties it all together

The main worker macro starts by replacing captions for Figure, Listing, and Table with sequence variables. It is assumed, but never verified, that the captions are sequentially numbered as they appear in the document with no gaps in numbering. In other words, the routines are a bit fragile, but they worked great for the documents that I desired to use.

My first test set about 100 captions. All cross-references failed because the fields had not yet updated. After creating all sequence fields, a dispatch causes the fields to update so that the next phase can begin. Note that if this is not done, then the CurrentPresentation attribute on the sequence fields will be incorrect.

Listing 398. *Set captions and references for a document.*

```
sub replaceCaptions()
    Dim oDoc          ' Document on which to operate.
    Dim oFrame        ' Documents main frame, used for dispatches.
    Dim oDispatcher    ' Dispatcher object.
    Dim i As Integer  ' General index variable.
    Dim n As Integer  ' Number of fields created.
    Dim s$            ' Summary of work performed.
    Dim fields()      ' Field names on which to operate.

    fields = Array("Listing", "Table", "Figure")
    oDoc = ThisComponent
    oFrame = oDoc.CurrentController.Frame
    oDispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

    For i = LBound(fields()) To UBound(fields())
        n = FindReplaceTextFields(ThisComponent, fields(i))
        s = s & "Fixed " & n & " captions for " & fields(i) & CHR$(10)
    Next

    ' This next part will NOT work until after all of the fields have updated.
    ' So, force an update.
    oDispatcher.executeDispatch(oFrame, ".uno:UpdateFields", "", 0, Array())

    For i = LBound(fields()) To UBound(fields())
        n = referenceSequenceVariables(ThisComponent, fields(i))
        s = s & "set " & n & " references to " & fields(i) & CHR$(10)
    Next
    MsgBox s
End Sub
```

14.13. Table of contents

Finding and inserting a table of contents (TOC) is easy unless you want to change the defaults. The following macro checks ThisComponent to see if the document contains a content index.

```
REM Find TOC if it exists.
oIndexes = ThisComponent.getDocumentIndexes()
bIndexFound = False
For i = 0 To oIndexes.getCount() - 1
    oIndex = oIndexes.getByIndex(i)
    If oIndex.supportsService("com.sun.star.text.ContentIndex") Then
        bIndexFound = True
        Exit For
    End If
Next
```

Use dispose to remove an existing index from the document.

When I create a TOC, I usually set CreateFromOutline to true to create the index based on the document outline styles (see Listing 399).

Listing 399. *Insert a standard TOC into a document.*

```
Sub InsertATOC
    REM Author: Andrew Pitonyak
    Dim oCurs          'Used to insert the text content.
    Dim oIndexes        'All of the existing indexes
    Dim oIndex          'TOC if it exists and a new one if not
    Dim i%              'Find an existing TOC
    Dim bIndexFound As Boolean 'Flag to track if the TOC was found

    REM Find TOC if it exists.
    oIndexes = ThisComponent.getDocumentIndexes()
    bIndexFound = False
    For i = 0 To oIndexes.getCount() - 1
        oIndex = oIndexes.getByIndex(i)
        If oIndex.supportsService("com.sun.star.text.ContentIndex") Then
            bIndexFound = True
            Exit For
        End If
    Next
    If Not bIndexFound Then
        Print "I did not find an existing content index"
        REM Create and insert a new TOC.
        REM The new TOC must be created by the document that will contain the TOC.
        oIndex = ThisComponent.createInstance("com.sun.star.text.ContentIndex")

        oIndex.CreateFromOutline = True
        oCurs = ThisComponent.getText().createTextCursor()
        oCurs.gotoStart(False)
        ThisComponent.getText().insertTextContent(oCurs, oIndex, False)
    End If
    REM Even the newly inserted index is not updated until right HERE!
    oIndex.update()
End Sub
```

The same code can be used to create a DocumentIndex, ContentIndex, UserDefinedIndex, IllustrationIndex, TableIndex, or ObjectIndex. Properties common to all index types are shown in Table 157.

Table 157. Common index properties.

Property	Description
BackColor	Background color. Set to -1 for none.
BackGraphicFilter	Filter for the graphic displayed as background graphic.
BackGraphicLocation	Location of the back graphic. Set using the com.sun.star.style.GraphicLocation constants. This includes NONE, LEFT_TOP, MIDDLE_TOP, RIGHT_TOP, LEFT_MIDDLE, MIDDLE_MIDDLE, RIGHT_MIDDLE, LEFT_BOTTOM, MIDDLE_BOTTOM, RIGHT_BOTTOM, AREA, and TILED
BackGraphicURL	URL to the graphic displayed as a background graphic. I did not experiment with this, it may require an internal URL??
CreateFromChapter	Set to True to create an index based on the current chapter rather than the entire document.
IsProtected	If true, the index is protected and cannot be edited as part of the text.
ParaStyleHeading	Paragraph style name used for the heading.
ParaStyleLevel1	Paragraph style named used for level 1. Note that there is an attribute for levels 1 through 10.
Title	Index title that is inserted as part of the TOC. I usually leave this blank and place the title before the TOC using the “Heading 1” paragraph style so that it is included as part of the TOC; silly, I know.

A TOC contains columns of data; for example, section numbering, section title, and page number. Each column is represented by an array of named properties. The supported properties are shown Table 158.

Table 158. Supported column tokens.

Token	Description
TokenType	Identifies the contents of the column. Every column has a token type as the first property. TokenType property values are strings shown in Table 159.
CharacterStyleName	Name of the character style applied to the element. Do not include for tab stops. An empty value causes the Default style to be used.
TabStopRightAligned	Tab stop is right aligned. Only valid for tab stops.
TabStopPosition	Position of the tab stop. Only valid for tab stops.
TabStopFillCharacter	Fill character in tab stops. Only valid for tab stops.
WithTab	If true insert tab character.
Text	Only valid in user defined text.
ChapterFormat	Valid in chapter info and entry number only. The chapter format is also mentioned in Table 149. For chapter info NUMBER and NAME_NUMBER are allowed and for an entry NUMBER and DIGIT are allowed.
ChapterLevel	Valid in chapter info and entry number only. Denotes the level up to which the chapter information is given. Values permitted 1 to 10 inclusive.

Some token types are not supported by all index types. The Value is the string name used to identify the token type. The Entry column contains the text used in the GUI while specifying the

columns. If an entry is empty, it is because I did not verify every type to see what values are used and I do not know the value.

Table 159. *Supported TokenType values.*

Value	Entry	Comment	Supported Index Types
TokenEntryNumber	E#	Chapter number	Content
TokenEntryText	E	Entry text	All
TokenTabStop	T	Tab stop	All
TokenText		User defined text	
TokenPageNumber	#	page number	All
TokenChapterInfo		Chapter information	Illustration, Table, User, Table of Objects, and Alphabetical
TokenHyperlinkStart	LS	Begin a hyperlink	
TokenHyperlinkEnd	LE	End a hyperlink	
TokenBibliographyDataField		Bibliographic data field	

The macro in Listing 400 inspects the level formatting for the TOC in the current document and then inserts text at the end of the current document.

Listing 400. *Inspect the TOC in the current document.*

```
Sub InspectCurrentTOCColumns
    'Inspect ThisComponent
    Dim oCurs          'Used to insert the text content.
    Dim oIndexes       'All of the existing indexes
    Dim oIndex         'TOC if it exists and a new one if not
    Dim i%             'Find an existing TOC
    Dim bIndexFound As Boolean 'Flag to track if the TOC was found
    Dim iLevel%        ' Iterate over the levels in the LevelFormat property.
    Dim iCol%          ' Iterate over the columns in each level.
    Dim iProp%         ' Iterate over the properties for a single column.
    Dim oLevel         ' Level object.
    Dim oCol           ' Column object.
    Dim s$

    REM Find TOC if it exists.
    oIndexes = ThisComponent.getDocumentIndexes()
    bIndexFound = False
    For i = 0 To oIndexes.getCount() - 1
        oIndex = oIndexes.getByIndex(i)
        If oIndex.supportsService("com.sun.star.text.ContentIndex") Then
            bIndexFound = True
            Exit For
        End If
    Next
    If Not bIndexFound Then
        Exit Sub
    End If
```

```

s = "Level" & CHR$(9) & "Column" & CHR$(9) & "Property" & CHR$(9) & "Name" & _
    CHR$(9) & "Value" & CHR$(13)
For iLevel = 0 To oIndex.LevelFormat.getCount() - 1
    oLevel = oIndex.LevelFormat.getByIndex(iLevel)
    For iCol = LBound(oLevel) To UBound(oLevel)
        oCol = oLevel(iCol)
        For iProp = LBound(oCol) To UBound(oCol)
            s = s & iLevel & CHR$(9) & iCol & CHR$(9) & iProp & CHR$(9) & _
                oCol(iProp).Name & CHR$(9) & oCol(iProp).Value & CHR$(13)
        Next
    Next
Next
ThisComponent.getText().insertString(ThisComponent.getText().End, s, False)
End Sub

```

I ran the macro in Listing 400 with OOo version 3.3.0 and noticed that level 0 contains no entries, and levels 1 – 10 are identical. The output for level 1 is shown in Table 160.

- Column 0 is the TokenEntryNumber formatted with the Default character style. In the TOC, this is the chapter numbering.
- A hyperlink begins in column 1, and it is formatted using the “Internet Link” character style.
- Column 2 contains the heading text. No character style is provided, but it is formatted using the “Internet Link” style because it is included in the hyperlink.
- Column 3 specifies the end of the hyperlink, so the hyperlink includes only the heading text.
- Column 4 specifies a right aligned tab stop that fills the empty space between the heading text and the page number with periods.
- Column 5 contains the page number.

Table 160. Level format for level 1.

Level	Column	Property	Name	Value
1	0	0	TokenType	TokenEntryNumber
1	0	1	CharacterStyleName	
1	1	0	TokenType	TokenHyperlinkStart
1	1	1	CharacterStyleName	Internet link
1	2	0	TokenType	TokenEntryText
1	2	1	CharacterStyleName	
1	3	0	TokenType	TokenHyperlinkEnd
1	4	0	TokenType	TokenTabStop
1	4	1	TabStopRightAligned	TRUE
1	4	2	TabStopFillCharacter	.
1	4	3	CharacterStyleName	
1	4	4	WithTab	TRUE
1	5	0	TokenType	TokenPageNumber
1	5	1	CharacterStyleName	

While creating a TOC manually, the columns are edited to include things such as chapter numbers, chapter text, hyperlink start, hyperlink end, page number, and other things. These values are stored in the `LevelFormat` property.

Listing 401. Insert a TOC with hyperlinks.

```

Sub InsertATOCWithHyperlinks
    REM Author: Andrew Pitonyak
    Dim oCurs      'Used to insert the text content.
    Dim oIndexes   'All of the existing indexes.
    Dim oIndex     'TOC if it exists and a new one if not.
    Dim i%         'Find an existing TOC.
    Dim bIndexFound As Boolean 'Flag to track if the TOC was found.
    Dim iLevel

    REM Find TOC if it exists.
    oIndexes = ThisComponent.getDocumentIndexes()
    bIndexFound = False
    For i = 0 To oIndexes.getCount() - 1
        oIndex = oIndexes.getByIndex(i)
        If oIndex.supportsService("com.sun.star.text.ContentIndex") Then
            bIndexFound = True
            Exit For
        End If
    Next
    If Not bIndexFound Then
        REM Create and insert a new TOC.
        REM The new TOC must be created by the document that will contain the TOC.
        oIndex = ThisComponent.createInstance("com.sun.star.text.ContentIndex")
    
```

```

oIndex.CreateFromOutline = True

' Ignore level 0
For iLevel = 1 To oIndex.LevelFormat.getCount() - 1
    oIndex.LevelFormat.replaceByIndex(iLevel, CreateTOCColumnEntries())
Next

oCurs = ThisComponent.getText().createTextCursor()
oCurs.gotoRange(ThisComponent.CurrentController.ViewCursor, False)
ThisComponent.getText().insertTextContent(oCurs, oIndex, False)
End If
REM Even the newly inserted index is not updated until right HERE!
oIndex.update()
End Sub

Function CreateTOCColumnEntries()
    Dim o
    o = Array( Array(MakeProperty("TokenType", "TokenEntryNumber"), _
        MakeProperty("CharacterStyleName", "")), _
        Array(MakeProperty("TokenType", "TokenHyperlinkStart"), _
        MakeProperty("CharacterStyleName", "Internet link")), _
        Array(MakeProperty("TokenType", "TokenEntryText"), _
        MakeProperty("CharacterStyleName", "")), _
        Array(MakeProperty("TokenType", "TokenHyperlinkEnd")), _
        Array(MakeProperty("TokenType", "TokenTabStop"), _
        MakeProperty("TabStopRightAligned", TRUE), _
        MakeProperty("TabStopFillCharacter", "."), _
        MakeProperty("CharacterStyleName", ""), _
        MakeProperty("WithTab", TRUE)), _
        Array(MakeProperty("TokenType", "TokenPageNumber"), _
        MakeProperty("CharacterStyleName", "")))
    CreateTOCColumnEntries() = o
End Function

Function MakeProperty(sName$, value)
    Dim oProp
    oProp = CreateObject("com.sun.star.beans.PropertyValue")
    oProp.Name = sName
    oProp.Value = value
    MakeProperty = oProp
End Function

```

14.14. Conclusion

Although this chapter didn't cover every object and capability supported by Writer, it did discuss the capabilities most frequently questioned on the mailing lists. Many of the techniques demonstrated in this chapter are representative of the techniques that are required for objects not discussed. For example, all objects supporting named access are retrieved in the same manner. Use the material covered here as a starting point in your exploration of Writer documents in OpenOffice.org.

15. Calc Documents

The primary purpose of a Calc document is to contain multiple spreadsheets, which in turn contain rows and columns of data — in other words, tables. This chapter introduces appropriate methods to manipulate, traverse, format, and modify the content contained in a Calc document.

OpenOffice.org supports three primary table types: text tables in Writer documents, database tables, and spreadsheets in Calc documents. Each of the different table types is tailored for a specific purpose. Text tables in Writer documents support complex text formatting but only simple table calculations. Spreadsheet documents, on the other hand, support complex calculations and only simple text formatting.

Conceptually, all document types have two components: the data they contain and the controller that determines how the data is displayed. In OpenOffice.org, the collection of data contained in a document is called the *model*. Each model has a controller that is responsible for the visual presentation of the data. The controller knows the location of the visible text cursor and the current page, and knows what is currently selected.

Every Calc document supports the `com.sun.star.sheet.SpreadsheetDocument` service. When I write a macro that must be user-friendly and requires a spreadsheet document, I verify that the document is the correct type by using the object method `supportsService()`. See Listing 402.

Listing 402. *Calc documents support the `com.sun.star.sheet.SpreadsheetDocument` service.*

```
REM A trick is used to avoid a bug in OOo.
REM This function tests the argument to see if it supports a service.
REM If the object does not support a service, a runtime error
REM occurs that complains that the variable is not set.
REM Assigning the argument to a temporary variable and
REM using that to call SupportsService avoids the error.
Function isCalcDocument(oDoc) As Boolean
    On Error Goto ErrorJumpPoint
    Dim s$ : s$ = "com.sun.star.sheet.SpreadsheetDocument"

    isCalcDocument = False
    If oDoc.SupportsService(s$) Then
        isCalcDocument = True
    End If
    ErrorJumpPoint:
End Function
```

An interface defines a series of methods. If an object implements an interface, it implements every method defined by that interface. A service defines an object by specifying the interfaces that it implements, the properties that it contains, and the other services that it exports. A service indirectly specifies the implemented methods by specifying interfaces. The interfaces supported by Calc documents provide a good overview of the provided functionality (see Table 161).

Table 161. Some interfaces supported by Calc documents.

Service	Description
com.sun.star.document.XActionLockable	Temporarily lock the document from user interaction and automatic cell updates. Locking an object makes it possible to prevent internal object updates while you quickly change multiple parts of the objects that might temporarily invalidate each other.
com.sun.star.drawing.XDrawPagesSupplier	Access all draw pages in this document; there is one draw page for each contained sheet.
com.sun.star.sheet.XCalculatable	Control automatic calculation of cells.
com.sun.star.sheet.XConsolidatable	Perform data consolidation.
com.sun.star.sheet.XGoalSeek	Perform a “Goal Seek” for a cell.
com.sun.star.sheet.XSpreadsheetDocument	Access the contained spreadsheets.
com.sun.star.style.XStyleFamiliesSupplier	Access the contained styles by type.
com.sun.star.util.XNumberFormatsSupplier	Access the number formats.
com.sun.star.util.XProtectable	Protect and unprotect the document.

The `createNewCalcDoc` function is used to create a new empty Calc document. This method is used by other methods in this chapter.

Listing 403. Create a new Calc document.

```
Function createNewCalcDoc
    Dim noArgs()           'An empty array for the arguments
    Dim sURL As String     'URL of the document to load
    Dim oDoc

    sURL = "private:factory/scalc"
    oDoc = StarDesktop.LoadComponentFromUrl(sURL, "_blank", 0, noArgs())
    createNewCalcDoc = oDoc
End Function
```

15.1. Accessing sheets

The primary purpose of a spreadsheet document is to act as a container for individual sheets through the `XSpreadsheetDocument` interface. The `XSpreadsheetDocument` interface defines the single method `getSheets()` that returns a `Spreadsheets` object used to manipulate the individual sheets (see Listing 404).

Listing 404. Obtain a `com.sun.star.sheet.Spreadsheets` service using a method or a property.

```
ThisComponent.getSheets() 'Method defined by XSpreadsheetDocument interface.
ThisComponent.Sheets      'Property of the spreadsheet document.
```

The `Spreadsheets` service allows the individual sheets to be returned by index, by enumeration, and by name (see Table 162). The `Spreadsheets` service also allows sheets to be created, moved, and deleted. Many of the methods shown in Table 162 are demonstrated in Listing 405.

Table 162. *Methods implemented by the com.sun.star.sheet.Spreadsheets service.*

Method	Description
copyByName(srcName, destName, index)	Copy the sheet named srcName to the specified index and name it destName.
createEnumeration()	Create an object that enumerates the spreadsheets.
getByIndex(index)	Obtain a spreadsheet based on the sheet's index.
getByName(name)	Obtain a spreadsheet based on the sheet's name.
getCount()	Return the number of sheets as a Long Integer.
hasByName(name)	Return True if the named spreadsheet exists.
hasElements()	Return True if the document contains at least one spreadsheet.
insertNewByName(name, index)	Create a new spreadsheet and insert it at the specified location with the supplied name.
moveByName(name, index)	Move the named spreadsheet to the specified index.
removeByName(name)	Remove the named spreadsheet.

Listing 405. *Manipulate sheets in a Calc document.*

```

Sub AccessSheets
    Dim oSheets          'The sheets object that contains all of the sheets
    Dim oSheet           'Individual sheet
    Dim oSheetEnum       'For accessing by enumeration
    Dim s As String      'String variable to hold temporary data
    Dim i As Integer     'Index variable
    Dim oDoc

    oDoc = createNewCalcDoc()
    oSheets = oDoc.Sheets

    REM Insert new sheet as the second sheet.
    oSheets.insertNewByName("CreatedSheet", 1)

    REM Create a new sheet named "First" at the start.
    oSheets.insertNewByName("First", 0)

    REM Verify that the sheet named "Sheet3" exists
    If oSheets.hasByName("Sheet3") Then
        oSheet = oSheets.getByIndex(3)
        oSheet.getCellByPosition(0, 0).setString("Test")

        REM Copy "Sheet3" to the end. That is copy, not move!
        oSheets.copyByName("Sheet3", "Copy1", oSheets.getCount())
    End If

    If oSheets.hasByName("Sheet1") Then
        oSheets.removeByName("Sheet1")
    End If

```

```

REM The sheets are indexed starting at zero, but getCount() indicates
REM exactly how many sheets there are.
For i = 0 To oSheets.getCount()-1
    s = s & "Sheet " & i & " = " & oSheets.getByIndex(i).Name & CHR$(10)
Next
Msgbox s, 0, "After Inserting New Sheets"

REM Now remove the new sheets that I inserted
oSheets.removeByName("First")
oSheets.removeByName("Copy1")

s = "" : i = 0
oSheetEnum = oSheets.createEnumeration()
Do While oSheetEnum.hasMoreElements()
    oSheet = oSheetEnum.nextElement()
    s = s & "Sheet " & i & " = " & oSheet.Name & CHR$(10)
    i = i + 1
Loop
Msgbox s, 0, "After Deleting Sheets"
End Sub

```

15.2. Sheet cells contain the data

A spreadsheet document contains individual sheets that are composed of rows and columns of cells. Each column is labeled alphabetically starting with the letter A, and each row is labeled numerically starting with the number 1. A cell can be identified by its name, which uses the column letter and the row number, or by its position. The upper-left cell is “A1” at position (0, 0) and cell “B3” is at location (1, 2).

The Calc user interface identifies cells with names such as “Sheet2.D5”. A cell, on the other hand, identifies itself by the row and column offset, which requires a bit of work to turn into human-readable form. The CellAddressConversion service converts a cell address into human readable form. The CellAddressConversion service is not yet documented, so I performed some tests. When a cell address is assigned to the Address property (See Listing 406), the PersistentRepresentation property is set to the full cell name including the sheet name. The UserInterfaceRepresentation property, however, contains the sheet name only if the cell is not contained in the active sheet.

Listing 406. Obtain the cell name using the CellAddressConversion service.

```

Dim oConv
Dim oCell
oConv = oDoc.createInstance("com.sun.star.table.CellAddressConversion")
oCell = ThisComponent.Sheets(2).getCellByPosition(0, 0) 'Cell A1
oConv.Address = oCell.getCellAddress()
Print oConv.UserInterfaceRepresentation 'A1
print oConv.PersistentRepresentation 'Sheet1.A1

```

I had difficulty with the CellAddressConversion service, so I opted to write my own. The following method accepts the sheet number, column, and row, and returns an address. The “cheat” that I used, is to assume that the sheet name is formatted as Sheet1, Sheet2, etc. This is trivial to change, but I opted to not do it. The only tricky part is to understand that columns are labeled

using Base 26 where the digits are A-Z and not 0-9; and being enough of a technical person to understand what that means.

Listing 407. *Format a single cell address assuming all sheets are named Sheet.*

```
Function AddressString(iSheet As Long, iCol As Long, iRow As Long, bWwithSheet As Boolean)
    Dim s$
    iCol = iCol + 1
    Do
        iCol = iCol - 1
        s = CHR$((iCol MOD 26) + 65) & s
        iCol = iCol \ 26 - 1
    Loop Until iCol < 0
    If bWwithSheet Then
        AddressString = "Sheet" & CStr(iSheet + 1) & "." & s & CStr(iRow + 1)
    Else
        AddressString = s & CStr(iRow + 1)
    End If
End Function
```

This next function accepts a range address.

Listing 408. *Format a single range address assuming all sheets are named Sheet.*

```
Function prettyRangeAddressName(oRangeAddr)
    Dim s1$, s2$
    Dim oConv
    Dim oCellAddr As New com.sun.star.table.CellAddress

    s1 = AddressString(oRangeAddr.Sheet, oRangeAddr.StartColumn, oRangeAddr.StartRow, True)
    s2 = AddressString(oRangeAddr.Sheet, oRangeAddr.EndColumn, oRangeAddr.EndRow, False)
    prettyRangeAddressName = s1 & ":" & s2
End Function
```

15.2.1. Cell address

In OOo, a cell's address is specified by the sheet that contains the cell, and the row and column in which the cell is located. OOo encapsulates a cell's address in a CellAddress structure (see Table 163). The CellAddress structure is available directly from a cell and it is also used as an argument to numerous object methods.

Table 163. *Properties of the com.sun.star.table.CellAddress structure.*

Property	Description
Sheet	Short Integer index of the sheet that contains the cell.
Column	Long Integer index of the column where the cell is located.
Row	Long Integer index of the row where the cell is located.

15.2.2. Cell data

A cell can contain four types of data. Use the method getType() to find out the type of data that it contains. A cell that contains no data is considered empty. A cell can contain a floating-point

Double value. Use the object methods `getValue()` and `setValue(Double)` to get and set a cell's value.

Tip If the cell contains a formula, you can still determine the type of data that is stored in the cell from the `FormulaResultType` property shown in Table 167.

A cell can contain textual data. The standard method of getting and setting textual data is to use the methods `getString()` and `setString(String)`. The real story, however, is that the `com.sun.star.table.Cell` service implements the `com.sun.star.text.XText` interface. The `XText` interface is the primary text interface used in Writer documents, and it allows individual cells to contain very complex data.

Tip Sheet cells support the `com.sun.star.text.XText` interface. It should come as no surprise, therefore, that cells also support the `com.sun.star.text.XTextFieldsSupplier` interface — in case you want to insert special text fields into a cell.

A cell can contain a formula. The methods `getFormula()` and `setFormula(String)` get and set a cell's formula. To determine if a formula contains an error, use the `getError()` method — the Long Integer return value is zero if there is no error. The macro in Listing 409 demonstrates how to inspect a cell's type.

Tip When setting a cell's formula, you must include the leading equals sign (=) and the formula must be in English. To set a formula using your own local language, use the `FormulaLocal` property shown in Table 167.

Listing 409. *Get a string representation of the cell type.*

```
Function GetCellType(oCell) As String
    Select Case oCell.getType()
    Case com.sun.star.table.CellContentType.EMPTY
        GetCellType = "Empty"
    Case com.sun.star.table.CellContentType.VALUE
        GetCellType = "Value"
    Case com.sun.star.table.CellContentType.TEXT
        GetCellType = "Text"
    Case com.sun.star.table.CellContentType.FORMULA
        GetCellType = "Formula"
    Case Else
        GetCellType = "Unknown"
    End Select
End Function
```

Listing 410 demonstrates how to get and set information in a cell. A numeric value, a string, and a formula are set in a cell. After setting each type, information is printed about the cell (see Figure 96). The macro in Listing 410 is very simple but it demonstrates some very important behavior. Inspect the output in Figure 96 to see what is returned by `getString()`, `getValue()`, and `getFormula()` for each different cell content type.

Listing 410. Get cell information.

```
Function SimpleCellInfo(oCell) As String
    SimpleCellInfo = oCell.AbsoluteName & " has type " & _
        GetCellType(oCell) & " String(" & oCell.getString() & ") Value(" & _
        oCell.getValue() & ") Formula(" & oCell.getFormula() & ")"
End Function

Sub GetSetCells
    Dim oCell
    Dim s As String
    Dim oDoc

    oDoc = createNewCalcDoc()

    oCell = oDoc.Sheets(0).getCellByPosition(0, 0) 'Cell A1
    oCell.setString("Andy")
    oCell = oDoc.Sheets(0).getCellByPosition(104, 0) 'Cell DA1
    s = SimpleCellInfo(oCell) & CHR$(10)
    oCell.setValue(23.2)
    s = s & SimpleCellInfo(oCell) & CHR$(10)
    oCell.setString("4")
    s = s & SimpleCellInfo(oCell) & CHR$(10)
    oCell.setFormula("=A1")
    s = s & SimpleCellInfo(oCell) & CHR$(10)
    oCell.setFormula("")
    s = s & SimpleCellInfo(oCell) & CHR$(10)
    MsgBox s, 0, "Cell Values And Types"
End Sub
```

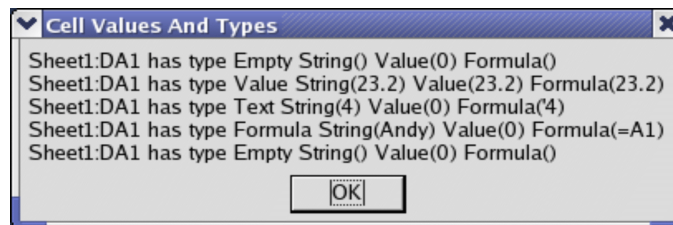


Figure 96. Values returned by `getType()`, `getString()`, `getValue()`, and `getFormula()` for different types of content.

Tip

The methods `getString()` and `getFormula()` return relevant values even when the cell type is not String or Formula (see Figure 96). Notice also that setting a string value of 4 does not set a numeric value, and the formula even shows a single quotation mark in front of the “4”. This provides a pretty good clue that you can also do things such as `setFormula("I am text")` to set text.

15.2.3. Cell properties

Cells contained in a sheet are defined by the `com.sun.star.sheet.SheetCell` service, which supports numerous properties for formatting the cell contents. Given that a cell also supports the Text service, it is of no surprise that it also supports properties related to text content: `CharacterProperties`, `CharacterPropertiesAsian`, `CharacterPropertiesComplex`, and `ParagraphProperties`. There are also cell-specific properties such as setting border lines. You

specify the border lines for cells by using a `BorderLine` structure as shown in Table 164. The `BorderLine` structure defines how a single border line is displayed, and the `TableBorder` structure defines how the lines in a given range are displayed (see Table 165).

Table 164. *Properties of the `com.sun.star.table.BorderLine` structure.*

Property	Description
Color	Line color as a Long Integer.
InnerLineWidth	Width of the inner part of a double line (in 0.01 mm) as a Short Integer — zero for a single line.
OuterLineWidth	Width of a single line, or width of the outer part of a double line (in 0.01 mm) as a Short Integer.
LineDistance	Distance between the inner and outer parts of a double line (in 0.01 mm) as a Short Integer.

Table 165. *Properties of the `com.sun.star.table.TableBorder` structure.*

Property	Description
TopLine	Line style at the top edge (see Table 164).
IsTopLineValid	If True, the TopLine is used when setting values.
BottomLine	Line style at the bottom edge (see Table 164).
IsBottomLineValid	If True, the BottomLine is used when setting values.
LeftLine	Line style at the left edge (see Table 164).
IsLeftLineValid	If True, the LeftLine is used when setting values.
RightLine	Line style at the right edge (see Table 164).
IsRightLineValid	If True, the RightLine is used when setting values.
HorizontalLine	Line style for horizontal lines between cells (see Table 164).
IsHorizontalLineValid	If True, the HorizontalLine is used when setting values.
VerticalLine	Line style for vertical lines between cells (see Table 164).
IsVerticalLineValid	If True, the VerticalLine is used when setting values.
Distance	Distance between the lines and other contents as a Short Integer.
IsDistanceValid	If True, the Distance is used when setting values.

When setting values in a `TableBorder` structure, not all values are always required. For example, when using a `TableBorder` structure to configure a cell's border, the individual values are used only if the corresponding "Is...Valid" property is set. This provides the ability to set a single value and leave the other values unchanged. If, on the other hand, a `TableBorder` structure is obtained by using a query (meaning you get the value), the flags indicate that not all lines use the same value.

Consider the following macro that sets all four borders on a single range.

Listing 411. *Add cell borders.*

```
Sub SetCalcBorder
    Dim oDoc
    Dim oSheet
```

```

Dim oCells
Dim oBorder

oDoc = StarDesktop.loadComponentFromURL("private:factory/scalc", "_default", 0, Array())
oSheet = oDoc.Sheets(0)
oCells = oSheet.getCellRangeByName("B2:C6")

' Create the structure so that I know the values
' are all zero.
oBorder = CreateUnoStruct("com.sun.star.table.BorderLine")
' If you want to modify an existing border, copy it.
'oBorder = oCells.LeftBorder

' Set color to red.
oBorder.Color = RGB(199, 50, 0)
oBorder.OuterLineWidth = 35
oCells.LeftBorder = oBorder

' Set color to blue.
oBorder.Color = RGB(0, 102, 199)
oBorder.InnerLineWidth = 35

oCells.TopBorder = oBorder
oBorder.LineDistance = 35
oCells.RightBorder = oBorder
oCells.BottomBorder = oBorder

oBorder = oSheet.getCellRangeByName("C2").LeftBorder
Print "Red portion of border = " & Red(oBorder.Color)
End Sub

```

	A	B	C
1			
2			
3			
4			
5			
6			

Figure 97. Cells with borders.

Please note the following:

- The BorderLine structure is copied by value, not reference. To modify the structure, therefore, you must copy the structure to a variable, modify the variable, and then copy the structure back to the left, right, top, or bottom border.
- You can set borders on a cell range as well as an individual cell.

- Although the left border on cell B2 and C2 are both a red line, the red line is not shown in cell C2 (see Figure 97). How does OOo decide which border should be used right or left, top or bottom? Although the behavior is not documented, the following precedence was observed:
 - A border with a line distance is more important than a border without a line distance.
 - Top and left borders are more important than right or bottom borders.

Table 166 contains cell-specific properties. Cell properties use the TableBorder structure to set the border types.

Table 166. Properties supported by the *com.sun.star.table.CellProperties* service.

Property	Description
CellStyle	Optional property; the name of the style of the cell as a String.
CellBackColor	The cell background color as a Long Integer (see IsCellBackgroundTransparent).
IsCellBackgroundTransparent	If True, the cell background is transparent and the CellBackColor is ignored.
HoriJustify	The cell's horizontal alignment as a <i>com.sun.star.table.CellHoriJustify</i> enum: • STANDARD – Default alignment is left for numbers and right for text. • LEFT – Content is aligned to the cell's left edge. • CENTER – Content is horizontally centered. • RIGHT – Content is aligned to the cell's right edge. • BLOCK – Content is justified to the cell's width (but this doesn't seem to work). • REPEAT – Content is repeated to fill the cell.
VertJustify	The cell's vertical alignment as a <i>com.sun.star.table.CellVertJustify</i> enum: • STANDARD – Use the default. • TOP – Align to the upper edge of the cell. • CENTER – Align to the vertical middle of the cell. • BOTTOM – Align to the lower edge of the cell.
IsTextWrapped	If True, the cell's content is automatically wrapped at the right border.
ParaIndent	The indentation of the cell's content (in 0.01 mm) as a Short Integer.
Orientation	If the RotateAngle is zero, this specifies the orientation of the cell's content as an enum of type <i>com.sun.star.table.CellOrientation</i> : • STANDARD – The cell's content is displayed from left to right. • TOPBOTTOM – The cell's content is displayed from top to bottom. • BOTTOMTOP – The cell's content is displayed from bottom to top. • STACKED – Same as TOPBOTTOM but each character is horizontal.
RotateAngle	Defines how much to rotate the cell's content (in 0.01 degrees) as a Long Integer. The entire string is rotated as one unit rather than as individual characters.
RotateReference	Defines the edge where rotated cells are aligned using the same enum as VertJustify.
AsianVerticalMode	If True, only Asian characters use a vertical orientation. In other words, in Asian mode only Asian characters are printed in horizontal orientation if the Orientation property is STACKED; for other orientations, this value is not used.
TableBorder	Description of the cell or cell range border (see Table 165). When used with a single cell, the values set the borders for the single cell. When used with a range of cells, the borders are for the outer edges of the range rather than the individual cells.

Property	Description
TopBorder	Description of the cell's top border line (see Table 164).
BottomBorder	Description of the cell's bottom border line (see Table 164).
LeftBorder	Description of the cell's left border line (see Table 164).
RightBorder	Description of the cell's right border line (see Table 164).
NumberFormat	Index of the cell's number format. The proper value can be determined by using the com.sun.star.util.XNumberFormatter interface supported by the document.
ShadowFormat	Defines the shadow format using a com.sun.star.table.ShadowFormat structure: - Location – The shadow's location as a com.sun.star.table.ShadowLocation enum with valid values of NONE, TOP_LEFT, TOP_RIGHT, BOTTOM_LEFT, and BOTTOM_RIGHT. - ShadowWidth – The shadow's size as a Short Integer. - IsTransparent – If True, the shadow is transparent. - Color – The shadow's color as a Long Integer.
CellProtection	Defines the cell's protection as a com.sun.star.util.CellProtection structure: - IsLocked – If True, the cell is locked from modifications by the user. - IsFormulaHidden – If True, the formula is hidden from the user. - IsHidden – If True, the cell is hidden from the user. - IsPrintHidden – If True, the cell is hidden on printouts.
UserDefinedAttributes	This property is used to store additional attributes in a com.sun.star.container.XNameContainer interface.

In general, the process of setting cell attributes is very simple. The macro in Listing 412 demonstrates some of the properties in Table 166 by setting the text of cell B1 to “Hello”, centering the text, and then rotating the text 30 degrees counterclockwise.

Listing 412. Center “Hello” and rotate it 30 degrees.

```
Sub RotateCellText
    Dim oCell
    Dim oDoc

    oDoc = createNewCalcDoc()
    oCell = oDoc.Sheets(0).getCellByPosition(1, 0) 'Cell B1
    oCell.setString("Hello")
    oCell.HoriJustify = com.sun.star.table.CellHoriJustify.CENTER
    oCell.RotateAngle = 3000 '30 degrees
End Sub
```

	A	B
1		Hello

Figure 98. Cell value rotated 30 degrees.

Cell properties (see Table 166) are generic to most types of cells including text table cells. Cells in a spreadsheet contain other properties as well (see Table 167).

Table 167. Properties supported by the com.sun.star.sheet.SheetCell service.

Property	Description
Position	The position of the cell in the sheet (in 0.01 mm) as a com.sun.star.awt.Point structure. This is the absolute position in the entire sheet, not the position in the visible area. • X – The x-coordinate as a Long Integer. • Y – The y-coordinate as a Long Integer.
Size	The size of the cell (in 0.01 mm) as a com.sun.star.awt.Size structure: • Width – The width as a Long Integer. • Height – The height as a Long Integer.
FormulaLocal	Optional string containing the formula using localized function names.
FormulaResultType	Formula result type with values from the com.sun.star.sheet.FormulaResult constant group: • VALUE = 1 – The formula returns a Double precision floating-point number. • STRING = 2 – The formula returns a String value. • ERROR = 4 – The formula has an error of some sort.
ConditionalFormat	The conditional formatting settings for this cell. When a conditional format is changed, it must be reinserted into the property set.
ConditionalFormatLocal	Optional property duplicating ConditionalFormat except that formula names use local names.
Validation	The data validation settings for this cell as a com.sun.star.beans.XPropertySet.
ValidationLocal	Optional property duplicating Validation except that formula names use local names.

The macro in Listing 413 demonstrates how to obtain a cell's position and size in a sheet.

Listing 413. Print cell dimensions.

```
Sub CellDimensions
    Dim oCell
    Dim s As String
    oCell = ThisComponent.Sheets(0).getCellByPosition(1, 0) 'Cell B1
    s = s & CStr(oCell.Position.X \ 100) & " mm from the left" & CHR$(10)
    s = s & CStr(oCell.Position.Y \ 100) & " mm from the top" & CHR$(10)
    s = s & CStr(oCell.Size.Width \ 100) & " mm wide" & CHR$(10)
    s = s & CStr(oCell.Size.Height \ 100) & " mm tall" & CHR$(10)
    MsgBox s, 0, "Size And Postion Of Cell " & oCell.AbsoluteName
End Sub
```

15.2.4. Cell annotations

Each cell can contain one annotation that consists of simple unformatted text. The cell's getAnnotation() method returns an object that supports the com.sun.star.sheet.CellAnnotation service. Cell annotations support the XSimpleText interface used in Writer documents as well as the more specific methods shown in Table 168.

Table 168. *Methods implemented by the com.sun.star.sheet.CellAnnotation service.*

Method	Description
getParent()	Get the object (cell) containing this annotation.
setParent(oCell)	Set the cell that contains this annotation.
getPosition()	Get the com.sun.star.table.CellAddress of the cell containing this annotation (see Table 163).
getAuthor()	Get the user who last changed the annotation as a String.
getDate()	Get the date the annotation was last changed as a formatted String.
getIsVisible()	Returns True if the annotation is visible.
setIsVisible(boolean)	Set whether the annotation is visible.

The spreadsheet object supports the method `getAnnotations()`, which returns all of the annotations in the spreadsheet. You can obtain individual annotations by using indexed access or enumeration. The object returned by `getAnnotations()` also supports the methods `removeByIndex(n)` and `insertNew(CellAddress, String)`.

Listing 414. *Manipulate cell annotations.*

```
Sub manipulateAnnotations
    Dim doc As Object
    Dim aSheet As Object
    Dim aCell As Object
    Dim annotations As Object
    Dim myAnnotation As Object

    doc = ThisComponent
    aSheet = doc.sheets.getByIndex(0)

    'Insert a new annotation.
    aCell = aSheet.getCellByPosition(0, 0)
    annotations = aSheet.annotations
    annotations.insertNew(aCell.cellAddress, "Text content")

    'Modify an existing annotation.
    'Cannot use the cell's pseudo-property "annotation" which is read-only
    myAnnotation = getAnnotationByCell(aCell)
    myAnnotation.isVisible = true
    myAnnotation.annotationShape.fillColor = rgb(148, 0, 107)
    myAnnotation.annotationShape.charFontName = "Tahoma"
    myAnnotation.annotationShape.charHeight = 12

End Sub

Function getAnnotationByCell(aCell As Object) As Object
    Dim result As Object
    Dim annotations As Object
    Dim annotationIndex As Long
    Dim anAnnotation As Object
    Dim isFound As Boolean
```

```

    annotations = aCell.spreadSheet.annotations
    While ((annotationIndex < annotations.count) and (not isFound))
        anAnnotation = annotations.getByIndex(annotationIndex)
        isFound = ((anAnnotation.position.row = aCell.cellAddress.row) and
(anAnnotation.position.column = aCell.cellAddress.column))
        If (isFound) Then
            result = anAnnotation
        End If
        annotationIndex = annotationIndex + 1
    Wend
    getAnnotationByCell = result
End Function

```

15.3. Uninterpreted XML Attributes

OpenOffice.org stores documents as XML. When an OOO document is read, it is parsed by an XML parser. User-defined attributes are not interpreted by the parser, but they are simply read, stored, and then written. The intent is that a parser can store attributes that it cannot handle by itself on reading an XML file. When the file is stored again, the unknown attributes can be written back without loss.

Uninterpreted attributes also allow you to add your own properties that are saved with the document.

Table 169. Methods to manipulate user defined attributes.

Method	Description
getByName	Get the named attribute.
getElementNames	Get an array of strings containing the existing attribute names.
hasByName(name)	Determine if the named attribute exists.
hasElements	Determine if any attributes exist.
insertByName(name, attr)	Insert a new attribute.
removeByName(name)	Remove an existing attribute.
replaceByName(name, attr)	Replace an existing attribute.

Listing 415. To manipulate UserDefinedAttributes, use a copy and then assign it back.

```

Sub UserDefinedAttributeToCell
    Dim oCell          'The cell that will contain the attribute
    Dim oUserData      'A copy of the UserDefinedAttributes
    Dim oMyAttribute As New com.sun.star.xml.AttributeData
    Dim oAttribute
    Dim oDoc

    oDoc = createNewCalcDoc()
    REM First, get the cell that will contain the new attribute
    oCell = oDoc.Sheets(0).getCellByPosition(1, 0) 'Cell B1

    REM Now add data to the attribute.
    REM The Namespace is usually left blank, but set it if you want to.

```

```

REM omyAttribute.Namespace = "http://what/ever/you/want"

REM Notice that the type is CDATA and not something like "String"
omyAttribute.Type = "CDATA"
omyAttribute.Value = "Andrew Pitonyak"

REM Here is where things usually go wrong with a statement like
REM oCell.UserDefinedAttributes.InsertByName("Andy", oMyAttribute)
REM This fails every time. Instead, first make a copy and then
REM operate on the copy.
oUserData = oCell.UserDefinedAttributes
If NOT oUserData.HasByName("Andy") Then
    oUserData.InsertByName("Andy", oMyAttribute)
    oCell.UserDefinedAttributes = oUserData
End If
oAttribute = oCell.UserDefinedAttributes.GetByName("Andy")
Print oAttribute.Value
End Sub

```

Tip All services that support UserDefinedAttributes work the same way.

15.4. Sheet cell ranges

In Writer documents, continuous text can be grouped in a text range. In a spreadsheet, cells can be grouped in rectangular regions with a `SheetCellRange`. Grouping cells together allows multiple cells to be operated on at one time. The `SheetCellRange` service supports many of the same interfaces and properties as a `SheetCell`.

Tip Each sheet in a spreadsheet document is also a `SheetRange`.

Each cell has a cell address (see Table 163) that identifies its location in the spreadsheet document. Each cell range has an analogous structure that identifies its location in the spreadsheet (see Table 170).

Table 170. Properties of the `com.sun.star.table.CellRangeAddress` structure.

Property	Description
Sheet	Short Integer index of the sheet that contains the cell.
StartColumn	Long Integer index of the column where the left edge is located.
StartRow	Long Integer index of the row where the top edge is located.
EndColumn	Long Integer index of the column of the right edge of the range.
EndRow	Long Integer index of the row of the bottom edge of the range.

A single sheet cell range exists in a single sheet and contains one rectangular region. Multiple ranges are encapsulated by the `SheetCellRanges` service, which supports most of the same properties and services as a `SheetCellRange`. The similarity in functionality simplifies the learning curve and offers a lot of otherwise complicated functionality.

The SheetCellRanges service provides access to each range through the use of the XElementAccess interface and the XIndexAccess interface. The methods in Table 171 also provide access to the contained cell ranges.

Table 171. Methods defined by the *com.sun.star.table.XSheetCellRanges* interface.

Method	Description
getCells()	Return the collection of contained cells as an XEnumerationAccess.
getRangeAddressesAsString()	Return a string with the addresses of all contained ranges. The output format is similar to “Sheet1.B2:D6;Sheet3.C4:D5”.
getRangeAddresses()	Return an array of services of type CellRangeAddress (see Table 170).

15.4.1. Sheet cell range properties

Sheet cell ranges and sheet cells both support the properties Position, Size, ConditionalFormat, ConditionalFormatLocal, Validation, and ValidationLocal (see Table 167). The Position property for a sheet cell range provides the location of the upper-left cell — this is equivalent to obtaining the position property from the upper-left cell in the range. The Size property for a cell returns the size of a single cell, and the Size property for a sheet cell range provides the size for all of the cells contained in the range.

Validation settings

Sheet cells and sheet cell ranges are able to validate the data that they contain, to prevent invalid data from populating the cells. You can display a dialog when invalid data is entered (see Table 173). The service *com.sun.star.sheet.TableValidation* controls the validation process.

Before demonstrating how a validation is performed, I must introduce a few enumerated values, properties, and methods. Specify the validation that is performed by using the enumerated values shown in Table 172.

Table 172. Validation types defined by the *com.sun.star.sheet.ValidationType* enum.

Value	Description
<i>com.sun.star.sheet.ValidationType.ANY</i>	All content is valid; no conditions are used.
<i>com.sun.star.sheet.ValidationType.WHOLE</i>	Compare a whole (integer) number against the specified condition.
<i>com.sun.star.sheet.ValidationType.DECIMAL</i>	Compare any number against the specified condition.
<i>com.sun.star.sheet.ValidationType.DATE</i>	Compare a date value against the specified condition.
<i>com.sun.star.sheet.ValidationType.TIME</i>	Compare a time value against the specified condition.
<i>com.sun.star.sheet.ValidationType.TEXT_LEN</i>	Compare a string length against the specified condition.
<i>com.sun.star.sheet.ValidationType.LIST</i>	Only allow strings in the specified list.
<i>com.sun.star.sheet.ValidationType.CUSTOM</i>	Specify a formula that determines if the contents are valid.

When a cell containing invalid data is found, the `ValidationAlertStyle` enum specifies how the invalid data should be handled (see Table 173). Table 174 lists the supported conditional operators.

Table 173. Validation alerts defined by the *com.sun.star.sheet.ValidationAlertStyle* enum.

Value	Description
<code>com.sun.star.sheet.ValidationAlertStyle.STOP</code>	Display an error message and reject the change.
<code>com.sun.star.sheet.ValidationAlertStyle.WARNING</code>	Display a warning message and ask the user if the change will be accepted. The default answer is No.
<code>com.sun.star.sheet.ValidationAlertStyle.INFO</code>	Display an information message and ask the user if the change will be accepted. The default answer is Yes.
<code>com.sun.star.sheet.ValidationAlertStyle.MACRO</code>	Execute a specified macro.

Table 174. Conditions defined by the *com.sun.star.sheet.ConditionOperator* enum.

Value	Description
<code>com.sun.star.sheet.ConditionOperator.NONE</code>	No condition is specified.
<code>com.sun.star.sheet.ConditionOperator.EQUAL</code>	The value must be equal to the specified value.
<code>com.sun.star.sheet.ConditionOperator.NOT_EQUAL</code>	The value must not be equal to the specified value.
<code>com.sun.star.sheet.ConditionOperator.GREATER</code>	The value must be greater than the specified value.
<code>com.sun.star.sheet.ConditionOperator.GREATER_EQUAL</code>	The value must be greater than or equal to the specified value.
<code>com.sun.star.sheet.ConditionOperator.LESS</code>	The value must be less than the specified value.
<code>com.sun.star.sheet.ConditionOperator.LESS_EQUAL</code>	The value must be less than or equal to the specified value.
<code>com.sun.star.sheet.ConditionOperator.BETWEEN</code>	The value must be between the two specified values.
<code>com.sun.star.sheet.ConditionOperator.NOT_BETWEEN</code>	The value must be outside of the two specified values.
<code>com.sun.star.sheet.ConditionOperator.FORMULA</code>	The specified formula must have a non-zero result.

The validation object defines the type of validation and how to react to the validation using properties of the object (see Table 175).

Table 175. Properties supported by the *com.sun.star.sheet.TableValidation* service.

Property	Description
<code>Type</code>	The type of validation to perform, as shown in Table 172.
<code>ShowInputMessage</code>	If True, an input message appears when the cursor is in an invalid cell.
<code>InputTitle</code>	Title (String) of the dialog with the input message.
<code>InputMessage</code>	Text (String) of the input message.
<code>ShowErrorMessage</code>	If True, an error message appears when invalid data is entered.
<code>ErrorTitle</code>	Title (String) of the dialog showing the error message.
<code>ErrorMessage</code>	Text (String) of the error message.
<code>IgnoreBlankCells</code>	If True, blank cells are allowed.

Property	Description
ErrorAlertStyle	The action that is taken when an error occurs, as shown in Table 173.

Finally, the comparison that is performed is specified using methods implemented by the TableValidation service (see Table 176).

Table 176. Methods supported by the *com.sun.star.sheet.TableValidation* service.

Method	Description
getOperator()	Get the operator used in the condition, as shown in Table 174.
setOperator(condition)	Set the operator used in the condition, as shown in Table 174.
getFormula1()	Get the comparison value (String) used in the condition, or the first value if two are needed.
setFormula1(String)	Set the comparison value used in the condition, or the first value if two are required.
getFormula2()	Get the second value (String) if two are required.
setFormula2(String)	Set the second value (String) if two are required.
getSourcePosition()	Get the CellAddress that is used as a base for relative references in the formulas (see Table 163).
setSourcePosition(CellAddress)	Set the CellAddress that is used as a base for relative references in the formulas (see Table 163).

The macro in Listing 416 sets a validation range in the first sheet — that is, the sheet with a numerical index of 0. The cells from B2 to D6 are set to disallow any values that are not between 1 and 10. The macro itself is unexpectedly simple.

Listing 416. Set validation range in Calc.

```
Sub SetValidationRange
    Dim oRange          'Range that will accept the validation
    Dim oValidation      'The validation object
    Dim oDoc

    oDoc = createNewCalcDoc()

    REM Sheets support returning a cell range based on UI type names.
    oRange = oDoc.Sheets(0).getCellRangeByName("B2:D6")

    REM Obtain the Validation object
    oValidation = oRange.Validation

    REM Configure the validation to perform
    oValidation.Type = com.sun.star.sheet.ValidationType.DECIMAL
    oValidation.ErrorMessage = "Please enter a number between one and ten"
    oValidation.ShowErrorMessage = True
    oValidation.ErrorAlertStyle = com.sun.star.sheet.ValidationAlertStyle.STOP
    oValidation.setOperator(com.sun.star.sheet.ConditionOperator.BETWEEN)
```

```

oValidation.setFormula1(1.0)
oValidation.setFormula2(10.0)

REM Now set the validation
oRange.Validation = oValidation
Print "Try to enter a number in cells B2:D6 that is not between 1 and 10."
End Sub

```

The example in Listing 416 sets a Decimal validation between 1 and 10. This validation example does not generate an error for a value of 0 or 0.5. Try a few examples to see what else might fail.

Conditional formatting

Conditional formatting allows the cell style to change based on the cell's content. Sheet cells and sheet cell ranges both support the ConditionalFormat property, which in turn supports the XSheetConditionalEntries interface. You can access conditional formatting entries by using element access, indexed access, or the methods `addNew(properties())`, `clear()`, and `removeByIndex(index)`.

You can also apply multiple conditional formatting entries to the same cell. The first one that matches is applied. Each formatting entry is represented by an array of PropertyValue structures. Conditional formatting is very similar to validation in that they both use values and types from Table 172 through Table 176.

Validation is a relatively simple condition on a data element, enforcing type and format constraints. Conditional formatting supports validation and a more extended set of checks, including attributes or metadata defined on elements or collections of elements — the names are similar, but the actual operations, applications, and implications are much more elaborate. It is more difficult to explain than it is to demonstrate. The macro in Listing 417 sets a range of cells to use the “Heading1” style if the cell contains a negative number.

Listing 417. Set conditional style in Calc.

```

Sub SetConditionalStyle
    Dim oRange      'Cell range to use
    Dim oConFormat   'Conditional format object
    Dim oCondition(2) As New com.sun.star.beans.PropertyValue

    REM Sheets support returning a cell range based on UI type names.
    oRange = ThisComponent.Sheets(0).getCellRangeByName("B2:D6")
    oConFormat = oRange.ConditionalFormat

    oCondition(0).Name = "Operator"
    oCondition(0).Value = com.sun.star.sheet.ConditionOperator.LESS
    oCondition(1).Name = "Formula1"
    oCondition(1).Value = 0
    oCondition(2).Name = "StyleName"
    oCondition(2).Value = "Heading1"
    oConFormat.addNew(oCondition())
    oRange.ConditionalFormat = oConFormat
End Sub

```

15.4.2. Sheet cell range services

Sheet cells and sheet cell ranges have numerous services in common — for example, `CellProperties` (see Table 166), `CharacterProperties`, `CharacterPropertiesAsian`, `CharacterPropertiesComplex`, `ParagraphProperties`, and `SheetRangesQuery`.

Retrieving cells and ranges

A `SheetCellRange` support the `CellRange` service, which in turn supports `CellProperties` (see Table 166). The `CellRange` service offers extra functionality that is appropriate for cell ranges, but not for individual cells — for example, retrieving cells and cell ranges. When a cell range is retrieved using the methods in Table 177, the cells are indexed relative to the top-left corner of the range. When the range is an entire sheet, the location (0, 0) refers to the cell “A1.” If the range includes the cells “B2:D6”, however, the location (0,0) refers to cell “B2.” The macros in Listing 410 through Listing 413 all use the method `getCellByPosition()`.

Table 177. *Methods supported by the `com.sun.star.table.XCellRange` interface.*

Method	Description
<code>getCellByPosition(left, top)</code>	Get a cell within the range.
<code>getCellRangeByPosition(left, top, right, bottom)</code>	Get a cell range within the range.
<code>getCellRangeByName(name)</code>	Get a cell range within the range based on its name. The string directly references cells using the standard formats — such as “B2:D5” or “\$B\$2” — or defined cell range names.

Tip

The methods `getCellByPosition()`, `getCellRangeByPosition()`, and `getCellRangeByName()` cannot return a value that is not in the range (see Listing 445).

Querying cells

Sheet cell ranges and sheet cells both support the ability to find cells with specific properties. This ability provides a mechanism for finding all cells that are referenced by the current cell’s formula and the ability to see which cells reference the current cell. While performing a query based on the cell’s content, the `CellFlags` in Table 178 specify the content type to search.

Table 178. *Values in the `com.sun.star.sheet.CellFlags` constant group.*

Value	Flag	Description
1	<code>com.sun.star.sheet.CellFlags.VALUE</code>	Select numbers not formatted as dates or times.
2	<code>com.sun.star.sheet.CellFlags.DATETIME</code>	Select numbers formatted as dates or times.
4	<code>com.sun.star.sheet.CellFlags.STRING</code>	Select strings.
8	<code>com.sun.star.sheet.CellFlags.ANNOTATION</code>	Select cell annotations.
16	<code>com.sun.star.sheet.CellFlags.FORMULA</code>	Select formulas.
32	<code>com.sun.star.sheet.CellFlags.HARDATTR</code>	Select explicit formatting — not styles.
64	<code>com.sun.star.sheet.CellFlags.STYLES</code>	Select cell styles.
128	<code>com.sun.star.sheet.CellFlags.OBJECTS</code>	Select drawing objects such as buttons and graphics.

Value	Flag	Description
256	com.sun.star.sheet.CellFlags.EDITATTR	Select formatting within the cell's content.

Each of the methods used to query a cell range (see Table 179) also returns a cell range (see Table 171).

Table 179. Methods to query a cell range.

Method	Description
queryDependents(boolean)	Return all cells that reference cells in this range. If True, search is recursive.
queryPrecedents(boolean)	Return all cells referenced by cells in this range. If True, search is recursive.
queryVisibleCells()	Return all visible cells.
queryEmptyCells()	Return all empty cells.
queryContentCells(CellFlags)	Return all cells with the specified content types (see Table 178).
queryFormulaCells(FormulaResult)	Return all cells containing a formula with the specified result type (see FormulaResultType in Table 167).
queryColumnDifferences(CellAddress)	Return all cells that differ from the comparison cell in the specified cell's row (see Table 163).
queryRowDifferences(CellAddress)	Return all cells that differ from the comparison cell in the specified cell's column (see Table 163).
queryIntersection(CellRangeAddress)	Return the range of cells that intersect the specified range (see Table 170).

Finding non-empty cells in a range

Use the method `queryContentCells(CellFlags)` to obtain a list of all cells in a range that are not empty. The `CellFlags` argument is set to return all cells that contain a value, string, formula, or date/time. The interesting thing about Listing 418 is not that it uses a query to find the nonempty cells, but rather that it demonstrates how to extract the cells from the query and enumerate all of the returned cells. In other words, the macro in Listing 418 demonstrates how to visit all of the nonempty cells in a specific range.

Listing 418. Find cells that are not empty in a range.

```
Function NonEmptyCellsInRange(oRange, sep$) As String
    Dim oCell          'The cell to use!
    Dim oRanges         'Ranges returned after querying for the cells
    Dim oAddrs()        'Array of CellRangeAddress
    Dim oAddr           'One CellRangeAddress
    Dim oSheet          'Sheet that contains the cell range
    Dim i As Long       'General index variable
    Dim nRow As Long    'Row number
    Dim nCol As Long    'Column number
    Dim s As String

    REM First, find the cells that are NOT empty in this range!
    REM I consider a cell to be not empty if it has a value,
    REM date/time, string, or formula.
```

```

oRanges = oRange.queryContentCells(_
    com.sun.star.sheet.CellFlags.VALUE OR _
    com.sun.star.sheet.CellFlags.DATETIME OR _
    com.sun.star.sheet.CellFlags.STRING OR _
    com.sun.star.sheet.CellFlags.FORMULA)

oAddrs() = oRanges.getRangeAddresses()
For i = 0 To UBound(oAddrs())

    REM Get a specific address range
    oAddr = oAddrs(i)

    For nRow = oAddr.StartRow To oAddr.EndRow
        For nCol = oAddr.StartColumn To oAddr.EndColumn
            oCell = oRange.Spreadsheet.getCellByPosition(nCol, nRow)
            s = s & oCell.AbsoluteName & sep$
        Next
    Next

Next
NonEmptyCellsInRange = s
End Function

```

Using complex queries

Although the query methods are easy to use, some of them are conceptually quite complicated. All of the query methods return a `SheetCellRanges` object. The macro in Listing 419 demonstrates how to find the cells referenced by a formula using the `queryPrecedents()` method, how to find the cells referencing a particular cell using the `queryDependents()` method, as well as how to find row and column differences using the `queryRowDifferences()` and `queryColumnDifferences()` methods.

The macro in Listing 419 configures the first sheet with values and formulas, and then the macro performs some queries. The code that performs the demonstrated queries is simple and short. The code that creates the data that is queried, however, is more complicated and instructive. More specifically, the macro in Listing 419 demonstrates how to clear a range of cells by using the `clearContents()` method. The dialog produced by Listing 419 is shown in Figure 99.

Listing 419. Query a range of cells to find references, dependencies, and differences.

```

Sub QueryCellRange
    Dim oCell           'Holds a cell temporarily
    Dim oCellAddress    'Holds a cell's address
    Dim oRange          'The primary range
    Dim oSheet          'The first sheet
    Dim i As Integer    'Temporary index variable
    Dim s As String     'Temporary String
    Dim oDoc             : oDoc = createNewCalcDoc()

    PopulateSheetForQuery(oDoc)

    REM Sheets support returning a cell range based on UI type names.
    oSheet = oDoc.Sheets(0)

```

```

REM Get the range I want to use!
oRange = oSheet.getCellRangeByName("A1:F8")
s = InputBox("Query which cell?", "Enter Cell Address", "C7")

REM This includes cells "C2:C7". Notice that it includes the cell itself
oCell = oSheet.getCellByPosition(2, 6)
s = s & "=SUM(C2:C6) directly references " & _
    oCell.queryPrecedents(False).getRangeAddressesAsString() & CHR$(10)

REM This includes cells "B2:B6;C2:C7"
s = s & "=SUM(C2:C6) Including indirect references " & _
    oCell.queryPrecedents(True).getRangeAddressesAsString() & CHR$(10)

REM Find direct and indirect references
oCell = oSheet.getCellByPosition(1, 2) 'B3
s = s & "Cells that reference B3 " & _
    oCell.queryDependents(True).getRangeAddressesAsString() & CHR$(10)

oCellAddress = oCell.CellAddress
s = s & "Column differences for B3 " & _
    oRange.queryColumnDifferences(oCellAddress).getRangeAddressesAsString()
s = s & CHR$(10)

s = s & "Row differences for B3 " & _
    oRange.queryRowDifferences(oCellAddress).getRangeAddressesAsString()
s = s & CHR$(10)

MsgBox s, 0, "Manipulating A Range"
End Sub

```

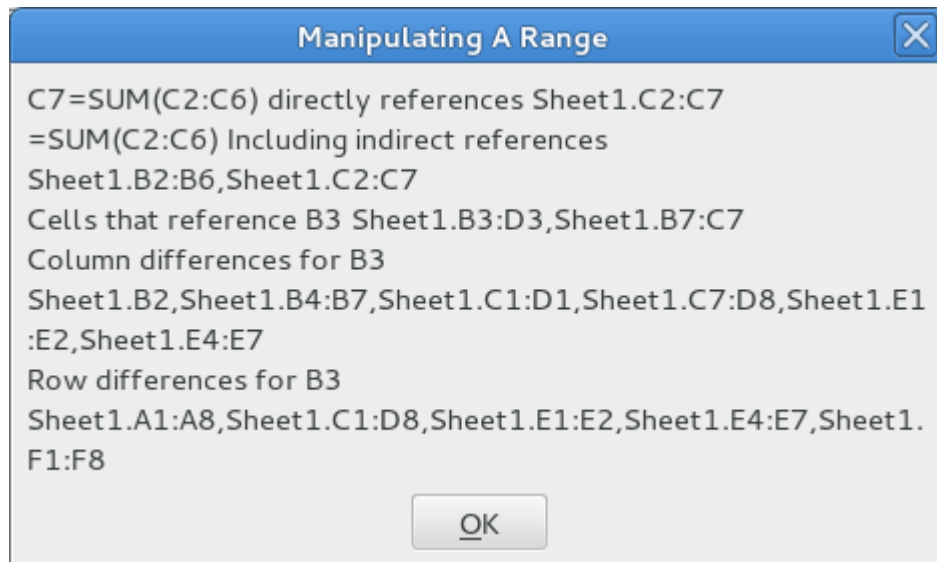


Figure 99. Query a range of cells to find references, dependencies, and similarities.

Table 180 shows the formulas and values created by the macro in Listing 419, assists in understanding the output shown in Figure 99.

Table 180. *The formulas and values set by Listing 419.*

	B	C	D	E
1	2	=B1-1		
2	1	=B2+1	=C2-1	
3	2	=B3+1	=C3-1	2
4	3	=B4+1	=C4-1	
5	4	=B5+1	=C5-1	
6	5	=B6+1	=C6-1	
7	=SUM(B2:B5)	=SUM(C2:C6)		
8	2			2

Query Precedents and Dependents

The first line in Figure 99 shows the result of queryPrecedents(False) on a range that contains only cell C7. As shown in Table 180, cell C7 directly references the cells C2:C6 and itself. By calling queryPrecedents(True) — the second line in Figure 99 — the cells B2:B6 are added because the cells in column C reference the cells in column B.

The method queryDependents(True) provides the third line in Figure 99, which displays all of the cells that reference B3, directly or indirectly. Table 180 shows that B3, B7, and C3 directly reference cell B3, and that cells C7 and D3 indirectly reference cell B3.

Query Column Differences

The fourth line in Figure 99 lists the “column differences” based on cell B3. While calculating column differences, only the row matters. In other words, the same result is returned if cell A3, C3, D3, or E3 is used instead of cell B3. Figure 100 shows the sheet; the cells that are considered different are marked with a black background. Figure 100 helps to illustrate what is, and is not, considered a difference. By choosing cell B3, all of the cells in each column (of the range) are compared to the cell in row 3 (in the same column). The first column, A, has no cells selected because they are all empty; therefore, all the cells in the column have the same value. In column B, cell B3 contains the constant value of 2. Cells B1 and B8 also contain the constant value 2, so these cells are not considered different. Column C is very interesting in that it does not contain constants, but rather it contains formulas that are similar. The cells C2, C4, C5, and C6 are similar to cell C3. The formulas are effectively the same (see Table 180) in that they all add 1 to the cell to the left. The formula in C1 is not similar to C3 so it is included in the difference list. Column D is similar to column C and column E is similar to column B. I leave it as an exercise for the reader to explain the row differences as shown in Figure 100.

	A	B	C	D	E	F
1		2	1			
2		1	2	1		
3		2	3	2	2	
4		3	4	3		
5		4	5	4		
6		5	6	5		
7		10	20			
8		2			2	
9						

Figure 100. Output from Listing 419 with “column differences” highlighted.

15.4.3. Searching and replacing

The thing that I find most interesting about searching in a spreadsheet document is that searching is not supported by the document object. Cell object and cell range objects support searching, however. Each sheet is also a sheet range, so it’s possible to search an entire sheet. It is not possible, therefore, to search an entire document at one time; you must search each sheet separately. Listing 420 demonstrates searching and replacing text in a single sheet.

Listing 420. Simple Calc search.

```
Sub SearchSheet
    Dim oSheet      'Sheet in which to replace
    Dim oReplace    'Replace descriptor
    Dim nCount      'Number of times replaced

    oSheet = ThisComponent.Sheets(3)
    oReplace = oSheet.createReplaceDescriptor()
    oReplace.setSearchString("Xyzzy")
    oReplace.setReplaceString("Something Else")
    oReplace.SearchWords = False
    nCount = oSheet.replaceAll(oReplace)
    MsgBox "Replaced " & nCount
End Sub
```

Searching a sheet in a Calc document is almost identical to searching for text in a Writer document. In a Calc document, setting SearchWords to True forces the entire cell to contain the search text. In a Writer document, setting SearchWords to True forces the text to not be part of another word.

Tip In a Calc document, setting SearchWords to True forces the entire cell to contain the search text.

15.4.4. Merging cells

Use the merge method to merge and unmerge a range of cells. After merging the range B2:D7, cell B2 appears in the area formerly used by the entire range. What is not shown in Figure 101 is that the cells that aren’t visible still exist and are accessible; they are simply not displayed. Use the getIsMerged() method to determine if all of the cells in a range are merged.

Listing 421. Merge a range of cells.

```
Sub MergeExperiment
    Dim oCell          'Holds a cell temporarily
    Dim oRange         'The primary range
    Dim oSheet         'The first sheet

    REM Merge a range of cells
    oSheet = ThisComponent.Sheets(0)
    oRange = oSheet.getCellRangeByName("B2:D7")
    oRange.merge(True)

    REM Now obtain a cell that was merged
    REM and I can do it!
    oCell = oSheet.getCellByPosition(2, 3)    'C4
    Print oCell.getValue()
End Sub
```

	A	B	C	D	E
1		2	1		
2					
3					2
4					
5					
6					
7				1	
8		2			2

Figure 101. Merging cells causes the top-left cell to use the entire merged area.

15.4.5. Retrieving, inserting, and deleting columns and rows

Use the methods `getColumns()` and `getRows()` to retrieve columns and rows covered by cells and cell ranges. After retrieving the columns for a range or a cell, you can retrieve the individual columns by using the interfaces `XElementAccess` or `XindexAccess`. You can use two additional methods — `insertByIndex(index, count)` and `removeByIndex(index, count)` — to insert and remove columns. After obtaining a single column, you can get the column's name with `getName()`, set cell properties for the entire column, and extract cells by using the cell range methods in Table 177.

All of the manipulations mentioned for columns — except for `getName()` — also apply to the rows obtained by using the `getRows()` method. The difference lies in the properties supported by the individual columns and rows (see Table 181).

Table 181. Individual row and column properties.

Type	Property	Description
Column	Width	Width of the column (in 0.01 mm) as a Long Integer.
Row	Height	Height of the row (in 0.01 mm) as a Long Integer.
Column	OptimalWidth	If True, the column always keeps its optimal width.
Row	OptimalHeight	If True, the row always keeps its optimal height.
Both	IsVisible	If True, the row or column is visible.
Both	IsStartOfNewPage	If True, a horizontal (vertical) page break is attached to the column (row).

The macro in Listing 422 obtains the range “B6:C9” and then traverses all of the nonempty cells. The hard part is done in the routine NonEmptyCellsInRange() as shown in Listing 418. Listing 422, however, demonstrates how to extract the individual rows from a range. Typically, when writing a macro, you know where the data lies so you simply write code to traverse the data directly. Figure 102 shows the output from Listing 422 when the macro is run using the data shown in Figure 100.

Listing 422. Enumerate rows in a range and find nonempty cells.

```
Sub TraverseRows
    Dim oRange          'The primary range
    Dim oSheet          'The fourth sheet
    Dim oRows           'Rows object
    Dim oRow            'A single row
    Dim oRowEnum        'Enumerator for the rows
    Dim s As String     'General String Variable

    oSheet = ThisComponent.Sheets(3)
    oRange = oSheet.getCellRangeByName("B6:C9")

    REM I now want to find ALL of the cells that are NOT empty in the
    REM rows that are related to the range. Notice that I do not want to
    REM limit myself to cells in the range, but that I am interested in the
    REM rows.

    oRows = oRange.getRows()
    REM Sure, I could access things by index, but you probably expected that!
    oRowEnum = oRows.createEnumeration()
    Do While oRowEnum.hasMoreElements()
        oRow = oRowEnum.nextElement()
        s = s & NonEmptyCellsInRange(oRow, " ") & CHR$(10)
    Loop
    MsgBox s, 0, "Non-Empty Cells In Rows"
End Sub
```

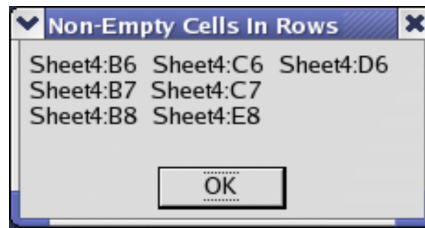


Figure 102. The nonempty cells are displayed from rows 6 through 8.

15.4.6. Retrieving and setting data as an array

You can quickly and easily obtain all data from a range as an array of arrays by using the `getDataArray()` method. The data in each cell is returned as either a number or a string. You can also set the data for a range by using the `setDataArray()` method; just be certain that the array dimensions match the range dimensions (see Listing 423).

Listing 423. *Get and set data in a Calc sheet.*

```
Sub GetAndSetData
    Dim oRange          'The primary range
    Dim oSheet          'The fourth sheet
    Dim oAllData        'Array containing the data
    Dim s As String     'General string variable
    Dim i As Integer    'General index variable

    oSheet = ThisComponent.Sheets(3)
    oRange = oSheet.getCellRangeByName("B6:E8")

    REM Get the data contained in the range!
    REM Data from empty cells is included.
    oAllData = oRange.getDataArray()
    For i = 0 To UBound(oAllData)
        REM oAllData(i) is an array, so simply join the data together
        REM for a quick printing!
        s = s & "(" & Join(oAllData(i), ", ") & ")" & CHR$(10)
    Next
    MsgBox s, 0, "Data In Range"

    REM Now quickly set some data.
    oRange = oSheet.getCellRangeByName("F1:G2")
    oRange.setDataArray(Array(Array(1, "One"), Array(2, "Two")))
End Sub
```

Tip

As of Calc 3.0, the maximum number of columns is 1024 (column is AMJ), the maximum number of rows is 65,536, and the maximum number cells in a sheet is 67,108,864. Laurent Godard performed tests and increased the number for testing purposes: see

<http://blogs.nuxeo.com/dev/2006/08/the-spreadsheet-next.html>

http://wiki.services.openoffice.org/wiki/Calc/hacks/number_of_rows

Sheet cell ranges also provide the ability to get and set formulas in bulk using arrays. Use the methods `getFormulaArray()` and `setFormulaArray()` to get and set the formulas in a range as an array — these two methods can be a real time saver.

If you inspect a sheet range, you will notice that a sheet supports `getData()` and `getDataArray()`. Both methods return an array of arrays. The method `getDataArray` returns numbers and strings. The `getData` method defined by the `XChartData` interface, however, returns numbers and ignores strings.

15.4.7. Computing functions on a range

It's possible to compute a value based on a specified range (see Listing 418 and Listing 422), but it is tedious. You can use the `getDataArray()` method (see Listing 423) to quickly obtain all of the data and simply process the data in the nested arrays. To easily apply a simple function to a range, use the object method `computeFunction(GeneralFunction)`. Table 182 lists the functions supported by the `GeneralFunction` enumeration.

Table 182. The `com.sun.star.sheet.GeneralFunction` enumeration.

Value	Description
<code>com.sun.star.sheet.GeneralFunction.NONE</code>	Nothing is calculated.
<code>com.sun.star.sheet.GeneralFunction.AUTO</code>	Use SUM if all values are numerical; otherwise use COUNT.
<code>com.sun.star.sheet.GeneralFunction.SUM</code>	Sum (add) all of the numerical values.
<code>com.sun.star.sheet.GeneralFunction.COUNT</code>	Count all of the values.
<code>com.sun.star.sheet.GeneralFunction.AVERAGE</code>	Average all of the numerical values.
<code>com.sun.star.sheet.GeneralFunction.MAX</code>	Maximum numerical value.
<code>com.sun.star.sheet.GeneralFunction.MIN</code>	Minimum numerical value.
<code>com.sun.star.sheet.GeneralFunction.PRODUCT</code>	Product (multiplication) of all the numerical values.
<code>com.sun.star.sheet.GeneralFunction.COUNTNUMS</code>	Count the numerical values.
<code>com.sun.star.sheet.GeneralFunction.STDEV</code>	Standard deviation based on a sample.
<code>com.sun.star.sheet.GeneralFunction.STDEVP</code>	Standard deviation based on the entire population.
<code>com.sun.star.sheet.GeneralFunction.VAR</code>	Variance based on a sample.
<code>com.sun.star.sheet.GeneralFunction.VARP</code>	Variance based on the entire population.

The macro in Listing 424 demonstrates the use of the compute function. Using the sheet shown in Figure 100 and the macro in Listing 424, there are seven nonempty cells in the range A5:C9.

Listing 424. Compute a function on a range.

```
Sub UseCompute
    Dim oRange          'The primary range
    Dim oSheet          'The fourth sheet
    Dim d As Double     'Return value

    oSheet = ThisComponent.Sheets(3)
    oRange = oSheet.getCellRangeByName("A5:C9")
```

```

d = oRange.computeFunction(com.sun.star.sheet.GeneralFunction.COUNT)
MsgBox "Non-Empty cells in A5:C9 = " & d, 0, "ComputFuntion()"
End Sub

```

15.4.8. Clearing cells and cell ranges

In a Calc document, click on a cell and press the Delete key. A dialog opens with a list of things that can be deleted. The ability to delete any combination of things from different types of content or formatting is amazingly powerful and flexible. The types of things that can be deleted are encapsulated by the CellFlags shown in Table 178. CellFlags may be combined using the OR operator and passed to the clearContents(CellFlags) method (see Listing 419).

Tip The clearContents() method is supported by cells, cell ranges, and even rows and columns.

15.4.9. Automatic data fill

Listing 419 painfully fills consecutive cells with data. Sheet cell ranges provide a better method for automatically filling a range with data by using the fillAuto(FillDirection, nCount) method. The FillDirection enumerated values shown in Table 183 control how the data is propagated.

Table 183. The com.sun.star.sheet.FillDirection enumeration.

Value	Description
com.sun.star.sheet.FillDirection.TO_BOTTOM	Rows are filled from top to bottom.
com.sun.star.sheet.FillDirection.TO_RIGHT	Columns are filled from left to right.
com.sun.star.sheet.FillDirection.TO_TOP	Rows are filled from bottom to top.
com.sun.star.sheet.FillDirection.TO_LEFT	Columns are filled from right to left.

Use the method fillAuto(FillDirection, nCount) to automatically fill an area. First, select the range that you want to fill. Second, set the initial value that will be incremented by the fillAuto() method. The location of the initial values depend on the direction that will be filled. For example, if using TO_LEFT, the rightmost cells in the range must contain an initial value so that they can be filled to the left.

While filling new values, the fillAuto() method increments the number by 1 while moving to the right or bottom, and decrements the number by 1 while moving to the left or top. If the number is formatted as a time or date, incrementing by 1 adds one day.

Tip When a time is incremented using fillAuto(), it is incremented by one day. If the cell is formatted to show only the time, it will appear as though the value does not change — although it has. This should be obvious if you think about it.

The fillAuto() method uses the last argument, nCount, to determine how many cells to move before entering a new value. A value of 1, therefore, fills every cell as the cursor moves. While filling new cells, the fillAuto() method won't move outside the range used to call fillAuto(). The

code snippet in Listing 425 assumes that Sheet 3 contains numerical values in the cell range E11:E20.

Listing 425. Fill values from E11:N20 using fillAuto.

```
oSheet = ThisComponent.Sheets(3)
oRange = oSheet.getCellRangeByName("E11:N20")
oRange.fillAuto(com.sun.star.sheet.FillDirection.TO_RIGHT, 1)
```

OOo also supports more complicated fill methods. The FillMode enumeration (see Table 184) directs the special functionality provided by the method fillSeries(). The fillAuto() method always uses the LINEAR mode to increment the value by one, whereas the fillSeries() method allows for any fill mode.

Table 184. The com.sun.star.sheet.FillMode enumeration.

Value	Description
com.sun.star.sheet.FillMode.SIMPLE	All of the values are the same (constant series).
com.sun.star.sheet.FillMode.LINEAR	The values change by a constant increment (arithmetic series).
com.sun.star.sheet.FillMode.GROWTH	The values change by a constant multiple (geometric series).
com.sun.star.sheet.FillMode.DATE	An arithmetic series for date values. This causes all numbers to be treated as dates, regardless of formatting.
com.sun.star.sheet.FillMode.AUTO	The cells are filled from a user-defined list.

The fillSeries() method recognizes dates and times based on the numerical format used to display them. Using the FillMode DATE forces all numbers to be recognized as dates rather than just numbers that are formatted as dates. When a date is filled, the day, month, or year can be changed as specified by the FillDateMode enumerated values (see Table 185).

Table 185. The com.sun.star.sheet.FillDateMode enumeration.

Value	Description
com.sun.star.sheet.FillDateMode.FILL_DATE_DAY	Increment the day by 1.
com.sun.star.sheet.FillDateMode.FILL_DATE_WEEKDAY	Increment the day by 1 but skip Saturday and Sunday.
com.sun.star.sheet.FillDateMode.FILL_DATE_MONTH	Increment the month (the day is unchanged).
com.sun.star.sheet.FillDateMode.FILL_DATE_YEAR	Increment the year (the day and month are unchanged).

The method fillSeries(FillDirection, FillMode, FillDateMode, nStep, nEndValue) provides the greatest flexibility for filling values. The nStep value indicates how the value is modified from cell to cell. The final argument specifies a final value not to exceed during the fill. The fillSeries() method will not modify a value outside of the range and stops adding values when passing the end value. While specifying an end value, remember that dates expressed as a regular number are rather large; 40000 refers to July 6, 2009. Listing 426 uses the fillSeries() method to fill a range of dates.

Tip

The fillAuto() method increments or decrements the value depending on the direction in which the values are filled. The fillSeries() method, however, always uses the nStep value regardless of the direction.

Listing 426. Fill values from E11:N20 using fillSeries().

```
oSheet = ThisComponent.Sheets(3)
oRange = oSheet.getCellRangeByName("E11:N20")
oRange.fillSeries(com.sun.star.sheet.FillDirection.TO_LEFT, _
    com.sun.star.sheet.FillMode.LINEAR, _
    com.sun.star.sheet.FillDateMode.FILL_DATE_DAY, 2, 40000)
```

If text containing a number is found in an initial cell, the copied value copies the text and increments the rightmost number in the text. For example, I entered the text “Text 1” and the filled text contained “Text 3”, “Text 5”, and so on.

15.4.10. Array formulas

The simplest usage of an array formula that I have seen involves placing an array formula in one cell and using the formula in multiple cells. Now that I have provided just enough detail to cause confusion, consider the simple example shown in Table 186.

Table 186. A simple formula in column I.

	F	G	H	I	J	K
3		1	3	=G3+H3		
4		2	4	=G4+H4		
5		3	5	=G5+H5		
6		4	6	=G6+H6		
7		5	7	=G7+H7		
8		6	8	=G8+H8		

Column I contains the formula to add column G to column H. This can be done using an array formula by entering one formula in one cell. To enter an array formula into column J that mimics the formula in column I, first place the cursor in cell J3. Enter the formula “=G3:G8+H3:H8” and then press Ctrl-Shift-Enter. The cells J3 through J8 now contain the single formula “{=G3:G8+H3:H8}” and the values in column J should match those in column I. Column I contains six formulas that are not directly related to each other, but the cells in column J use only one formula. The macro in Listing 427 sets a sheet to look like Table 186 and then sets column J to contain an array formula that calculates the same values as column I.

Listing 427. Demonstrate an array formula.

```
Sub ArrayFormula
    Dim oRange          'The primary range
    Dim oSheet          'The fourth sheet
    Dim oCell           'Holds a cell temporarily
    Dim i As Integer    'General Index Variable
    Dim oDoc            'Reference newly created calc document.
```

```

oDoc = StarDesktop.loadComponentFromURL("private:factory/scalc", "_default", 0, Array())
oSheet = oDoc.Sheets(0)

REM Set the two top cells in G3:H8
oCell = oSheet.getCellByPosition(6, 2)      'Cell G3
oCell.SetValue(1)

oCell = oSheet.getCellByPosition(7, 2)      'Cell H3
oCell.SetValue(3)

REM Fill the values down!
oRange = oSheet.getCellRangeByName("G3:H8")
oRange.fillAuto(com.sun.star.sheet.FillDirection.TO_BOTTOM, 1)

REM This demonstrates setting each cell individually.
For i = 3 To 8
    oCell = oSheet.getCellByPosition(8, i-1)      'Cell I3 - I8
    oCell.setFormula("=G" & i & "+H" & i)
Next

REM Setting a single array formula is much easier in this case.
oRange = oSheet.getCellRangeByName("J3:J8")
oRange.setArrayFormula("=G3:G8+H3:H8")

REM Add some headings!
oRange = oSheet.getCellRangeByName("G2:J2")
oRange.setDataArray(Array(Array("G", "H", "Formula", "Array Formula")))
End Sub

```

15.4.11. Computing multiple functions on a range

OOo supports using a series of single variable formulas against a series of values. A typical example involves a single column (or row) of numbers with adjacent columns (or rows) containing formulas using the numbers. The enumerated values in Table 187 specify if rows or columns are used.

Table 187. The *com.sun.star.sheet.TableOperationMode* enumeration

Value	Description
com.sun.star.sheet.TableOperationMode.COLUMN	Apply the operation down the columns.
com.sun.star.sheet.TableOperationMode.ROW	Apply the operation across the rows.
com.sun.star.sheet.TableOperationMode.BOTH	Apply the operation to both rows and columns.

The `setTableOperation()` method provides the ability to quickly apply multiple single variable functions to the same set of data producing a table of values. The method `setTableOperation()` accepts four arguments as follows:

1. `CellRangeAddress` — Cell range address that contains the functions to apply.
2. `TableOperationMode` — Identifies if the data is in rows or columns (see Table 187).

3. CellAddress — Cell address that is used if using columns (row mode or both).
4. CellAddress — Cell address that is used if using rows (column mode or both).

The macro in Listing 428 generates a column of numbers from 0 to 6.3, the functions Sin() and Cos() are then applied to the columns.

Listing 428. Use `setTableOperation` to quickly set an operation on a column.

```
Sub MultipleOpsColumns
    Dim oRange          'The primary range
    Dim oSheet          'The fourth sheet
    Dim oCell           'Holds a cell temporarily
    Dim oBlockAddress   'Address of the block to fill
    Dim oCellAddress    'Row or column cell
    Dim oDoc            'Reference newly created calc document.

    oDoc = StarDesktop.loadComponentFromURL("private:factory/scalc", "_default", 0, Array())
    oSheet = oDoc.Sheets(0)

    REM Set the top-most value!
    oCell = oSheet.getCellByPosition(0, 9)      'Cell A10
    oCell.setValue(0)

    REM Fill the values down! for 0 to 6.4
    oRange = oSheet.getCellRangeByName("A10:A74")
    oRange.fillSeries(com.sun.star.sheet.FillDirection.TO_BOTTOM, _
        com.sun.star.sheet.FillMode.LINEAR, _
        com.sun.star.sheet.FillDateMode.FILL_DATE_DAY, 0.1, 6.4)

    REM Now set the Sin() and Cos() Header values
    oCell = oSheet.getCellByPosition(1, 8)      'Cell B9
    oCell.setString("Sin()")
    oCell = oSheet.getCellByPosition(2, 8)      'Cell C9
    oCell.setString("Cos()")

    REM Now set the Sin() and Cos() formulas
    oCell = oSheet.getCellByPosition(1, 9)      'Cell B10
    oCell.setFormula("=Sin(A10)")
    oCell = oSheet.getCellByPosition(2, 9)      'Cell C10
    oCell.setFormula("=Cos(A10)")

    REM Obtain the entire block on which to operate.
    oRange = oSheet.getCellRangeByName("A11:C74")

    REM Obtain the address that contains the formulas to copy.
    oBlockAddress = oSheet.getCellRangeByName("B10:C10").getRangeAddress()

    REM The address of the cell that contains the column of data.
    oCellAddress = oSheet.getCellByPosition(0, 9).getCellAddress()

    REM I really only need the column value because the row value is not used.
    oRange.setTableOperation(oBlockAddress, _
        com.sun.star.sheet.TableOperationMode.COLUMN, _
```

```

        oCellAddress, oCellAddress)
End Sub

```

If the table operation mode is set to BOTH rather than just ROW or COLUMN, then a single function is applied to two variables. The macro in Listing 429 creates the multiplication table that I memorized in third grade.

Listing 429. Use a table operation to create a multiplication table

```

Sub createMultiplicationTableWrapper()
    createMultiplicationTable()
End Sub

Function createMultiplicationTable()
    Dim oRowCell      'The row cell
    Dim oColCell      'The column cell
    Dim oRange        'The primary range
    Dim oSheet        'The fourth sheet
    Dim oCell         'Holds a cell temporarily
    Dim oBlockAddress 'Address of the block to fill
    Dim oCellAddress  'Row or column cell
    Dim oDoc          'Reference newly created calc document.

    oDoc = StarDesktop.loadComponentFromURL("private:factory/scalc", "_default", 0, Array())
    oSheet = oDoc.Sheets(0)

    REM Set the row of constant values
    oRowCell = oSheet.getCellByPosition(1, 9)    'Cell B10
    oRowCell.setValue(1)
    oRange = oSheet.getCellRangeByName("B10:K10")
    oRange.fillAuto(com.sun.star.sheet.FillDirection.TO_RIGHT, 1)
    oRange.CharWeight = com.sun.star.awt.FontWeight.BOLD
    oRange.CharHeight = 14

    REM Set the column of constant values
    oColCell = oSheet.getCellByPosition(0, 10)    'Cell A11
    oColCell.setValue(1)
    oRange = oSheet.getCellRangeByName("A11:A20")
    oRange.fillAuto(com.sun.star.sheet.FillDirection.TO_BOTTOM, 1)
    oRange.CharWeight = com.sun.star.awt.FontWeight.BOLD
    oRange.CharHeight = 14

    REM Set the formula that will be used. It references the first values!
    oCell = oSheet.getCellByPosition(0, 9)        'Cell A10
    oCell.setFormula("=A11*B10")

    REM Get the range of the cells
    oRange = oSheet.getCellRangeByName("A10:K20")

    REM Fill the multiplication tables for the values 1x1 through 10x10
    oRange.setTableOperation(oRange.getRangeAddress(), _
        com.sun.star.sheet.TableOperationMode.BOTH, _
        oColCell.getCellAddress(), _
        oRowCell.getCellAddress())

```

```

createMultiplicationTable() = oDoc
End Function

```

15.4.12. Cells with the same formatting

Sheet cell ranges provide two methods for obtaining groups of cells that contain the same format. The `getCellFormatRanges()` method returns an object that supports both index and enumeration access. The equally formatted cells are split into multiple rectangular ranges. The enumerated ranges are returned as a `SheetCellRange` object.

The object method `getUniqueCellFormatRanges()` is very similar to the method `getCellFormatRanges()` except that the returned values are objects of type `SheetCellRanges`. The primary difference is that all of the similarly formatted objects are returned in one container. The macro in Listing 430 displays the similarly formatted ranges using the two different methods.

Listing 430. *Use two different methods to display groups of cells that contain the same format..*

```

Sub DisplaySimilarRanges
    Dim oSheetCellRange 'An individual sheet cell range
    Dim oSheetCellRanges 'Sheet cell ranges
    Dim oAddr           'An address object from the sheet cell range
    Dim s$              'Utility string variable
    Dim x               'The returned range objects are stored here
    Dim i%              'Utility index variable
    Dim oSheet
    Dim oDoc             'Reference newly created calc document.

    oDoc = createMultiplicationTable()
    oSheet = oDoc.Sheets(0)

    REM Do this for the entire sheet!
    x = oSheet.getCellFormatRanges()
    s = "**** getCellFormatRanges() " & CHR$(10)
    For i = 0 To x.getcount()-1
        oSheetCellRange = x.getByIndex(i)
        oAddr = oSheetCellRange.getRangeAddress()
        s = s & prettyRangeAddressName(oAddr) & CHR$(10)
    Next

    REM SheetCellRanges
    x = oSheet.getUniqueCellFormatRanges()
    s = s & CHR$(10) & "**** getUniqueCellFormatRanges() " & CHR$(10)
    For i = 0 To x.getcount()-1
        oSheetCellRanges = x.getByIndex(i)
        s = s & i & " = "& oSheetCellRanges.getRangeAddressesAsString() & CHR$(10)
    Next
    MsgBox s, 0, "Like Ranges"
End Sub

```

Figure 103 demonstrates very clearly the difference between the two methods. The `getCellFormatRanges()` method continually lists the rectangular ranges that are formatted similarly: with each change of format a new range starts. The `getUniqueCellFormatRanges()`

method groups the found ranges in containers that contain ranges of similar format. So out of the seven ranges there are five (index 0) resp. two (index 1) that are formatted similarly. Both methods contain the same ranges; it is only a question of grouping.

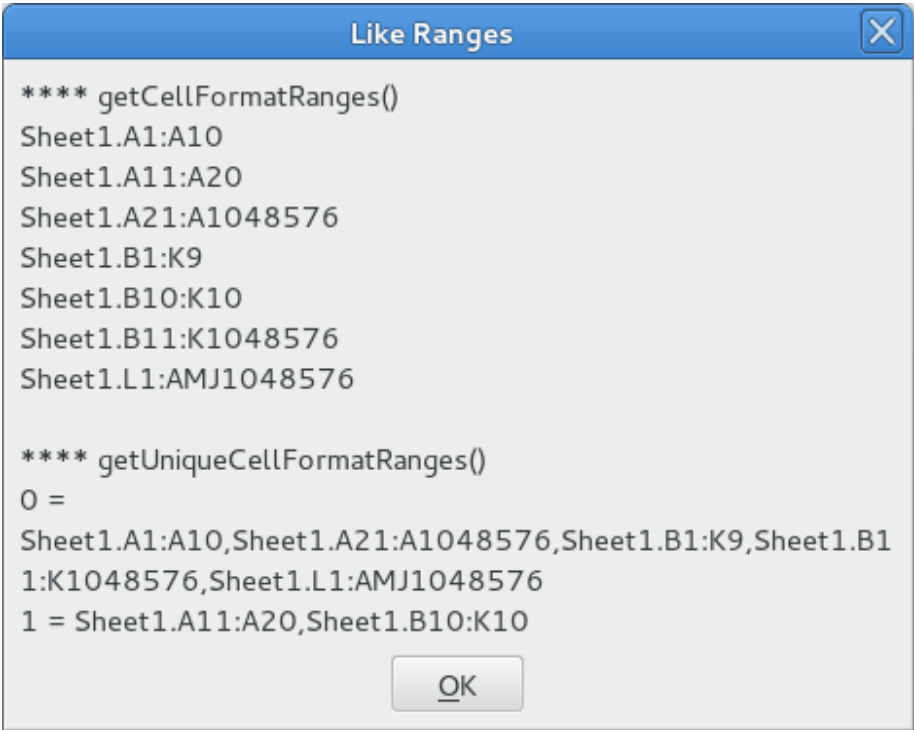


Figure 103. The two methods group the data differently.

15.4.13. Sorting

In general, OOo knows what type of data a cell contains. Knowledge of the contained data types reduces the need to inform OOo of the type of data that it is sorting, but it is still possible using the TableSortFieldType enumeration (see Table 188). When you request a sort operation, an array of TableSortField structures identifies which columns or rows are used to determine the sort order and how they are sorted (see Table 189).

Table 188. The com.sun.star.table.TableSortFieldType enumeration.

Value	Description
com.sun.star.table.TableSortFieldType.AUTOMATIC	Automatically determine the data type.
com.sun.star.table.TableSortFieldType.NUMERIC	Sort the data as a number.
com.sun.star.table.TableSortFieldType.ALPHANUMERIC	Sort the data as text.

Table 189. *The com.sun.star.table.TableSortField structure.*

Property	Description
Field	Zero-based index of the row or column in the table to sort. The index is relative to the start of the sort range.
IsAscending	If True, sort the data in ascending order.
IsCaseSensitive	If True, the sort is case sensitive.
FieldType	Specify the data type as a TableSortFieldType (see Table 188).
CollatorLocale	The Locale object to use when sorting text.
CollatorAlgorithm	The sorting algorithm used by the collator when sorting text. Check the interface com.sun.star.i18n.XCollator to investigate what algorithms are supported for your locale. I have always used the default value.

When you request a sort operation, an array of properties is passed to the sort routine. The properties determine what to sort and how it is sorted. One of the supported properties is SortFields (see Table 190), which contains the array of TableSortField structures that determine how the rows or columns are sorted.

Table 190. *The old way to specify a sort using a SortDescriptor.*

Property	Description
IsCaseSensitive	If True, the sort is case sensitive.
SortAscending	If True, sort the data in ascending order. This property typically is not used, because the TableSortField specifies IsAscending for each field.
SortColumns	If True, columns are sorted. If False, rows are sorted.
CollatorLocale	The Locale object to use when sorting text (usually set in the TableSortField).
CollatorAlgorithm	Sorting algorithm to use (usually set in the TableSortField).
SortFields	Array of type TableSortField (see Table 189) that directs what is sorted.
MaxSortFieldsCount	Long Integer that specifies the maximum number of sort fields the descriptor can hold. This value cannot be set, but it can be read.
ContainsHeader	If True, the first row or column is considered a header and is not sorted.
Orientation	This property is deprecated and should no longer be used!

Table 189 and Table 190 reveal that there is a great deal of redundancy. For example, you can specify if the sort is case sensitive either globally (using Table 190) or for each specific field (using Table 189). Although the redundant fields in Table 190 are not required and are therefore typically not used, a new set of sort descriptors has been introduced (see Table 191). Although you can use properties from Table 190 or Table 191, you cannot mix the two. My recommendation is that you use the new set in Table 191; the OO development team has already deprecated the use of the Orientation property in Table 190.

Table 191. *The new way to specify a sort using a SortDescriptor2.*

Property	Description
SortFields	Array of type TableSortField (see Table 189) that directs what is sorted.
MaxSortFieldsCount	Long Integer that specifies the maximum number of sort fields the descriptor can hold. This value cannot be set, but it can be read.
IsSortColumns	If True, columns are sorted. If False, rows are sorted.
BindFormatsToContent	If True, cell formats are moved with the contents during the sort. This property matters only if different cells in the sort range use different formatting.
IsUserListEnabled	If True, a user-defined sorting list is used from the GlobalSheetSettings.
UserListIndex	Specify which user-defined sorting list is used as a Long Integer.
CopyOutputData	If True, the sorted data is copied to another position in the document.
OutputPosition	CellAddress that specifies where to copy the sorted data (if CopyOutputData is True).
ContainsHeader	If True, the first row or column is considered a header and is not sorted.

The first step in sorting a range is to define the fields on which to sort by using an array of type SortField. Next, define the properties from Table 191 that you intend to use in the sort. Finally, call the sort() routine on the range to sort. The macro in Listing 431 performs a descending sort on the first column.

Listing 431. *Sort one column in a Calc sheet.*

```
Sub SortColZero
    Dim oSheet
    Dim oRange
    Dim oSortFields(0) as new com.sun.star.util.SortField
    Dim oSortDesc(0) as new com.sun.star.beans.PropertyValue

    oSheet = ThisComponent.Sheets(0)
    REM Set the range on which to sort
    oRange = oSheet.getCellRangeByName("B28:D33")

    REM Sort on the first field in the range
    oSortFields(0).Field = 0
    oSortFields(0).SortAscending = FALSE

    REM Set the sort fields to use
    oSortDesc(0).Name = "SortFields"
    oSortDesc(0).Value = oSortFields()

    REM Now sort the range!
    oRange.Sort(oSortDesc())
End Sub
```

Sorting on two columns rather than just one is as easy as adding a second sort field. Listing 432 sorts on the second and third columns.

Listing 432. *Sort two columns in a Calc sheet.*

```
Sub SortColOne
```



```

Dim oSheet
Dim oRange
Dim oSortFields(1) as new com.sun.star.util.SortField
Dim oSortDesc(0) as new com.sun.star.beans.PropertyValue

oSheet = ThisComponent.Sheets(0)

REM Set the range on which to sort
oRange = oSheet.getCellRangeByName("B28:D33")

REM Sort on the second field in the range
oSortFields(0).Field = 1
oSortFields(0).SortAscending = True
oSortFields(0).FieldType = com.sun.star.util.SortFieldType.NUMERIC

REM Sort on the third field in the range
oSortFields(1).Field = 2
oSortFields(1).SortAscending = True
oSortFields(1).FieldType = com.sun.star.util.SortFieldType.ALPHANUMERIC

REM Set the sort fields to use
oSortDesc(0).Name = "SortFields"
oSortDesc(0).Value = oSortFields()

REM Now sort the range!
oRange.Sort(oSortDesc())
End Sub

```

The method `createSortDescriptor()` returns an array of property values that define how a sort should occur. Inspecting this created sort descriptor indicates that you can use a maximum of three fields when sorting (see the `MaxSortFieldsCount` in Table 191). The macro in Listing 433 creates a sort descriptor and then displays the properties that it contains (see Figure 104).

Listing 433. Display sort descriptor properties in Calc.

```

Sub DisplaySortDescriptor
On Error Resume Next
Dim oSheet
Dim oRange      ' A range is needed to create the sort descriptor.
Dim oSortDescript
Dim i%
Dim s$
Dim oDoc        'Reference newly created calc document.

oDoc = StarDesktop.loadComponentFromURL("private:factory/scalc", "_default", 0, Array())
oSheet = oDoc.Sheets(0)
oRange = oSheet.getCellRangeByName("B28:D33")
oSortDescript = oRange.createSortDescriptor()
For i = LBound(oSortDescript) To UBound(oSortDescript)
    s = s & oSortDescript(i).Name & " = "
    s = s & oSortDescript(i).Value
    s = s & CHR$(10)
Next

```

```
MsgBox s, 0, "Sort Descriptor"  
End Sub
```

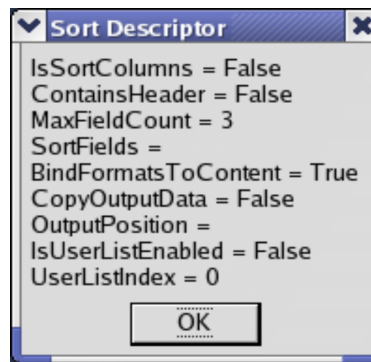


Figure 104. Sort descriptor attributes.

15.5. Sheets

Most of a Calc document's functionality is contained in the individual sheets rather than the document as a whole. The interfaces implemented by the spreadsheet document (see Table 161) are primarily related to the document as a whole, rather than the individual sheets.

The sheets in a Calc document implement the SheetCellRange service, which provides extensive functionality. The functionality provided by sheet cell ranges applies to any sheet cell range and is not limited to sheets. In other words, any range is able to use the methods implemented by the SheetCellRange service. The individual sheets support other interfaces that are not directly related to sheet cell ranges (see Table 192).

Table 192. Interfaces implemented by the *com.sun.star.sheet.Spreadsheet* service.

Interfaces	Description
<code>com.sun.star.sheet.XSpreadsheet</code>	Methods to create a cell cursor.
<code>com.sun.star.container.XNamed</code>	Access the spreadsheet name.
<code>com.sun.star.util.XProtectable</code>	Methods to protect and unprotect the individual sheet.
<code>com.sun.star.sheet.XDataPilotTablesSupplier</code>	Access the DataPilot tables with the method <code>getDataPilotTables()</code> . In LibreOffice, the Data Pilot is called Pivot Tables.
<code>com.sun.star.sheet.XScenariosSupplier</code>	Access the scenarios with the method <code>getScenarios()</code> .
<code>com.sun.star.sheet.XSheetAnnotationsSupplier</code>	Access the annotations with the method <code>getAnnotations()</code> .
<code>com.sun.star.drawing.XDrawPageSupplier</code>	Access the sheet's draw page with the method <code>getDrawPage()</code> .
<code>com.sun.star.table.XTableChartsSupplier</code>	Access the document's chart objects with the method <code>getCharts()</code> .
<code>com.sun.star.sheet.XCellRangeMovement</code>	Move cell ranges inside the sheet or to other spreadsheets in this document.
<code>com.sun.star.sheet.XPrintAreas</code>	Access to the print-area settings of this sheet.
<code>com.sun.star.sheet.XSheetPageBreak</code>	Access and modify the page breaks in this sheet.
<code>com.sun.star.sheet.XScenario</code>	Provide methods for a scenario sheet.
<code>com.sun.star.sheet.XSheetOutline</code>	Access the row and column outline settings for the sheet.
<code>com.sun.star.sheet.XSheetAuditing</code>	Look for linked cells (detective).
<code>com.sun.star.sheet.XSheetLinkable</code>	Methods to link to existing sheets in other documents.

15.5.1. Linking to an external spreadsheet

An individual sheet may link to a sheet from another spreadsheet document. Linking causes the “link sheet” to act as a container for the “linked sheet.” After linking to a sheet, although you can modify the linked sheet in the container, these updates are not propagated back to the original document. If the linked sheet is changed in the original document, the change is not visible in the link document unless the link itself is refreshed. You can link documents by using one of the enumerated values in Table 193.

Table 193. Values supported by the *com.sun.star.sheet.SheetLinkMode* enumeration.

Value	Description
<code>com.sun.star.sheet.SheetLinkMode.NONE</code>	The sheet is not linked.
<code>com.sun.star.sheet.SheetLinkMode.NORMAL</code>	Copy the entire content including values and formulas.
<code>com.sun.star.sheet.SheetLinkMode.VALUE</code>	Copy the content by value; each formula's returned value is copied rather than the formula itself.

Use the `link()` method to establish a link with a sheet in another document. Table 194 lists the link-related methods supported by a sheet.

Table 194. Methods defined by the *com.sun.star.sheet.XSheetLinkable* interface.

Method	Description
getLinkMode()	Get the sheet's link mode (see Table 193).
getLinkMode(SheetLinkMode)	Set the link mode (see Table 193).
getLinkUrl()	Get the link URL.
setLinkUrl(url)	Set the link URL.
getLinkSheetName()	Get the name of the linked sheet.
setLinkSheetName(name)	Set the name of the linked sheet.
link(url, sheetName, filterName, filterOptions, SheetLinkMode)	Link the sheet to another sheet in another document.

The macro in Listing 434 creates a sheet named “LinkIt” and then links to a sheet in a specified external document. If the “LinkIt” sheet already exists, the link is obtained from the spreadsheet document and the link is refreshed. Refreshing a link causes the data linked into the current document to be updated.

Listing 434. Link to an external sheet.

```
Sub LinkASheet
    Dim oSheets          'The sheets object that contains all of the sheets
    Dim oSheet           'Individual sheet
    Dim oSheetEnum       'For accessing by enumeration
    Dim s As String      'String variable to hold temporary data
    Dim i As Integer     'Index variable
    Dim sURL As String   'URL of the document to import
    Dim oLink            'The link object

    sURL = "file:///C:/My%20Documents/CH15/test.ods"
    oSheets = ThisComponent.Sheets

    If oSheets.hasByName("LinkIt") Then
        REM The links are available from the document object
        REM based on the URL used to load them.
        oLink = ThisComponent.SheetLinks.getByNamed(sURL)
        oLink.refresh()
        MsgBox "The sheet named LinkIt was refreshed"
        Exit Sub
    End If

    REM Insert the new sheet at the end.
    oSheets.insertNewByName ("LinkIt", oSheets.getCount())
    oSheet = oSheets.getByNamed("LinkIt")

    oSheet.link(sURL, "Sheet1", "", "", com.sun.star.sheet.SheetLinkMode.NORMAL)
End Sub
```

The first application that I saw for linked sheets was to consolidate a list of various investments that were tracked in different Calc documents. Each of the Calc documents contained a summary

sheet for the investments in the document. A single summary document inserted links to the summary page for each of the other investment sheets.

Although linked sheets are nice, they are sometimes overkill. If you don't want to reference an entire sheet from another document, you can set the formula to directly access just one cell. See Listing 435.

Listing 435. Link cell A1 to K89 in another document.

```
oCell = thiscomponent.sheets(0).getcellbyposition(0,0) ' A1
oCell.setFormula("=" & "'file:///home/USER/CalcFile2.odt'#$Sheet2.K89")
```

15.5.2. Finding dependencies by using auditing functions

The methods queryDependents() and queryPrecedents(), listed in Table 179, return a list of cells that depend on a range. The query methods are useful for writing macros that manipulate each dependent cell. The auditing functionality provided by the XSheetAuditing interface provides methods for visualizing cell dependencies (see Table 195).

Table 195. Methods supported by the com.sun.star.sheet.XSheetAuditing interface.

Method	Description
hideDependents(CellAddress)	Remove arrows for one level of dependents; return True if cells are marked.
hidePrecedents(CellAddress)	Remove arrows for one level of precedents; return True if cells are marked.
showDependents(CellAddress)	Draw arrows from the CellAddress (see Table 163) to its dependents; return True if cells are marked.
showPrecedents(CellAddress)	Draw arrows to the CellAddress (see Table 163) from its precedents; return True if cells are marked.
showErrors(CellAddress)	Draw arrows from the CellAddress (see Table 163) containing an error and the cells causing the error; return True if cells are marked.
showInvalid()	Show all cells containing invalid values; return True if cells are marked.
clearArrows()	Remove all auditing arrows from the sheet.

Each time that the showPrecedents() method is called, another level of precedents is marked with arrows. After the first call, arrows are drawn from all cells directly referenced by the specified cell. The QueryRange macro in Listing 419 displays the dependencies (see Figure 99); Listing 436, however, displays the precedents in the spreadsheet. Figure 105 shows one level of precedents.

The cell B7 contains the formula “=Sum(B1:B6)”. As Figure 105 shows, cell B7 refers to all of the “summed” cells. If you call the method showPrecedents() again, you'll see the cells that reference the cells B2:B6. The showPrecedents() method returns True as long as more precedent cells are marked with arrows. Figure 106 shows the next level of precedents.

Listing 436. Display precedents.

```
Function SimpleCalcDocAddition()
    Dim oDoc 'Reference newly created calc document.
    Dim oSheet
```

```

Dim oRange

oDoc = StarDesktop.loadComponentFromURL("private:factory/scalc", "_default", 0, Array())
oSheet = oDoc.Sheets(0)
oSheet.getCellByPosition(0, 0).setValue(1)
oRange = oSheet.getCellRangeByName("A1:A6")
oRange.fillAuto(com.sun.star.sheet.FillDirection.TO_BOTTOM, 1)
oRange = oSheet.getCellRangeByName("B1:B6")
oRange.setArrayFormula("=A1:A6+1")
oSheet.getCellByPosition(1, 6).setFormula("=Sum(B1:B6) ")
SimpleCalcDocAddition = oDoc
End Function

Sub ShowCellPrecedence()
    Dim oDoc
    Dim oSheet
    Dim oAddr

    oDoc = SimpleCalcDocAddition()
    oSheet = oDoc.Sheets(0)
    oAddr = oSheet.getCellByPosition(1, 6).CellAddress
    oSheet.showPrecedents(oAddr)
    Print "See one level of Precedents for cells B1:B6"
    oSheet.showPrecedents(oAddr)
    Print "See two levels of Precedents for cell B1:B6"
End Sub

```

	A	B
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7		27

Figure 105. One level of precedents.

	A	B
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7		27

Figure 106. Two levels of precedents with an array formula in B1:B6.

The arrows in Figure 106 demonstrate two levels of precedents assuming an array formula. If a simple formula is used in each cell rather than an array formula, the figure changes noticeably. As an exercise, consider why this is.

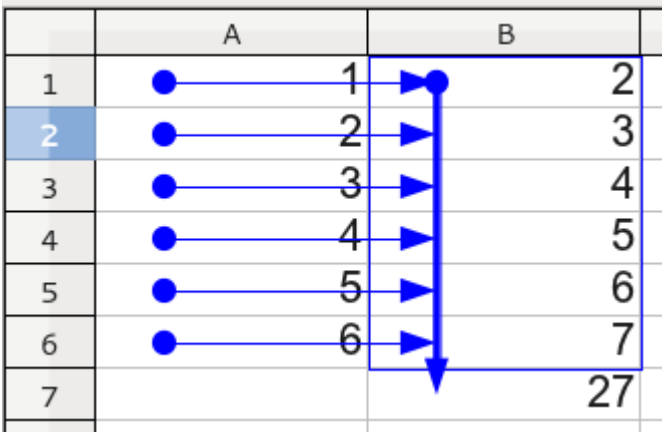


Figure 107. Two levels of precedents with formulas in B2:B6.

15.5.3. Outlines

Outlines in a Calc document group rows and columns together so that you can collapse and expand the groups with a single mouse click. When you create an outline, you must specify whether it’s row-centric or column-centric by using TableOrientation enumeration (see Table 196). The methods in Table 197 behave like their menu counterparts in the OOo GUI for dealing with spreadsheet outlines.

Table 196. Values defined by the `com.sun.star.table.TableOrientation` enumeration.

Value	Description
<code>com.sun.star.table.TableOrientation.ROWS</code>	Use rows.
<code>com.sun.star.table.TableOrientation.COLUMNS</code>	Use columns.

Table 197. Methods supported by the `com.sun.star.sheet.XSheetOutline` interface.

Method	Description
<code>group(CellRangeAddress, TableOrientation)</code>	Group the cells in the cell range into one group.
<code>ungroup(CellRangeAddress, TableOrientation)</code>	Remove the innermost levels from the group.
<code>autoOutline(CellRangeAddress)</code>	Create outline groups based on formula references.
<code>clearOutline()</code>	Remove all outline groups from the sheet.
<code>hideDetail(CellRangeAddress)</code>	Collapse an outline group.
<code>showDetail(CellRangeAddress)</code>	Open (uncollapse) an outline group.
<code>showLevel(n, CellRangeAddress)</code>	Show outlined groups from levels one through n.

15.5.4. Copying, moving, and inserting cells

In a Writer document, the primary method for moving or copying text content is to use the clipboard. The Spreadsheet service, however, provides methods for directly moving and inserting cells. When new cells are inserted, you specify how cells are moved out of the way by using the `CellInsertMode` enumeration (see Table 198).

Table 198. Values defined by the `com.sun.star.sheet.CellInsertMode` enumeration.

Value	Description
<code>com.sun.star.sheet.CellInsertMode.NONE</code>	No cells are moved.
<code>com.sun.star.sheet.CellInsertMode.DOWN</code>	Move cells down.
<code>com.sun.star.sheet.CellInsertMode.RIGHT</code>	Move cells right.
<code>com.sun.star.sheet.CellInsertMode.ROWS</code>	Move the entire row down.
<code>com.sun.star.sheet.CellInsertMode.COLUMNS</code>	Move the entire column right.

Use the `insertCells(CellRangeAddress, CellInsertMode)` method to create space the size of the cell range address. If the insert mode is `COLUMNS`, then the entire column, starting with the leftmost column in the range, is shifted to the right the width of the range. If the insert mode is `RIGHT`, then the entire column is not shifted right; only the rows in the range are shifted. The cell insert modes `ROWS` and `DOWN` behave similarly to the `COLUMNS` and `RIGHT` modes. Using the cell insert range of `NONE` causes no cells to be moved; in other words, nothing happens. Listing 437 moves cells down.

Listing 437. Move the range `L4:M5` down.

```
Dim oSheet
Dim oRangeAddress 'The range to move
oSheet = ThisComponent.Sheets(0)
oRangeAddress = oSheet.getCellRangeByName("L4:M5").getRangeAddress()
oSheet.insertCells(oRangeAddress, com.sun.star.sheet.CellInsertMode.DOWN)
```

Tip The `insertCells()` and `removeRange()` methods silently fail if the insertion will cause an array formula to be split.

The `removeRange(CellRangeAddress, CellInsertMode)` method is essentially an “undo” command for the `insertCells()` method.

Use the `copyRange(CellAddress, CellRangeAddress)` method to copy a range of cells to the location specified by the cell address. The top-left cell in the cell range address is positioned at the specified cell address when the range is copied. The net effect of the `copyRange()` method is the same as copying a range of cells to the clipboard, positioning the cursor at the specified cell, and then pasting the cells into place (see Listing 438).

Listing 438. Copy the range `L4:M5` to `N8`.

```
Dim oSheet
Dim oRangeAddress 'The range to copy
Dim oCellAddress 'Destination address
```



```

oSheet = ThisComponent.Sheets(0)
oRangeAddress = oSheet.getCellRangeByName("L4:M5").getRangeAddress()
oCellAddress = oSheet.getCellByPosition(13, 7).getCellAddress() 'N8
oSheet.copyRange(oCellAddress, oRangeAddress)

```

Use the `moveRange(CellAddress, CellRangeAddress)` method to move (rather than copy) a range of cells. The behavior of the `moveRange()` method is similar to that of the `copyRange()` method except that the cells are moved, rather than copied; the cells in the original range are left empty.

15.5.5. Copy data between documents

The `copyRange` method only supports copying a range in the same document. Other methods must be used to copy data between different documents.

Data functions

Use `getData()` and `setData()` to copy numbers. Use `getDataArray()` and `setDataArray()` to copy numbers and strings. The data functions are fast and easy, but they only copy data, not formatting.

Clipboard

Use dispatch commands to copy data using the clipboard. The clipboard provides many options when copying data. The primary issue with using the clipboard, is that the clipboard is inherently unsafe because other applications may be using the clipboard at the same time. Although the clipboard is not inherently safe, Paste Special provides numerous options when pasting data not available using other methods.

Listing 439. Use the clipboard to copy data between two documents.

```

oDispatcher = createUnoService("com.sun.star.frame.DispatchHelper")
oFrame1 = oDoc1.CurrentController.Frame
' Use the documents controller to select the cells A1:B2.
oSheet = oDoc1.Sheets(0)
rng = oSheet.getCellRangeByName("A1:B2")
oDoc1.CurrentController.Select(rng)
' Use a dispatch to copy to the clipboard.
oDispatcher.executeDispatch(oFrame1, ".uno:Copy", "", 0, Array())
' Upper left corner of where to paste the data.
rng = oDoc2.Sheets(0).getCellRangeByName("A1")
' Place the view cursor there then paste the clipboard.
oDoc2.CurrentController.Select(rng)
oFrame2 = oDoc2.CurrentController.Frame
oDispatcher.executeDispatch(oFrame2, ".uno:Paste", "", 0, Array())

```

Use the macro recorder with Paste Special to see how arguments are set with paste special. One example is as follows:

Listing 440. Arguments for Paste Special.

```

Dim args1(5) as new com.sun.star.beans.PropertyValue
args1(0).Name = "Flags"
args1(0).Value = "SVDNT"

```

```

args1(1).Name = "FormulaCommand"
args1(1).Value = 0
args1(2).Name = "SkipEmptyCells"
args1(2).Value = false
args1(3).Name = "Transpose"
args1(3).Value = false
args1(4).Name = "AsLink"
args1(4).Value = false
args1(5).Name = "MoveMode"
args1(5).Value = 4
oDispatcher.executeDispatch(oFrame2, ".uno:InsertContents", "", 0, args1())

```

Transferable content

Transferable content, the newest method available for copying content, returns a copy of the selected data as though it was copied to the clipboard, without using the clipboard. Transferable content is also available in Writer documents. Unlike the clipboard, there is no inherent problems with transferable content, but you also do not have the flexibility of paste special.

Listing 441. Use transferable content to copy data between two documents.

```

Dim o
Dim oSheet
Dim oRange
Dim oDoc

oRange = oDoc1.Sheets(0).getCellRangeByName("B2:C3")
oDoc1.CurrentController.select(oRange)
o = oDoc1.CurrentController.getTransferable()

oRange = oDoc2.Sheets(0).getCellRangeByName("F1")
oDoc2.CurrentController.select(oRange)
oDoc2.CurrentController.insertTransferable(o)

```

15.5.6. Data pilot and pivot tables

OpenOffice.org has a feature named Data Pilot, but LibreOffice changed the name the Pivot Tables. This section assumes that you already know and understand the feature, and merely demonstrates how to implement some of the functions using macros.

Tip

I could write a large section on the numerous uses of data pilot tables. To get a feel for the possibilities, look at the input table in Table 199 and then inspect the summary data in Figure 108, which is automatically generated by the data pilot functionality.

The DataPilot is a powerful mechanism that allows you to combine, compare, and analyze large amounts of data. The DataPilot manipulates portions of the data from the “source table” and then displays the results in a new location. Unfortunately, numerous details are involved with the creation and manipulation of data pilot tables, but this is due to their enormous flexibility.

There are numerous details with respect to creating and using data pilot tables. Although the numerous details are logical and straightforward, they are so many that it is easy to become lost

in the details. It's instructive, therefore, to review a simple example that clarifies the details. I'll present the specific types and enumerations following the example; you can review these as required.

A data pilot example

For this example, I assume a fake company that sells books, candy, and pens. The company has offices in three states and multiple salespeople in each state. I created a spreadsheet that shows the sales for each product broken down by salesperson and year. The final goal of this example is to create a data pilot table that summarizes the sales of each product by type and state. The initial data used for this example is shown in Table 199.

Table 199. Data values used for the DataPilot / PivotTable examples.

Item	State	Team	2002	2003	2004
Books	Michigan	Jean	\$14,788.00	\$30,222.00	\$23,490.00
Candy	Michigan	Jean	\$26,388.00	\$15,641.00	\$32,849.00
Pens	Michigan	Jean	\$16,569.00	\$32,675.00	\$25,396.00
Books	Michigan	Volker	\$21,961.00	\$21,242.00	\$29,009.00
Candy	Michigan	Volker	\$26,142.00	\$22,407.00	\$32,841.00
Pens	Michigan	Volker	\$29,149.00	\$18,320.00	\$34,429.00
Books	Ohio	Rebecca	\$21,845.00	\$33,503.00	\$32,200.00
Candy	Ohio	Rebecca	\$23,799.00	\$23,597.00	\$23,020.00
Pens	Ohio	Rebecca	\$28,328.00	\$17,930.00	\$23,303.00
Books	Ohio	Catherine	\$22,797.00	\$31,386.00	\$39,490.00
Candy	Ohio	Catherine	\$13,613.00	\$16,174.00	\$35,163.00
Pens	Ohio	Catherine	\$17,103.00	\$32,563.00	\$35,804.00
Books	Kentucky	Michelle	\$29,952.00	\$19,133.00	\$33,480.00
Candy	Kentucky	Michelle	\$15,348.00	\$31,094.00	\$39,722.00
Pens	Kentucky	Michelle	\$24,358.00	\$27,236.00	\$27,129.00
Books	Kentucky	Andy	\$17,199.00	\$26,386.00	\$30,450.00
Candy	Kentucky	Andy	\$11,628.00	\$18,232.00	\$28,953.00
Pens	Kentucky	Andy	\$23,828.00	\$32,031.00	\$37,551.00

Generating the data

The data shown in Table 199 is generated by the macro in Listing 442, which sets both the data and the formatting. Watch for the following techniques:

- Generating random data.
- Setting all of the data at one time using the method `setDataArray()`.
- Centering the headers and setting the cell background color.
- Formatting a cell as currency.

Listing 442. *Create the document used for the DataPilot / PivotTable examples.*

```
Function createPivotTableDoc()  
    Dim oDoc          'Reference newly created calc document.  
    Dim oSheet  
    Dim oRange  
    Dim oData  
    Dim oFormats  
  
    oData = Array(Array("Item", "State", "Team", "2002", "2003", "2004"), _  
        Array("Books", "Michigan", "Jean", 14788, 30222, 23490), _  
        Array("Candy", "Michigan", "Jean", 26388, 15641, 32849), _  
        Array("Pens", "Michigan", "Jean", 16569, 32675, 25396), _  
        Array("Books", "Michigan", "Volker", 21961, 21242, 29009), _  
        Array("Candy", "Michigan", "Volker", 26142, 22407, 32841), _  
        Array("Pens", "Michigan", "Volker", 29149, 18320, 34429), _  
        Array("Books", "Ohio", "Rebecca", 21845, 33503, 32200), _  
        Array("Candy", "Ohio", "Rebecca", 23799, 23597, 23020), _  
        Array("Pens", "Ohio", "Rebecca", 28328, 17930, 23303), _  
        Array("Books", "Ohio", "Catherine", 22797, 31386, 39490), _  
        Array("Candy", "Ohio", "Catherine", 13613, 16174, 35163), _  
        Array("Pens", "Ohio", "Catherine", 17103, 32563, 35804), _  
        Array("Books", "Kentucky", "Michelle", 29952, 19133, 33480), _  
        Array("Candy", "Kentucky", "Michelle", 15348, 31094, 39722), _  
        Array("Pens", "Kentucky", "Michelle", 24358, 27236, 27129), _  
        Array("Books", "Kentucky", "Andy", 17199, 26386, 30450), _  
        Array("Candy", "Kentucky", "Andy", 11628, 18232, 28953), _  
        Array("Pens", "Kentucky", "Andy", 23828, 32031, 37551))  
  
    oDoc = StarDesktop.loadComponentFromURL("private:factory/scalc", "_default", 0, Array())  
    oSheet = oDoc.Sheets(0)  
    oSheet.getCellRangeByName("A1:F19").setDataArray(oData)  
  
    ' Set the title area  
    oRange = oSheet.getCellRangeByName("A1:F1")  
    oRange.HoriJustify = com.sun.star.table.CellHoriJustify.CENTER  
    oRange.CellBackColor = RGB(225, 225, 225)  
  
    ' Format numbers as currency  
    oFormats = oDoc.NumberFormats  
    Dim aLocale As New com.sun.star.lang.Locale  
    oRange = oSheet.getCellRangeByName("D2:F19")  
    oRange.NumberFormat = oFormats.getStandardFormat(_  
        com.sun.star.util.NumberFormat.CURRENCY, aLocale)  
    createPivotTableDoc = oDoc  
End Function
```

Creating the data pilot table

The macro in Listing 443 creates and inserts a data pilot table as follows:

1. Create the data pilot table descriptor using the method `createDataPilotDescriptor()`.
2. Set the source range of the data to use.

3. Configure which column is used for which purpose.
4. Insert the data pilot table descriptor into the set of tables.

Listing 443. Create a data pilot table.

```
Sub CreateDataPilotTable()  
    Dim oSheet          'Sheet that contains the data pilot  
    Dim oRange          'Range for the data pilot source  
    Dim oRangeAddress    'The address of the range object  
    Dim oTables         'Collection of data pilot tables  
    Dim oTDescriptor     'A single data pilot descriptor  
    Dim oFields         'Collection of all fields  
    Dim oField          'A single field  
    Dim oCellAddress As New com.sun.star.table.CellAddress  
    Dim oDoc  
  
    oDoc = createPivotTableDoc()  
    oSheet = oDoc.Sheets.getByIndex(0)  
    oRange = oSheet.getCellRangeByName("A1:F19")  
  
    REM Sure, I could simply specify the address, but this is much more fun!  
    REM Set the destination address to be two rows below the data.  
    oRangeAddress = oRange.getRangeAddress()  
    oCellAddress.Sheet = oRangeAddress.Sheet  
    oCellAddress.Column = oRangeAddress.StartColumn  
    oCellAddress.Row = oRangeAddress.EndRow + 2  
  
    oTables = oSheet.getDataPilotTables()  
  
    REM Step 1, create the descriptor  
    oTDescriptor = oTables.createDataPilotDescriptor()  
  
    REM Step 2, Set the source range  
    oTDescriptor.setSourceRange(oRangeAddress)  
  
    REM Step 3, Set the fields  
    oFields = oTDescriptor.getDataPilotFields()  
  
    REM Column 0 in the source is Item and I want this as a row item.  
    oField = oFields.getByIndex(0)  
    oField.Orientation = com.sun.star.sheet.DataPilotFieldOrientation.ROW  
  
    REM Column 1 in the source is State and I want this as a column item.  
    oField = oFields.getByIndex(1)  
    oField.Orientation = com.sun.star.sheet.DataPilotFieldOrientation.COLUMN  
  
    REM Column 3 in the source is 2002. Create a sum in the data for this!  
    oField = oFields.getByIndex(3)  
    oField.Orientation = com.sun.star.sheet.DataPilotFieldOrientation.DATA  
    oField.Function = com.sun.star.sheet.GeneralFunction.SUM  
  
    oTables.insertNewByName("MyFirstDataPilot", oCellAddress, oTDescriptor)  
End Sub
```

21	Filter				
22					
23	Sum - 2002	State v			
24	Item v	Kentucky	Michigan	Ohio	Total Result
25	Books	\$47,151.00	\$36,749.00	\$44,642.00	\$128,542.00
26	Candy	\$26,976.00	\$52,530.00	\$37,412.00	\$116,918.00
27	Pens	\$48,186.00	\$45,718.00	\$45,431.00	\$139,335.00
28	Total Result	\$122,313.00	\$134,997.00	\$127,485.00	\$384,795.00

Figure 108. The macro in Listing 443 inserts the data pilot table immediately after the source data.

Manipulating data pilot tables

The `getDataPilotTables()` method, supported by each spreadsheet, returns an object that supports the service `com.sun.star.sheet.DataPilotTables`. The returned service provides access to the data pilot tables in the spreadsheet using both indexed and enumeration access. The `DataPilotTables` object also supports the methods in Table 200.

Table 200. Methods defined by the `com.sun.star.sheet.XDataPilotTables` interface.

Method	Description
<code>createDataPilotDescriptor()</code>	Create a new data pilot descriptor.
<code>insertNewByName(name, CellAddress, DataPilotDescriptor)</code>	Add a new data pilot table to the collection that uses the provided <code>CellAddress</code> (see Table 163) as the top-left corner of the table.
<code>removeByName(name)</code>	Delete a data pilot table from the collection.

Data pilot fields

Each “field” in the created data pilot table is represented by a column in the source data pilot table (cell range) and is named using the topmost cell of the column in the range. The field name is available through the methods `getName()` and `setName(String)`.

Each field contains an `Orientation` property of type `DataPilotFieldOrientation` that specifies how the field is used in the final output (see Table 201). The `Function` property specifies the function used to calculate results for this field based on the `GeneralFunction` enumeration (see Table 182).

Table 201. Values defined by the com.sun.star.sheet.DataPilotFieldOrientation enumeration.

Value	Description
com.sun.star.sheet.DataPilotFieldOrientation.HIDDEN	Do not use the field.
com.sun.star.sheet.DataPilotFieldOrientation.COLUMN	Use the field as a column field.
com.sun.star.sheet.DataPilotFieldOrientation.ROW	Use the field as a row field.
com.sun.star.sheet.DataPilotFieldOrientation.PAGE	Use the field as a page field.
com.sun.star.sheet.DataPilotFieldOrientation.DATA	Use the field as a data field.

Filtering data pilot fields

The fields in the created table may be conditionally displayed based on a FilterOperator (see Table 202).

Table 202. Values defined by the com.sun.star.sheet.FilterOperator enumeration.

Value	Description
com.sun.star.sheet.FilterOperator.EMPTY	Select empty entries.
com.sun.star.sheet.FilterOperator.NOT_EMPTY	Select nonempty entries.
com.sun.star.sheet.FilterOperator.EQUAL	The entry's value must equal the specified value.
com.sun.star.sheet.FilterOperator.NOT_EQUAL	The entry's value must not be equal to the specified value.
com.sun.star.sheet.FilterOperator.GREATER	The entry's value must be greater than the specified value.
com.sun.star.sheet.FilterOperator.GREATER_EQUAL	The entry's value must be greater than or equal to the specified value.
com.sun.star.sheet.FilterOperator.LESS	The entry's value must be less than the specified value.
com.sun.star.sheet.FilterOperator.LESS_EQUAL	The entry's value must be less than or equal to the specified value.
com.sun.star.sheet.FilterOperator.TOP_VALUES	Select a specified number with the greatest values.
com.sun.star.sheet.FilterOperator.TOP_PERCENT	Select a specified percentage with the greatest values.
com.sun.star.sheet.FilterOperator.BOTTOM_VALUES	Select a specified number with the lowest values.
com.sun.star.sheet.FilterOperator.BOTTOM_PERCENT	Select a specified percentage with the lowest values.

The individual filter operators are combined using the FilterConnection enumeration (see Table 203). Each individual filter field is stored in a TableFilterField structure (see Table 204). The entire collection of TableFilterFields is stored in a SheetFilterDescriptor. The descriptor supports the methods getFilterFields() and setFilterFields(), to get and set the table filter fields as an array of TableFilterField structures. The properties in Table 205 are defined by the SheetFilterDescriptor to direct the filtering process.

Table 203. Values defined by the com.sun.star.sheet.FilterConnection enumeration.

Value	Description
com.sun.star.sheet.FilterConnection.AND	Both conditions must be satisfied.
com.sun.star.sheet.FilterConnection.OR	At least one of the conditions must be satisfied.

Table 204. Properties in the com.sun.star.sheet.TableFilterField structure.

Property	Description
Connection	Specify how the condition is connected to the previous condition as a FilterConnection (see Table 203).
Field	Specify which field (column) is used for the condition as a Long Integer.
Operator	Specify the condition type as a FilterOperator (see Table 202).
IsNumeric	If True, the NumericValue property is used; otherwise the StringValue is used.
NumericValue	Specify a numeric value for the condition as a Double.
StringValue	Specify a String value for the condition.

Table 205. Properties defined by the com.sun.star.sheet.SheetFilterDescriptor service.

Property	Description
IsCaseSensitive	If True, string comparisons are case sensitive.
SkipDuplicates	If True, duplicate entries are not included in the result.
UseRegularExpressions	If True, string values in the TableFilterField structure are interpreted as regular expressions.
SaveOutputPosition	If True (and CopyOutputData is True), the OutputPosition is saved for future calls.
Orientation	Specify if columns or rows are filtered using the TableOrientation enumeration (see Table 196).
ContainsHeader	If True, the first row (or column) is assumed to be a header and is not filtered.
CopyOutputData	If True, the filtered data is copied to the OutputPosition.
OutputPosition	Specify where filtered data is copied as a CellAddress (see Table 163).
MaxFieldCount	The maximum number of filter fields in the descriptor as a Long Integer.

Tables

Each data pilot table is based on a spreadsheet cell range. Each data pilot table also supports the object method `getOutputRange()`, which returns a `CellRangeAddress` (see Table 170). The `refresh()` method re-creates the table using the current data in the source range. Each data pilot table also supports the `DataPilotDescriptor` service, which defines the methods in Table 206.

Table 206. Methods defined by the com.sun.star.sheet.XDataPilotDescriptor interface.

Method	Description
getTag()	Get the data pilot table tag as a String.
setTag(String)	Set the data pilot table tag.
getSourceRange()	Return the CellRangeAddress (see Table 170) containing the data for the data pilot table.
setSourceRange(CellRangeAddress)	Set the cell range containing the data for the data pilot table.
getFilterDescriptor()	Get the SheetFilterDescriptor (see Table 205) that specifies which data from the source cell range is used for the data pilot table.
getDataPilotFields()	Get the data pilot fields as an object that supports indexed access.
getColumnFields()	Get the data pilot fields used as column fields as an object that supports indexed access.
getRowFields()	Get the data pilot fields used as row fields as an object that supports indexed access.
getPageFields()	Get the data pilot fields used as page fields as an object that supports indexed access.
getDataFields()	Get the data pilot fields used as data fields as an object that supports indexed access.
getHiddenFields()	Get the data pilot fields that are not used as column, row, page, or data fields.

15.5.7. Sheet cursors

In a Calc document, a cursor is a cell range that contains methods to move through the contained cells. Cursors are not used as frequently with Calc documents as with Writer documents because, unlike Writer documents, most content is directly accessible by index or name. Sheet cursors, like cell ranges, are limited to one sheet at a time. The SheetCellCursor service used by Calc documents is similar to cell cursors used in text tables (see Table 207).

Table 207. Primary components supported by the com.sun.star.sheet.SheetCellCursor service.

Component	Description
com.sun.star.table.CellCursor	Methods to control the position of a cell cursor.
com.sun.star.table.CellRange	Methods to access cells or subranges of a cell range (see Table 177).
com.sun.star.sheet.XSheetCellCursor	Advanced methods to control the position of the cursor.
com.sun.star.sheet.SheetCellRange	A rectangular range of cells in a spreadsheet document; this is an extension of the CellRange service for use in spreadsheet documents.
com.sun.star.sheet.XUsedAreaCursor	Methods to find the used area in a sheet.

The primary methods supported by the SheetCellCursor are shown in Table 208.

Table 208. Primary methods supported by the *com.sun.star.sheet.SheetCellCursor* service.

Interface	Method	Description
XCellCursor	gotoStart()	Move the cursor to the first filled cell at the beginning of a contiguous series of filled cells. This cell may be outside the cursor's range.
XCellCursor	gotoEnd()	Move the cursor to the last filled cell at the end of a contiguous series of filled cells. This cell may be outside the cursor's range.
XCellCursor	gotoNext()	Move the cursor to the next (right) unprotected cell.
XCellCursor	gotoPrevious()	Move the cursor to the previous (left) unprotected cell.
XCellCursor	gotoOffset(nCol, nRow)	Shift the cursor's range relative to the current position. Negative numbers shift left and up; positive numbers shift right and down.
XCellRange	getCellByPosition(left, top)	Get a cell within the range.
XCellRange	getCellRangeByPosition(left, top, right, bottom)	Get a cell range within the range.
XCellRange	getCellRangeByName(name)	Get a cell range within the range based on its name. The string directly references cells using the standard formats — such as “B2:D5” or “\$B\$2” — or defined cell range names.
XSheetCellCursor	collapseToCurrentRegion()	Expand the range to contain all contiguous nonempty cells.
XSheetCellCursor	collapseToCurrentArray()	Expand the range to contain the current array formula.
XSheetCellCursor	collapseToMergedArea()	Expand the range to contain merged cells that intersect the range.
XSheetCellCursor	expandToEntireColumns()	Expand the range to contain all columns that intersect the range.
XSheetCellCursor	expandToEntireRows()	Expand the range to contain all rows that intersect the range.
XSheetCellCursor	collapseToSize(nCols, nRows)	Without changing the upper-left corner, set the cursor range size.
XSheetCellRange	getSpreadsheet()	Get the sheet object that contains the cell range.
XUsedAreaCursor	gotoStartOfUsedArea()	Set the cursor to the start of the used area.
XUsedAreaCursor	gotoEndOfUsedArea()	Set the cursor to the end of the used area.

Cell ranges, and therefore cell cursors, deal with rectangular regions. The use of rectangular regions may be obvious now that I state it, but it caught me by surprise when I tested the `gotoStart()` and `gotoEnd()` methods listed in Table 208. I started with the configuration shown in Figure 100 when writing the code in Listing 444.

Listing 444. Simple cursor movement commands use contiguous blocks.

```
oCurs = oSheet.createCursorByRange(oSheet.getCellRangeByName("C3"))
oCurs.gotoStart() REM Move the cursor to cell B1
oCurs.gotoEnd()   REM Move the cursor to cell E8
oCurs.gotoStart() REM Move the cursor to cell E8
```

The first line in Listing 444 positions the cursor at cell C3, right in the middle of a block of values. In Figure 100, the leftmost contiguous column is B and the topmost contiguous row is 1. The method `gotoStart()`, therefore, positions the cursor in the top-left corner at location B1. This is where things become a little bit unexpected. The rightmost contiguous column is E and the bottommost contiguous row is 8. The method `gotoEnd()`, therefore, positions the cursor at

location E8. As it is shown in Figure 100, the cell E8 is completely disconnected from the contiguous group of cells. The cursor is positioned to cell E8, even if it does not contain a value. The cursor is no longer related to the original block of cells, so the method `gotoStart()` does not move the cursor back to cell B1.

To understand the behavior of Listing 444, it's important to understand how OOo determines contiguous cells, because it isn't documented anywhere that I can find. Experimentally, I have determined that the set of contiguous nonempty cells is defined as the smallest range (square block of cells) that can be enclosed by empty cells. If cell E9 contained a value, then even though cells E8 and E9 are not directly connected via nonempty cells to the original block of cells, they would both be considered part of the block of contiguous nonempty cells.

The method `collapseToCurrentRegion()` causes the cursor to contain the block of contiguous cells. The only caveat is that after the range is collapsed, it always contains the original range, even if this range includes unnecessary empty cells. The method `collapseToCurrentArray()` is similar to `collapseToCurrentRegion()`, except that it returns a range that contains an array formula. The upper-left corner of the region must include an array formula for the `collapseToCurrentArray()` method to work.

The code snippet in Listing 445 creates a cursor over a range and then demonstrates that `getCellByPosition()` and `getCellRangeByPosition()` are relative to the upper-left corner of the range. The method `getCellRangeByName()` generates an exception if a cell outside the range is requested.

Listing 445. *Some commands work only relative to the range.*

```
oCurs = oSheet.createCursorByRange(oSheet.getCellRangeByName("C3:F12"))
oCell = oCurs.getCellByPosition(0, 0) REM Cell C3
oRange = oCurs.getCellRangeByPosition(1, 0, 3, 2) REM D3:F5
oRange = oCurs.getCellRangeByName("C4:D6") REM C4:D6
oRange = oCurs.getCellRangeByName("C2:D6") REM Error C2 not in range!
```

Tip

The methods `getCellByPosition()`, `getCellRangeByPosition()`, and `getCellRangeByName()` cannot return a value that is not in the range.

15.6. Calc documents

Many of the document-level methods and properties affect the entire document — for example, the ability to save and print a document. Other methods and properties exist purely as a convenience and the information is also available at the sheet level. For example, the Calc document acts as a draw-page supplier to access all of the draw pages, even though they are available individually from the sheet that contains them.

15.6.1. Named range

The official definition of a named range is a named formula expression. Typically, a named range represents a cell range, but it may also refer to external data. Naming ranges allows you to give

meaningful names to things that you reference. Named ranges may therefore be used as an address in a formula; for example, after defining a range named Scores, I can use the formula =Sum(Scores). The NamedRangeFlag constants define how a named range may be used (see Table 209).

Table 209. Constants defined by the com.sun.star.sheet.NamedRangeFlag constant group.

Value	Name	Description
1	FILTER_CRITERIA	The range contains filter criteria.
2	PRINT_AREA	The range can be used as a print range.
4	COLUMN_HEADER	The range can be used as column headers for printing.
8	ROW_HEADER	The range can be used as row headers for printing.

Each named range service supports the methods in Table 210.

Table 210. Methods implemented by the com.sun.star.sheet.NamedRange service.

Method	Description
getReferredCells()	Get the CellRange referenced by named range.
getContent()	The named range content is a string and can be a reference to a cell, cell range, or formula expression.
setContent(String)	Set the content of the named range.
getReferencePosition()	Get the CellAddress used as a base for relative references in the content.
setReferencePosition(CellAddress)	Set the reference position.
getType()	Get the type as a NamedRangeFlag constant (see Table 209).
setType(NamedRangeFlag)	Set the type of the named range.

The document's NamedRanges property contains the collection of named ranges in the document. You can extract each individual named range by using named and indexed access.

The method addNewByname() accepts four arguments; the name, content, position, and type. The fourth argument to the method addNewByName() is a combination of flags that specify how the named range will be used — the most common value is 0, which is not a defined constant value. The name and the content are both of type String. The CellAddress specifies the base address for relative cell references.

Listing 446. *Create a named range that references \$Sheet1.\$B\$3:\$D\$6.*

```
Sub AddNamedRange()
    Dim oRange      ' The created range.
    Dim oRanges      ' All named ranges.
    Dim sName$       ' Name of the named range to create.
    Dim oCell        ' Cell object.
    Dim s$

    sName$ = "MyNRange"
    oRanges = ThisComponent.NamedRanges
```

```

If NOT oRanges.HasByName(sName$) Then
    REM I can obtain the cell address by obtaining the cell
    REM and then extracting the address from the cell.
    Dim oCellAddress As New com.sun.star.table.CellAddress
    oCellAddress.Sheet = 0      'The first sheet.
    oCellAddress.Column = 1    'Column B.
    oCellAddress.Row = 2       'Row 3.

    REM The first argument is the range name.
    REM The second argument is formula or expression to
    REM use. The second argument is usually a string that
    REM defines a range.
    REM The third argument specifies the base address for
    REM relative cell references.
    REM The fourth argument is a set of flags that define
    REM how the range is used, but most ranges use 0.
    REM The fourth argument uses values from the
    REM NamedRangeFlag constants.
    s$ = "$Sheet1.$B$3:$D$6"
    oRanges.AddNewByName(sName$, s$, oCellAddress, 0)
End If

REM Get a range using the created named range.
oRange = ThisComponent.NamedRanges.GetByName(sName$)

REM Print the string contained in cell $Sheet1.$B$3
oCell = oRange.GetReferredCells().GetCellByPosition(0,0)
Print oCell.GetString()
End Sub

```

The third argument, a cell address, acts as the base address for cells referenced in a relative way. If the cell range is not specified as an absolute address, the referenced range will be different based on where in the spreadsheet the range is used. The relative behavior is illustrated in Listing 447, which also illustrates another usage of a named range—defining an equation. The macro creates the named range `AddLeft`, which refers to the equation `A3+B3` with `C3` as the reference cell. The cells `A3` and `B3` are the two cells directly to the left of `C3`, so, the equation `=AddLeft` calculates the sum of the two cells directly to the left of the cell that contains the equation. Changing the reference cell to `C4`, which is below `A3` and `B3`, causes the `AddLeft` equation to calculate the sum of the two cells that are to the left on the previous row.

Listing 447. Create the `AddLeft` named range.

```

Sub AddNamedFunction()
    Dim oSheet      'Sheet that contains the named range.
    Dim oCellAddress 'Address for relative references.
    Dim oRanges     'The NamedRanges property.
    Dim oRange      'Single cell range.
    Dim sName As String 'Name of the equation to create.

    sName = "AddLeft"
    oRanges = ThisComponent.NamedRanges
    If NOT oRanges.HasByName(sName) Then
        oSheet = ThisComponent.getSheets().getByIndex(0)
    End If
End Sub

```

```

oRange = oSheet.getCellRangeByName("C3")
oCellAddress = oRange.getCellAddress()
oRanges.addNewByName(sName, "A3+B3", oCellAddress, 0)
End If
End Sub

```

Use the method `addNewFromTitles(CellRangeAddress, Border)` to create named cell ranges from titles in a cell range. The `Border` value is taken from Table 211 and it specifies where the titles are located in the cell range.

Table 211. Enumerated values defined by the `com.sun.star.sheet.Border` enumeration.

Value	Description
<code>com.sun.star.sheet.Border.TOP</code>	Select the top border.
<code>com.sun.star.sheet.Border.BOTTOM</code>	Select the bottom border.
<code>com.sun.star.sheet.Border.RIGHT</code>	Select the right border.
<code>com.sun.star.sheet.Border.LEFT</code>	Select the left border.

The next macro creates three named ranges based on the top row of a cell range.

Listing 448. Create many named ranges.

```

Sub AddManyNamedRanges()
    Dim oSheet      'Sheet that contains the named range.
    Dim oAddress     'Range address.
    Dim oRanges     'The NamedRanges property.
    Dim oRange       'Single cell range.

    oRanges = ThisComponent.NamedRanges
    oSheet = ThisComponent.getSheets().getByIndex(0)
    oRange = oSheet.getCellRangeByName("A1:C20")
    oAddress = oRange.getRangeAddress()
    oRanges.addNewFromTitles(oAddress, com.sun.star.sheet.Border.TOP)
End Sub

```

Use the method `outputList(CellAddress)` to write a list of the named ranges to a sheet. The first column contains the name of each named range, and the second column contains the content. Finally, use the `removeByName(name)` method to remove a named range.

15.6.2. Database range

Although a database range can be used as a regular named range, a database range also defines a range of cells in a spreadsheet to be used as a database. Each row in a range corresponds to a record and each cell corresponds to a field. You can sort, group, search, and perform calculations on the range as if it were a database.

A database range provides behavior that is useful when performing database related activities. For example, you can mark the first row as headings. To create, modify, or delete a database range, use **Data > Define Range** to open the Define Data Range dialog.

In a macro, a database range is accessed, created, and deleted from the DatabaseRanges property. The following macro creates a database range named “MyName” and sets the range to be used as an auto filter.

Listing 449. Create a database range and an auto filter.

```
Sub AddNewDatabaseRange()  
    Dim oRange 'DatabaseRange object.  
    Dim oAddr 'Cell address range for the database range.  
    Dim oSheet 'First sheet, which will contain the range.  
    Dim oDoc 'Reference ThisComponent with a shorter name.  
  
    oDoc = ThisComponent  
    If NOT oDoc.DatabaseRanges.hasByName("MyName") Then  
        oSheet = ThisComponent.getSheets().getByIndex(0)  
        oRange = oSheet.getCellRangeByName("A1:F10")  
        oAddr = oRange.getRangeAddress()  
        oDoc.DatabaseRanges.addNewByName("MyName", oAddr)  
    End If  
    oRange = oDoc.DatabaseRanges.getByName("MyName")  
    oRange.AutoFilter = True  
End Sub
```

15.6.3. Filters

Use filters to limit the visible rows in a spreadsheet. Generic filters, common to all sorts of data manipulations, are automatically provided by the auto filter capability. You can also define your own filters.

Listing 450. Create a simple sheet filter.

```
Sub SimpleSheetFilter()  
    Dim oSheet ' Sheet that will contain the filter.  
    Dim oFilterDesc ' Filter descriptor.  
    Dim oFields(0) As New com.sun.star.sheet.TableFilterField  
  
    oSheet = ThisComponent.getSheets().getByIndex(0)  
  
    REM If argument is True, creates an empty filter  
    REM descriptor. If argument is False, create a  
    REM descriptor with the previous settings.  
    oFilterDesc = oSheet.createFilterDescriptor(True)  
  
    With oFields(0)  
        REM I could use the Connection property to indicate  
        REM how to connect to the previous field. This is  
        REM the first field so this is not required.  
        '.Connection = com.sun.star.sheet.FilterConnection.AND  
        '.Connection = com.sun.star.sheet.FilterConnection.OR  
  
        REM The Field property is the zero based column  
        REM number to filter. If you have the cell, you  
        REM can use .Field = oCell.CellAddress.Column.  
        .Field = 5  
    End With  
End Sub
```



```

REM Compare using a numeric or a string?
.IsNumeric = True

REM The NumericValue property is used
REM because .IsNumeric = True from above.
.NumericValue = 80

REM If IsNumeric was False, then the
REM StringValue property would be used.
REM .StringValue = "what ever"

REM Valid operators include EMPTY, NOT_EMPTY, EQUAL,
REM NOT_EQUAL, GREATER, GREATER_EQUAL, LESS,
REM LESS_EQUAL, TOP_VALUES, TOP_PERCENT,
REM BOTTOM_VALUES, and BOTTOM_PERCENT
.Operator = com.sun.star.sheet.FilterOperator.GREATER_EQUAL
End With

REM The filter descriptor supports the following
REM properties: IsCaseSensitive, SkipDuplicates,
REM UseRegularExpressions,
REM SaveOutputPosition, Orientation, ContainsHeader,
REM CopyOutputData, OutputPosition, and MaxFieldCount.
oFilterDesc.setFilterFields(oFields())
oFilterDesc.ContainsHeader = True
oSheet.filter(oFilterDesc)
End Sub

```

When a filter is applied to a sheet, it replaces any existing filter for the sheet. Setting an empty filter in a sheet will therefore remove all filters for that sheet.

Listing 451. Remove the current sheet filter.

```

Sub RemoveSheetFilter()
    Dim oSheet          ' Sheet to filter.
    Dim oFilterDesc      ' Filter descriptor.

    oSheet = ThisComponent.getSheets().getByIndex(0)
    oFilterDesc = oSheet.createFilterDescriptor(True)
    oSheet.filter(oFilterDesc)
End Sub

```

A more advanced filter may filter more than one column.

Listing 452. A simple sheet filter using two columns.

```

Sub SimpleSheetFilter_2()
    Dim oSheet          ' Sheet to filter.
    Dim oRange          ' Range to be filtered.
    Dim oFilterDesc      ' Filter descriptor.
    Dim oFields(1) As New com.sun.star.sheet.TableFilterField

    oSheet = ThisComponent.getSheets().getByIndex(0)
    oRange = oSheet.getCellRangeByName("E12:G19")

```

```

REM If argument is True, creates an
REM empty filter descriptor.
oFilterDesc = oRange.createFilterDescriptor(True)

REM Setup a field to view cells with content that
REM start with the letter b.
With oFields(0)
    .Field = 0          ' Filter column A.
    .IsNumeric = False  ' Use a string, not a number.
    .StringValue = "b.*" ' Everything starting with b.
    .Operator = com.sun.star.sheet.FilterOperator.EQUAL
End With

REM Setup a field that requires both conditions and
REM this new condition requires a value greater or
REM equal to 70.
With oFields(1)
    .Connection = com.sun.star.sheet.FilterConnection.AND
    .Field = 5          ' Filter column F.
    .IsNumeric = True   ' Use a number
    .NumericValue = 70   ' Values greater than or equal to 70
    .Operator = com.sun.star.sheet.FilterOperator.GREATER_EQUAL
End With

oFilterDesc.setFilterFields(oFields())
oFilterDesc.ContainsHeader = False
oFilterDesc.UseRegularExpressions = True
oSheet.filter(oFilterDesc)
End Sub

```

An advanced filter supports up to eight filter conditions, as opposed to the three supported by the simple filter. The criteria for an advanced filter is stored in a sheet. The first step in creating an advanced filter is entering the filter criteria into the spreadsheet.

1. Select an empty space in the Calc document. The empty space may reside in any sheet in any location in the Calc document.
2. Duplicate the column headings from the area to be filtered into the area that will contain the filter criteria.
3. Enter the filter criteria underneath the column headings. The criterion in each column of a row is connected with AND. The criteria from each row are connected with OR.

Applying an advanced filter using a macro is simple. The cell range containing the filter criteria is used to create a filter descriptor, which is then used to filter the range containing the data.

Listing 453. Use an advanced filter.

```

Sub UseAnAdvancedFilter()
    Dim oSheet      'A sheet from the Calc document.
    Dim oRanges      'The NamedRanges property.
    Dim oCritRange   'Range that contains the filter criteria.

```

```

Dim oDataRange 'Range that contains the data to filter.
Dim oFiltDesc 'Filter descriptor.

REM Range that contains the filter criteria
oSheet = ThisComponent.getSheets().getByIndex(1)
oCritRange = oSheet.getCellRangeByName("A1:G3")

REM You can also obtain the range containing the
REM filter criteria from a named range.
REM oRanges = ThisComponent.NamedRanges
REM oRange = oRanges.getByName("AverageLess80")
REM oCritRange = oRange.getReferredCells()

REM The data that I want to filter
oSheet = ThisComponent.getSheets().getByIndex(0)
oDataRange = oSheet.getCellRangeByName("A1:G16")

oFiltDesc = oCritRange.createFilterDescriptorByObject(oDataRange)
oDataRange.filter(oFiltDesc)
End Sub

```

15.6.4. Protecting documents and sheets

Calc documents and individual spreadsheets support the XProtectable interface. Use the methods `protect(password)` and `unprotect(password)` to activate or disable protection. The password is passed in as a String. The `isProtected()` method returns True if protection is currently active.

15.6.5. Controlling recalculation

By default, a Calc document automatically recalculates formulas when the cells to which they refer are modified. At times, it is useful to disable automatic recalculation. The methods in Table 212 allow you to control recalculation in the entire document.

Table 212. Methods defined by the `com.sun.star.sheet.XCalculatable` interface.

Method	Description
<code>calculate()</code>	Recalculate all cells with changed content.
<code>calculateAll()</code>	Recalculate all cells.
<code>isAutomaticCalculationEnabled()</code>	True if automatic calculation is enabled.
<code>enableAutomaticCalculation(Boolean)</code>	Enable or disable automatic calculation.

15.6.6. Using Goal Seek

“Goal Seek” attempts to solve equations with one unknown variable. In other words, after defining a formula with multiple fixed values and one variable value, Goal Seek tries to find an acceptable value for the unknown variable.

Consider a very simple example. If you jump off a cliff, gravity will accelerate you toward the ground at 32 feet per second each second. In other words, in one second you will be traveling at 32

feet per second and in two seconds you will be traveling at 64 feet per second. The equation is given as “velocity = acceleration * time”. I have a constant value for the acceleration due to gravity, and I want to know how long before I am traveling at 100 feet per second. Admittedly, this is a trivial example, but it is easy to understand.

Use the document method `seekGoal(CellAddress, CellAddress, String)` to perform a Goal Seek operation. The first cell address identifies the cell that contains the formula to solve. The second cell address identifies the cell that contains the variable that will change. Place the best guess that you can make into this cell. The final string contains the value that you want to obtain from the formula. The macro in Listing 454 sets the formula and then performs a Goal Seek operation.

Listing 454. Set a simple goal seek.

```
Sub GoalSeekExample
    Dim oSheet 'oSheet will contain the first sheet.
    Dim oGCell 'B24, gravity value of 32
    Dim oTCell 'C24, time value
    Dim oRCell 'Resulting equation "=B24*C24"
    Dim oGoal 'Returned goal object
    oSheet = ThisComponent.Sheets(0)
    oGCell = oSheet.getCellByPosition(1, 23)
    oGCell.setValue(32)

    oTCell = oSheet.getCellByPosition(2, 23)
    oTCell.setValue(1)

    oRCell = oSheet.getCellByPosition(0, 23)
    oRCell.setFormula("=B24 * C24")

    oGoal=ThisComponent.seekGoal(oRCell.CellAddress, oTCell.CellAddress, "100")
    MsgBox "Result = " & oGoal.Result & CHR$(10) & _
        "The result changed by " & oGoal.Divergence & _
        " in the last iteration", 0, "Goal Seek"
End Sub
```

The `seekGoal()` method returns a structure containing two floating-point Double values. The `Result` property identifies the proposed solution. The `Divergence` property identifies the difference between the last guessed result and the current result. If the divergence is small, the result is probably reasonably accurate. If, however, the divergence is large, the result is probably inaccurate. I performed a test where there was no solution. The divergence value was roughly 1.0E308.

Do not let the simplicity of the provided example fool you. I used the `seekGoal()` method to provide a solution to a problem that had no closed form solution — I had to use a numerical algorithm to solve the problem.

15.7. Write your own Calc functions

With OOo, it's trivial to write and use your own functions that are recognized by the Calc document. It's as simple as writing a macro and then addressing it directly. I wrote an example of

this in Listing 455 that returns information about the passed argument as a String. Table 213 shows the return values for different arguments.

Listing 455. Print information about the argument.

```
Function WahooFunc(Optional x) As Variant
    Dim s$
    s = "I am in WahooFunc. "
    If IsMissing(x) Then
        s = s & "No argument was passed"
    ElseIf NOT IsArray(x) Then
        s = s & "Scalar argument (" & CStr(x) & ") is type " & TypeName(x)
    Else
        s = s & "Argument is an array (" & LBound(x, 1) & " To " & UBound(x, 1) & _
            ", " & LBound(x, 2) & " To " & UBound(x, 2) & ")"
    End If
    WahooFunc = s
End Function
```

Table 213. Return values for different arguments to WahooFunc (assume cell E9 contains 2).

Function	Return
"=WahooFunc()"	I am in WahooFunc. No argument was passed.
"=WahooFunc(E9)"	I am in WahooFunc. Scalar argument (2) is type Double.
"=WahooFunc(2)"	I am in WahooFunc. Scalar argument (2) is type Double.
"=WahooFunc(A11:C15)"	I am in WahooFunc. Argument is an array (1 To 5, 1 To 3).

The argument that is passed to the function that you write, may be missing. If the argument is declared as optional, use the IsMissing() method to test if a value has been passed. For example, "=WahooFunc()".

You can directly pass a single scalar value either as a constant, or by referencing a single cell. For example, "=WahooFunc(32)" and "=WahooFunc(E7)" both pass a scalar value to the function. The first example passes the numeric value 32 and the second example passes the contents of cell E7.

Tip

The actual type of argument that is passed to your own functions depends on how it is called (see Table 213). When a range is used as the argument, the data is passed as a two-dimensional array that does not use the traditional lower bound of zero.

If the argument refers to a range, a two-dimensional array is passed to the function. For example, "=WahooFunc(E7:F32)" passes the contents of the cells in the range E7 through F32 as a two-dimensional array. Be absolutely certain to use the functions LBound() and UBound(), because the lower bounds start at 1 rather than at the expected value of 0 (see Listing 456).

Listing 456. Add all elements together.

```
Function SumAll(myArray as Variant)
    Dim iRow%, iCol%
    Dim d As Double
    For iRow = LBound(myArray, 1) To UBound(myArray, 1)
```

```

    For iCol = LBound(myArray, 2) To UBound(myArray, 2)
        d = d + myArray(iRow, iCol)
    Next
Next
SumAll = d
End Function

```

The Calc function must be visible / available to the Calc document, so I store my Calc functions in the Standard library of the document that will use the functions. Sometimes, when I write a new function, the new function is not noticed until the document has been saved, closed, and then reopened.

Tip If Calc does not see a newly created function, save, close, and reopen the Calc document.

Warn Do not return a time or date in a Calc function, return a double instead.

Macros used as Calc functions have their values converted to a double or a string before they are stored in a cell. The date type is converted to a string, which is not useful for date calculations. The proper way to return a date and/or time is to convert the value to a Double and return the double. Calc will treat the value as a date / time based on the cell formatting.

15.8. Using the current controller

Each OOo document contains a controller that interacts with the user. Therefore, the current controller knows what text is selected, the location of the current cursor, and which sheet is active.

15.8.1. Selected cells

The document's controller interacts with the user and therefore knows which cells are selected. In a Calc document, a selected cell can be a few different things. The following cases are numbered to simply reference each case; the numbering serves no other purpose.

1. One cell selected. To select an entire cell, click in a cell once and then hold down the Shift key and click in the cell again.
2. Partial text in a single cell selected. Double-click in a single cell and then select some text.
3. Nothing appears selected. Single-click in a cell or tab between cells.
4. Multiple cells selected. Single-click in a cell and then drag the cursor.
5. Multiple disjointed selections. Select some cells. Hold down the Ctrl key and select some more.

So far, I have not been able to distinguish between the first three cases; they all look like one cell is selected. If only one cell is selected, the current selection returned by the current controller is the sheet cell containing the cursor. If a single cell is selected (cases 1 through 3), the selections object supports the SheetCell service (see Listing 457).

If multiple cells are selected as a single range (case 4), the selections object is a SheetCellRange. Check to see if the selections object supports the SheetCellRanges service. If so, then more than one cell range is selected. Use the getCount() method of the selections object to see how many ranges are selected.

Listing 457. Determine if anything is selected.

```
Function CalcIsAnythingSelected(oDoc) As Boolean
    Dim oSelections

    REM Assume that nothing is selected.
    CalcIsAnythingSelected = False
    If IsNull(oDoc) Then Exit Function
    REM The current selection in the current controller.
    REM If there is no current controller, this returns NULL.
    oSelections = oDoc.getCurrentSelection()
    If IsNull(oSelections) Then Exit Function

    If oSelections.supportsService("com.sun.star.sheet.SheetCell") Then
        Print "One Cell selected = " & oSelections.getImplementationName()
        MsgBox "getString() = " & oSelections.getString()
    ElseIf oSelections.supportsService("com.sun.star.sheet.SheetCellRange") Then
        Print "One Cell Range selected = " & oSelections.getImplementationName()
    ElseIf oSelections.supportsService("com.sun.star.sheet.SheetCellRanges") Then
        Print "Multiple Cell Ranges selected = " & _
            oSelections.getImplementationName()
        Print "Count = " & oSelections.getCount()
    Else
        Print "Something else selected = " & oSelections.getImplementationName()
    End If
    CalcIsAnythingSelected = True
End Function
```

Enumerating the selected cells

Listing 458 is a utility routine that sets the text of the cells in a range to a specific value. Although Listing 458 is specifically designed to operate on a cell range, it uses methods that are also supported by a single cell. The macro, therefore, may be used by a cell or a cell range object. The method used in Listing 458 is a modification of an algorithm introduced to me by Sasa Kelecevic, on the OOO dev mailing list.

Listing 458. Set all text in a range to a value.

```
Sub SetRangeText(oRange, s As String)
    Dim nCol As Long 'Column index variable
    Dim nRow As Long 'Row index variable
    Dim oCols        'Columns in the selected range
    Dim oRows        'Rows in the selected range

    oCols = oRange.Columns : oRows = oRange.Rows
    For nCol = 0 To oCols.getCount() - 1
        For nRow = 0 To oRows.getCount() - 1
            oRange.getCellByPosition(nCol, nRow).setString(s)
        
```

```

        Next
    Next
End Sub

```

To demonstrate how to inspect all of the selected cells, the macro in Listing 459 sets the text in every selected cell to a specific text value.

Listing 459. *Set all selected cells to a value.*

```

Sub SetSelectedCells(s As String)
    Dim oSelections 'The selections object
    Dim oCell       'If one cell is selected
    Dim oRanges     'If multiple ranges are selected, use this
    Dim i As Long   'General index variable

    REM The current selection in the current controller.
    REM If there is no current controller, this returns NULL.
    oSelections = ThisComponent.getCurrentSelection()
    If IsNull(oSelections) Then Exit Sub

    If oSelections.supportsService("com.sun.star.sheet.SheetCell") Then
        oCell = oSelections
        oCell.setString(s)

        REM For consistency, I could use the same call for a range as a cell
        REM but this is only because a cell can also return the columns and rows.
        REM SetRangeText(oSelections, s)
    ElseIf oSelections.supportsService("com.sun.star.sheet.SheetCellRange") Then
        SetRangeText(oSelections, s)
    ElseIf oSelections.supportsService("com.sun.star.sheet.SheetCellRanges") Then
        oRanges = oSelections
        For i = 0 To oRanges.getCount() - 1
            SetRangeText(oRanges.getByIndex(i), s)
        Next
    Else
        Print "Something else selected = " & oSelections.getImplementationName()
    End If
End Sub

```

While writing the macro in Listing 459, I experienced some initially perplexing behavior. My initial version of Listing 459 would sometimes generate an error as it attempted to enter the subroutine, depending on the selected values. The run-time error would complain that an unsupported property or invalid value was used. To explain the problem (and the solution), notice that there are two places in the code where I assign `oSelections` to a temporary variable and then I use the variable. Listing 460 contains the small sections of code extracted from Listing 459.

Listing 460. *I assign `oSelections` to a temporary variable before use.*

```

oCell = oSelections
oCell.setString(s)
.....
oRanges = oSelections
For i = 0 To oRanges.getCount() - 1
    SetRangeText(oRanges.getByIndex(i), s)

```


Next

If oSelections refers to a cell, it doesn't support the object method getCount(). If oSelections refers to a SheetCellRanges object, it doesn't support the setString() object method. I can but assume that the Basic interpreter attempts to resolve the properties and methods that an object references when it is assigned a value. For example, if the selections object is a cell, an error is raised because a cell doesn't support the getCount() method. On the other hand, if more than one cell is selected, the returned object does not support the setString() method. Although this is not the first time that I have experienced this problem, it is the first time that I determined the nature of the problem and how to avoid it.

Selecting text

The current controller is used to determine the current selection, and it can also be used to set the current selection. Use the current controller's select(obj) method to select cells in a sheet. The documentation essentially says that if the controller recognizes and is able to select the object passed as an argument, it will. Listing 461 demonstrates how to select cells.

Listing 461. Select B28:D33 and then select cell A1 instead.

```
Dim oSheet, oRange, oCell, oController
oController = ThisComponent.getCurrentController()
oSheet = ThisComponent.Sheets(3)
oRange = oSheet.getCellRangeByName("B28:D33")
oController.select(oRange)
oCell = oSheet.getCellByPosition(0, 0)
oController.select(oCell)
```

The active cell

The active cell contains the cursor. There is an active cell even when multiple cells are selected. Unfortunately, OOO does not provide a method to return the active cell when more than one cell is selected. Paolo Mantovani posted a very nice solution to this problem on the dev mailing list as shown in Listing 462. The disadvantage to the macro in Listing 462 is that the active cell is no longer active after the macro runs.

Listing 462. Obtain the active cell.

```
REM Author: Paolo Mantovani
REM email: mantovani.paolo@tin.it
Sub RetrieveTheActiveCell()
    Dim oOldSelection 'The original selection of cell ranges
    Dim oRanges       'A blank range created by the document
    Dim oActiveCell   'The current active cell
    Dim oConv         'The cell address conversion service
    Dim oDoc
    oDoc = ThisComponent

    REM store the current selection
    oOldSelection = oDoc.CurrentSelection

    REM Create an empty SheetCellRanges service and then select it.
```

```

REM This leaves ONLY the active cell selected.
oRanges = oDoc.CreateInstance("com.sun.star.sheet.SheetCellRanges")
oDoc.CurrentController.Select(oRanges)

REM Get the active cell!
oActiveCell = oDoc.CurrentSelection

oConv = oDoc.CreateInstance("com.sun.star.table.CellAddressConversion")
oConv.Address = oActiveCell.getCellAddress
Print oConv.UserInterfaceRepresentation
print oConv.PersistentRepresentation

REM Restore the old selection, but lose the previously active cell
oDoc.CurrentController.Select(oOldSelection)
End Sub

```

15.8.2. General functionality

When searching for view-related functionality, the current controller is a good place to start. Table 214 and Table 215 contain most of the methods and properties supported by the current controller that have not already been discussed.

Table 214. *Methods supported by the current controller not already discussed.*

Methods	Description
getActiveSheet()	Get the active spreadsheet.
setActiveSheet(XSpreadsheet)	Cause the specified sheet to become active.
getIsWindowSplit()	Return True if the view is split.
getSplitHorizontal()	Long Integer horizontal split position (in pixels).
getSplitVertical()	Long Integer vertical split position (in pixels).
getSplitColumn()	Long Integer column before which the view is split.
getSplitRow()	Long Integer row before which the view is split.
splitAtPosition(x, y)	Split the view at the specified position. If x=0, the split is only horizontal. If y=0, the split is only vertical.
hasFrozenPanels()	True if the view contains frozen panes.
freezeAtPosition(nCol, nRow)	Freeze panes with the specified number of columns and rows.
getFirstVisibleColumn()	Get the first visible column in the pane as a Long Integer.
setFirstVisibleColumn(Long)	Set the first visible column in the pane.
getFirstVisibleRow()	Get the first visible row in the pane as a Long Integer.
setFirstVisibleRow(Long)	Set the first visible row in the pane.
getVisibleRange()	Get the visible range in the pane as a CellRangeAddress.

Table 215. Properties supported by the current controller not already discussed.

Property	Description
ShowFormulas	If True, formulas are displayed instead of their results.
ShowZeroValues	If True, zero values are visible.
IsValueHighlightingEnabled	If True, strings, values, and formulas are displayed in different colors.
ShowNotes	If True, a marker is shown for notes in cells.
HasVerticalScrollBar	If True, a vertical scroll bar is used in the view.
HasHorizontalScrollBar	If True, a horizontal scroll bar is used in the view.
HasSheetTabs	If True, sheet tabs are used in the view.
IsOutlineSymbolsSet	If True, outline symbols are displayed.
HasColumnRowHeaders	If True, column and row headers are visible.
ShowGrid	If True, cell grid lines are displayed.
GridColor	Grid color as a Long Integer.
ShowHelpLines	If True, help lines are displayed while moving drawing objects.
ShowAnchor	If True, anchor symbols are displayed when drawing objects are selected.
ShowPageBreaks	If True, page breaks are displayed.
SolidHandles	If True, solid (colored) handles are displayed when drawing objects are selected.
ShowObjects	If True, embedded objects are displayed.
ShowCharts	If True, charts are displayed.
ShowDrawing	If True, drawing objects are displayed.
HideSpellMarks	If True, spelling marks are not displayed; this seems to be disabled.
ZoomType	Document zoom type as a com.sun.star.view.DocumentZoomType with the following values: • OPTIMAL = 0 – Fit the current page content width (no margins). • PAGE_WIDTH = 1 – Fit the page width at the current selection. • ENTIRE_PAGE = 2 – Fit an entire page. • BY_VALUE = 3 – The zoom is relative and set by ZoomValue. • PAGE_WIDTH_EXACT = 4 – Fit the current width and fit exactly to page end.
ZoomValue	Zoom value if the ZoomType is set to BY_VALUE . ZoomType must be set before the ZoomValue is set.

15.9. Control Calc from Microsoft Office

It turns out that you can control OOo from within the Microsoft Office family of products. The trick is to create a service manager, which starts OOo if it is not currently running. Accessing OOo documents from Microsoft Office is similar to accessing OOo documents using other non-StarBasic languages. Basic provides nice shortcuts that are not available from Microsoft Office. For example, in Basic, when I want the third sheet, I simply use `oDoc.Sheets(2)`; in Microsoft

Office, however, you cannot access the Sheets property as an array. I wrote and ran the macro in Listing 463 from Microsoft Visual Basic from Microsoft Excel.

Listing 463. *ControlOOo() demonstrates how to control OOo from Excel.*

```
Sub ControlOOo()  
    Rem The service manager is always the first thing to create.  
    Rem If OOo is not running, it is started.  
    Set oManager = CreateObject("com.sun.star.ServiceManager")  
  
    Rem Create the desktop.  
    Set oDesktop = oManager.CreateInstance("com.sun.star.frame.Desktop")  
  
    Rem Open a new empty Calc document.  
    Dim args()  
    Dim s As String  
    Set s = "private:factory/scalc"  
    Set oDoc = oDesktop.loadComponentFromURL(s, "_blank", 0, args())  
  
    Dim oSheet As Object  
    Dim oSheets As Object  
    Dim oCell As Object  
  
    Set oSheets = oDoc.sheets.CreateEnumeration  
    Set oSheet = oSheets.nextElement  
    Set oCell = oSheet.getCellByPosition(0, 0)  
    oCell.setFormula ("Hello From Excel") 'Cell A1  
    oCell.CellBackColor = RGB(127, 127, 127)  
End Sub
```

15.10. Accessing Calc functions

You can call Calc functions from a macro.

Listing 464. *Call the MIN function directly.*

```
Sub callFunction  
    Dim oFA  
    oFA = createUnoService( "com.sun.star.sheet.FunctionAccess" )  
  
    ' Calculate min of numbers.  
    print oFA.callFunction( "MIN", array( 10, 23, 5, 345 ) )  
End Sub
```

15.11. Finding URLs in Calc

In a Calc document, links (URLs) are stored in text fields. The following macro enumerates the text content in cell A1 of the first sheet for URLs. Note that the presentation and the URL may be different.

Listing 465. *Finds URLs in a sheet cell.*

```
Sub FindHyperLinkInCell  
    Dim oCell, oText, oParEnum, oParElement  
    Dim oEnum, oElement
```

```

oCell = ThisComponent.Sheets(0).getCellByPosition(0, 0)
oParEnum = oCell.getText().createEnumeration()
Do While oParEnum.hasMoreElements ()
    oParElement = oParEnum.nextElement()
    oEnum = oParElement.createEnumeration()
    Do While oEnum.hasMoreElements ()
        oElement = oEnum.nextElement()
        If oElement.TextPortionType = "TextField" Then
            If oElement.TextField.supportsService("com.sun.star.text.TextField.URL") Then
                'STRING Representation =
                'STRING TargetFrame =
                'STRING URL =
                Print oElement.TextField.URL
            End If
        End If
    End If
Loop
Loop
End Sub

```

15.12. Importing and Exporting XML Files Using Calc

This section, originally written by Volker Lenhardt, discusses simple methods for importing and exporting Extensible Markup Language (XML) files. The ideas and code are taken from a working project.

XML is a formal description of tags and how tags can be combined to form a hierarchical data structure. You can choose the tag and attributes names as you like – within limits. HTML is XML with fewer rules – so valid HTML need not be valid XML. Wikipedia is a good place to learn more about XML.

Warn

Some methods caused OOo version 3.3 to crash; for example, `getElementsByTagName` crashed for some tag names. The methods functioned correctly with LibreOffice, I have not checked them against AOO.

XML files are simple text files. If you need to regularly convert XML to Calc or Base you could write an XSLT filter, which is not covered in this book.

Most modern programming languages have implemented two tools to import XML data: SAX (Simple API for XML) and DOM (Document Object Model). UNO defines `com.sun.star.xml.sax.XParser` for SAX processing and `com.sun.star.xml.dom.DocumentBuilder` for DOM processing. The main difference between these two tools is the way that they read the data from the file. SAX reads the data continually, the Basic code implements a listener to handle the data as it is read. DOM, on the other hand, reads all of the data into an in memory hierarchically structured object. DOM is fast, but uses more memory, SAX can easily handle large files. This section covers DOM, not SAX. The coverage ignores complex topics such as validation, name spaces, and exceptions.

Tip

It is very important that you are acquainted with the data structure of the XML file in question.

15.12.1. Read an XML file

The `com.sun.star.xml.dom.DocumentBuilder` service has two parse methods that return the entire data tree as a DOM object. The first parse method accepts an input stream, which may come from anywhere; for a simple text file, `SimpleFileAccess` can return an input stream. The second parse method accepts the URL of the file (see Listing 466).

Listing 466. *Read an XML file.*

```
REM Build a DOM Document tree from an XML file.
Function BuildDOMDoc(sUrl As String)
    Dim oDocBuilder
    Dim oDOM
    BuildDOMDoc = oDOM

    On Error Goto Catch

    oDocBuilder = createUnoService("com.sun.star.xml.dom.DocumentBuilder")

    REM oDocBuilder reads the complete XML file and returns an object
    REM containing a tree structure of all "nodes".
    oDOM = oDocBuilder.parseURI(sURL)

    REM There were no errors.
    oDOM.normalize()
    BuildDOMDoc = oDOM
    Exit Function

Catch:
    MsgBox "Error while importing the XML data " & CHR$(10) & CHR$(10) & _
        & Error$ & " " & CHR$(10) & " ", 16, "XML Import"
End Function
```

The XML file in Listing 467 is based on a real example. The entire document contains an outer `slots` tag, which represents a database. Each record is called a slot, some are active, some are reserved, some are free. Each slot represents a single client.

Listing 467. *Sample XML file.*

```
<slots>
  <slot status="active" ID="10317">
    <document ID="1234" title="Boom boom" />
    <validTo>12/03/2011</validTo>
    <customer name="Becker" gender="m" ID="11051" email="bb@tennis.de" />
  </slot>
  <slot status="free" ID="60072">
  </slot>
</slots>
```

The problem was to create a mail merge to ask active clients about to expire if they desired to prolong their service. The macro needed to appropriately filter the data based on the status

attribute and ValidTo element, then extract the name, gender, email, ID, and document title. The elements, attributes, and text contents are represented as “nodes” in the DOM tree; the entire document is the root node. The com.sun.star.xml.dom.XNode API documentation describes this primary DOM datatype as follows:

“The Node interface is the primary datatype for the entire Document Object Model. It represents a single node in the document tree. While all objects implementing the Node interface expose methods for dealing with children, not all objects implementing the Node interface may have children. For example, Text nodes may not have children, and adding children to such nodes results in a DOMException being raised.

The “nodeName”, “nodeValue”, and “attributes” properties provide access to commonly used data from a node. The property returns null if there is no obvious mapping of these attributes for a specific nodeType; for example, a comment cannot have attributes. Specific node types may contain additional more convenient mechanisms to get and set relevant information.

Table 216 shows the specialized node interfaces that are derived from XNode together with their types, names, and values. The node type is the corresponding value from the com.sun.star.xml.dom.NodeType enumeration.

Table 216. Specialized Nodes.

Interface	Node Type	Node Name	Node Value
XAttr	ATTRIBUTE_NODE	The name of the attribute	The value of the attribute
XCDATASection	CDATA_SECTION_NODE	"#cdata-section"	The content of the CDATA section
XComment	COMMENT_NODE	"#comment"	The content of the comment
XDocument	DOCUMENT_NODE	"#document"	Null
XDocumentFragment	DOCUMENT_FRAGMENT_NODE	"#document-fragment"	Null
XDocumentType	DOCUMENT_TYPE_NODE	The document type name	Null
XElement	ELEMENT_NODE	The tag name	Null
XEntity	ENTITY_NODE	The entity name	Null
XEntityReference	ENTITY_REFERENCE_NODE	The name of entity referenced	Null
XNotation	NOTATION_NODE	The notation name	Null
XProcessingInstruction	PROCESSING_INSTRUCTION_NODE	The target	The entire content excluding the target
XText	TEXT_NODE	"#text"	The content of the text node

For the methods implemented with XNode see Table 217.

Table 217. Methods implemented in the *com.sun.star.xml.dom.XNode* interface.

Method	Description
<code>appendChild(newChild)</code>	Adds the node <code>newChild</code> to the end of the list of children of this node.
<code>cloneNode(deep As boolean)</code>	Returns a duplicate of this node, i.e., serves as a generic copy constructor for nodes. Parameter <code>deep</code> : True = clone node together with any children, False = clone without children
<code>getAttributes()</code>	Returns a <code>NamedNodeMap</code> containing the attributes of this node (if it is an <code>Element</code>) or null otherwise.
<code>getChildNodes()</code>	Returns a <code>NodeList</code> that contains all children of this node.
<code>getFirstChild()</code>	Returns the first child of this node.
<code>getLastChild()</code>	Returns the last child of this node.
<code>getLocalName()</code>	Returns the local part of the qualified name of this node.
<code>getNamespaceURI()</code>	The namespace URI of this node, or null if it is unspecified.
<code>getNextSibling()</code>	Returns the node immediately following this node.
<code>getNodeName()</code>	The name of this node, depending on its type; see the table above.
<code>getNodeType()</code>	A code representing the type of the underlying object, as defined above.
<code>getNodeValue()</code>	Returns the value of this node.
<code>getOwnerDocument()</code>	The <code>Document</code> object associated with this node.
<code>getParentNode()</code>	The parent of this node.
<code>getPrefix()</code>	The namespace prefix of this node, or null if it is unspecified.
<code>getPreviousSibling()</code>	The node immediately preceding this node.
<code>hasAttributes()</code>	Returns whether this node (if it is an element) has any attributes.
<code>hasChildNodes()</code>	Returns whether this node has any children.
<code>insertBefore(newChild, refChild)</code>	Inserts the node <code>newChild</code> before the existing child node <code>refChild</code> .
<code>isSupported(feature, ver)</code>	Tests whether the DOM implementation implements a specific feature and that feature is supported by this node.
<code>normalize()</code>	Puts all <code>Text</code> nodes in the full depth of the sub-tree underneath this <code>Node</code> , including attribute nodes, into a "normal" form where only structure (e.g., elements, comments, processing instructions, CDATA sections, and entity references) separates <code>Text</code> nodes, i.e., there are neither adjacent <code>Text</code> nodes nor empty <code>Text</code> nodes.
<code>removeChild(oldChild)</code>	Removes the child node indicated by <code>oldChild</code> from the list of children, and returns it.
<code>replaceChild(newChild, oldChild)</code>	Replaces the child node <code>oldChild</code> with <code>newChild</code> in the list of children, and returns the <code>oldChild</code> node.
<code>setNodeValue(nodeValue)</code>	The value of this node, depending on its type; see the table above.
<code>setPrefix(prefix)</code>	The namespace prefix of this node, or null if it is unspecified.

A node list is usually returned as a `com.sun.star.xml.dom.XNodeList` interface. The most used methods from the `XNodeList` interface are `getLength()`, to determine the number of listed nodes, and `item(Long)` to get a single node based on its location in the list. A node list is sometimes returned as a the `com.sun.star.xml.dom.XNamedNodeMap` interface, which provides access to the contained nodes by name. The following macro demonstrates three methods for getting the value of an element attribute.

Listing 468. *Three ways to get an attribute from an element.*

```
'The long way
oAttList = oElement.getAttributes           ' Get all attributes.
oAttNode = oAttList.getNamedItem("ID")     ' Get the attribute node from attributes.
sAttVal = oAttNode.getValue                 ' Get the value from the attribute node.
'A shorter way
oAttNode = oElement.getAttributeNode("ID") ' Get the attribute node from the element.
sAttVal = oAttNode.getValue                 ' Get the value from the attribute node.
'The shortest way
sAttVal = oElement.getAttribute("ID")      ' Get the value directly from the node.
```

Recursion is typically used to parse DOM documents.

```
oElemList = oElem.getChildNodes           'At the very start oElem is oDOM
For i = 0 To oElemList.getLength - 1
    oElem = oElemList.item(i)
    Select Case oElem.getNodeType
        Case com.sun.star.xml.dom.ELEMENT_NODE
            sName = oElem.getNodeName
            If oElem.hasAttributes Then
                oAtts = oElem.getAttributes
                For j = 0 To oAtts.getLength - 1
                    sName = oAtts.item(j).getName
                    sValue = oAtts.item(j).getValue
                    'Further work to be done
                Next
            End If
            If oElem.hasChildNodes Then
                'Start the recursion here
            End If
        Case com.sun.star.xml.dom.TEXT_NODE
            sText = oElem.getNodeValue
            'Further work to be done
        'More Case statements for the node types in Table 216
    End Select
Next
```

TIP	When you encounter a text node between different element nodes, you most probably have found ignorable white space; so you can probably ignore it.

In the slots example, we do not need to iterate over all the nodes because we know the structure and we only need specific nodes. Use `getElementsByTagName(name)` to start with a limited number of elements. Listing 469 shows how to extract the desired data into a Calc sheet. First, the XML file is created, and then it is read into a new Calc document.

Listing 469. *Parse an XML file to extract specific data into a CalcFile..*

```
REM Import an XML file using DOM and extract specific data into a Calc file
Sub XMLImport
    Dim sURL As String           'Whole path of the XML file in URL notation
    Dim oCalcDoc                 'Destination spreadsheet document
    Dim oSheet                   'Destination table
    Dim oWriterDoc               'Text document to show the original XML text file
```

```

Dim oDOM                'The DOM document tree
Dim tDeadline as Date   'A specific date to filter the data

REM Create an example XML file with fancy data
sURL = GetWorkDir() & "/oome_xml_import.xml"
MakeXMLExampleFile(sURL)
Wait 500                'Give the OS time to store the file to disk
tDeadline = DateValue(CDate(Now + 20))

REM Build the DOM tree
oDOM = BuildDOMDoc(sURL)
If IsEmpty(oDOM) Then Exit Sub 'There was an error loading the XML file

REM Copy specific data into a new Calc spreadsheet
oCalcDoc = StarDesktop.loadComponentFromURL( _
    "private:factory/scalc", "_blank", 0, Array())
oSheet = oCalcDoc.Sheets(0)
GetData(oDOM, oSheet, tDeadline)

REM Show the original XML text file in a new Writer text document
REM LibreOffice requires that the FilterName be specified, AOO does not.
Dim noArgs(1) As New com.sun.star.beans.PropertyValue
noArgs(0).Name = "FilterName"
noArgs(0).Value = "Text"
oWriterDoc = StarDesktop.loadComponentFromURL(sURL, "_blank", 0, noArgs() )
End Sub

REM Read the relevant data from the DOM document
Sub GetData(oDOMDoc, oSheet, tDeadline as Date)
    Dim oVtList                'List of all validTo nodes in the DOM document
    Dim oSlotItem              'slot = parent node of the validTo node
    Dim oSlotChildList         'List of all child nodes of a slot
    Dim oSlotChild             'A specific child node of a slot
    Dim sID As string          'Value of the slot attribute "ID"
    Dim i%, j%, k%
    Dim iCustCount As Integer  'Number of customers using a slot
    Dim aoCusts()              'Array of customer nodes within a slot
    Dim iCustLimit As Integer  'Default upper value for the array dimension
    Dim sTitle As String       'The title of the slot's document
    Dim iEntries As Integer    'The number of rows written to the sheet

    REM A slot node includes all information on the specific slot service.
    REM List all nodes named "validTo", so all slots with the "status"
    REM attribute's value "reserved" or "free" are not included.

    oVtList = oDOMDoc.getElementsByTagName("validTo")
    iCustLimit = 10 'There will seldom be more than 10 customers to a slot.

    For i = 0 To oVtList.getLength - 1
        ' Use item to get an element return by getElementsByTagName.
        ' Use getParentNode to get the parent "slot" node.
        oSlotItem = oVtList.item(i).getParentNode

```

```

' IsProperSlot checks to see if the node is active.
If IsProperSlot(oSlotItem) Then
    sID = oSlotItem.getAttribute("ID")
    REM The validTo element has exactly one text node
    REM containing a date string, so getFirstChild is used to
    REM get the text element. GetNodeValue returns the text
    REM in that element, which is a date.
    If IsWithinDateTarget _
        (oVtList.item(i).getFirstChild.getNodeValue, tDeadline) > 0 Then

        ' Everything passes, so collect data.
        ' A list of child elements of the slot is created to inspect
        ' each contained element in order to extract both the document title
        ' and the customers' data. The Select Case construction is used:
        oSlotChildList = oSlotItem.getChildNodes
        REM Iterate over the list of child nodes to collect the data.
        ReDim aoCusts(1 To iCustLimit)
        iCustCount = 0
        sTitle = ""
        For j = 0 To oSlotChildList.getLength - 1
            oSlotChild = oSlotChildList.item(j)
            Select Case oSlotChild.getNodeName
                Case "document"
                    If oSlotChild.hasAttribute("title") Then
                        sTitle = oSlotChild.getAttribute("title")
                    End If
                Case "customer"
                    iCustCount = iCustCount + 1
                    REM Just in case there really are more customers than thought
                    If iCustCount > iCustLimit Then
                        ReDim Preserve aoCusts(1 To iCustCount)
                    End If
                    aoCusts(iCustCount) = oSlotChild
                Case Else
            End Select
        Next
        If sTitle <> "" AND iCustCount > 0 Then
            For k = 1 To iCustCount
                REM Write the data to the Calc sheet
                ' Having collected all relevant data of a slot GetData
                ' can store it in the Calc sheet calling the ImportSlot
                ' subroutine. At last, when we are through with all
                ' slots, the Calc sheet is sorted for convenience.
                ImportSlot(aoCusts(k), sID, sTitle, oSheet)
            Next
            iEntries = iEntries + iCustCount
        End If
    End If
End If
Next
SortCustomers(oSheet, iEntries)
End Sub

```

```

REM Check, if the specific slot has the suiting attribute values
Function IsProperSlot(oSlot) As Boolean
    IsProperSlot = False
    If oSlot.hasAttribute("status") AND oSlot.hasAttribute("ID") Then
        If oSlot.getAttribute("status") = "active" Then
            IsProperSlot = True
        End If
    End If
End Function

REM Check, if validTo shows a date earlier than the deadline
Function IsWithinDateTarget(sDate As String, tDeadline As Date) As Integer
    Dim tValidTo As Date
    On Error Goto Catch

    REM If the Basic function DateValue cannot convert the string
    REM to a date value, the runtime error 5 is thrown. In this case
    REM the return value of -1 lets the calling routine establish a
    REM device to track down erroneous database entries.
    tValidTo = DateValue(sDate)
    If tDeadline < tValidTo Then
        IsWithinDateTarget = 0
    Else
        IsWithinDateTarget = 1
    End If
    Exit Function

Catch:
    IsWithinDateTarget = -1
End Function

REM Writes the relevant slot data to the Calc sheet
Sub ImportSlot(oCust, sID As String, sTitle As String, oSheet)
    REM There is one row for each document belonging to each customer.
    REM So if multiple customers share one document there are as many rows
    REM as there are customers to a document.
    Static iRow As Integer      'Row count

    If iRow = 0 Then            'The header
        oSheet.getCellByPosition(0, iRow).String = "Name"
        oSheet.getCellByPosition(1, iRow).String = "Email"
        oSheet.getCellByPosition(2, iRow).String = "Gender"
        oSheet.getCellByPosition(3, iRow).String = "ID"
        oSheet.getCellByPosition(4, iRow).String = "Title"
    End If
    iRow = iRow + 1
    If oCust.hasAttribute("name") Then
        oSheet.getCellByPosition(0, iRow).String = _
            oCust.getAttribute("name")
    End If

```

```

If oCust.hasAttribute("email") Then
    oSheet.getCellByPosition(1, iRow).String = _
        oCust.getAttribute("email")
End If
If oCust.hasAttribute("gender") Then
    oSheet.getCellByPosition(2, iRow).String = _
        oCust.getAttribute("gender")
End If
oSheet.getCellByPosition(3, iRow).String = sID
oSheet.getCellByPosition(4, iRow).String = sTitle

End Sub

REM Sort the sheet for convenience
Sub SortCustomers(oSheet, iRows As Integer)
    Dim oRange
    Dim aSortFields(2) As new com.sun.star.util.SortField
    Dim aSortDesc(1) As new com.sun.star.beans.PropertyValue

    REM Sort options:
    REM (1) Customer's name
    REM (2) Customer's email (in case different persons have identical names)
    REM (3) Slot ID
    With aSortFields(0): .Field = 0
        .SortAscending = True: End With
    With aSortFields(1): .Field = 1
        .SortAscending = True: End With
    With aSortFields(2): .Field = 2
        .SortAscending = True: End With

    With aSortDesc(0): .Name = "SortFields"
        .Value = aSortFields(): End With
    With aSortDesc(1): .Name = "ContainsHeader"
        .Value = True : End With

    oRange = oSheet.getCellRangeByPosition(0, 0, 4, iRows)
    oRange.sort(aSortDesc())
End Sub

```

15.12.2. Write XML File

Two steps are used to export the Calc document to an XML file.

1. Build the DOM document tree from scratch.
2. Parse the tree and build the output simple text lines from the data. In other words, you have to add all the elements in angle brackets, the attribute pairs, the comments regarding the rules of XML – and format the tree output in a readable way.

As you can see from Table 216, the DOM document is a specialized node. It can use the parent XNode methods and a couple of additional ones that are listed in Table 218.

Table 218. *Methods implemented in the com.sun.star.xml.dom.XDocument interface.*

Method	Description (taken from the API documentation)
createAttribute(name)	Creates an Attr of the given name.
createAttributeNS(namespaceURI, qualifiedName)	Creates an attribute of the given qualified name and namespace URI.
createCDATASection(data)	Creates a CDATASection node whose value is the specified string.
createComment(text)	Creates a Comment node given the specified string.
createDocumentFragment()	Creates an empty DocumentFragment object.
createElement(tagName)	Creates an element of the type specified.
createElementNS(namespaceURI, qualifiedName)	Creates an element of the given qualified name and namespace URI.
createEntityReference(name)	Creates an EntityReference object.
createProcessingInstruction(target, data)	Creates a ProcessingInstruction node given the specified name and data strings.
createTextNode(text)	Creates a Text node given the specified string.
getDoctype()	The Document Type Declaration (see DocumentType) associated with this document.
getDocumentElement()	This is a convenience attribute that allows direct access to the child node that is the root element of the document.
getElementById(elementIdstring)	Returns the Element whose ID is given by elementId.
getElementsByTagName(tagname)	Returns a NodeList of all the Elements with a given tag name in the order in which they are encountered in a preorder traversal of the Document tree
GetElementsByTagNameNS(namespaceURI, localName)	Returns a NodeList of all the Elements with a given local name and namespace URI in the order in which they are encountered in a preorder traversal of the Document tree.
getImplementation()	The DOMImplementation object that handles this document.
importNode(importedNode, deep)	Imports a node from another document to this document.

Use the DocumentBuilder service to create an empty DOM document object.

```
oDocBuilder = createUnoService("com.sun.star.xml.dom.DocumentBuilder")
oDOM = oDocBuilder.newDocument
```

Next, populate the document with child nodes. Child nodes are created and then added to the parent element. Child nodes must be created by the document's create methods. It does not matter when the nodes are created, or in which order. Use appendChild to append a child to a node. Each new child is added to the end of the list of the already added child nodes. If the child elements are to be output in sequence, you must take care to add them in the desired order. Use createComment to create a comment node with the comment text string as the argument. Use createElement to create an element node with the tag name as the argument.

You must obey the XML naming rules: Names can contain letters, numbers, and other characters. Spaces are not allowed. The first character may not be a number or punctuation. The name may not start with the string "xml" (be it upper or lower case).

```
REM Child nodes added to the root node
oNComment = oDOM.createComment(sComment)
```

```

oNPlayers = oDOM.createElement("Players")
oDOM.appendChild(oNComment)
oDOM.appendChild(oNPlayers)

REM A child node added to another node
oNPlayer = oDOM.createElement("Player")
oNPlayers.appendChild(oNPlayer)

```

Attributes need not be created by the DOM document. You can set – that means add – them directly using the XElement method `setAttribute(name, value)`, because the order is not so relevant for the attributes. They are listed in a `NamedNodeMap` and can be retrieved by name. The `setAttribute` method has two arguments, the attribute's name and the attribute's value, both are strings:

```
oNPlayer.setAttribute(oSheet.getCellByPosition(iCol, 0).String, sCellContent)
```

Use the `createTextNode` method to create a text node with the text string as the argument. As a text node can have no attributes and no child nodes there is seldom a need to reference the node by a variable:

```
oNName.appendChild(oDOM.createTextNode(sCellContent))
```

To show the whole process with a real Calc sheet I devised a list of a couple of current and former tennis professionals (see Figure 109). This sheet is taken as the data source of the XML output.

	A	B	C	D	E	F	G	H	I	J	K
1	ID	Gender	Nationality	Name	Record	Titles	HighRank	Melbourne	Paris	London	NewYork
2	1	m	USA	Andre Agassi	870:274	60	1	1995, 2000, 2001, 2003	1999	1992	1994, 1999
3	2	m	D	Boris Becker	713:214	49	1	1991, 1996		1985, 1986, 1989	

Figure 109. Detail of the source sheet for the XML export.

The macro in Listing 468 reads a sheet as shown in Figure 109 and returns a DOM document tree.

Listing 470. Creation a DOM from a Calc sheet.

```

REM Convert a sheet data table into XML using DOM.
REM Each row is a record. The first row contains the headers.
REM Assign each column to the proper node as text or attribute.
REM This function is devised for the specific sheet data.
REM The function returns a DOM document.
Function MakeXMLFromSheet(oSheet)
    Dim oDocBuilder 'The DOM DocumentBuilder interface
    Dim oDOM         'The DOM document tree
    Dim iRow&, iCol&
    Dim sCellContent$ 'All XML values are strings
    Dim sComment$     'A comment that is not part of the data
    Dim oNComment     ' Comment node
    Dim oNPlayers     ' "Players" element
    Dim oNPlayer      ' "Player" element is a child of the "Players" element
    Dim oNName        ' "Name" element is a child of a "Player" element
    Dim ONStats       ' "Statistics" element is a child of the "Player" element
    Dim oNGSlams      ' "GrandSlams" element node is a child of a "Player" element
    Dim oNGSlam       ' Each Grand Slam element is a child of the "GrandSlams" element

```

```

REM Create the empty DOM document tree.
oDocBuilder = createUnoService("com.sun.star.xml.dom.DocumentBuilder")
oDOM = oDocBuilder.newDocument

REM The DOM document will hold the following tree:
REM <!-- ... -->
REM <Players>
REM   <Player ID="..." Gender="..." Nationality="...">
REM     <Name>The player's name</Name>
REM     <Statistics Record="..." Titles="..." HighRank="..." />
REM     <GrandSlams>
REM       <Melbourne>List of victory years</Melbourne>
REM       <Paris>...</Paris>
REM       ... (more Grand Slam nodes)
REM     </GrandSlams>
REM   </Player>
REM   ... (more Player nodes)
REM </Players>

sComment = "This is a list of famous tennis players who dominated" _
          & CHR$(10) & _
          "    the world ATP tennis circus in the past decades."
REM Create the comment node and the "Players" element and add them
REM to the end of the list of children of the root node.
oNComment = oDOM.createComment(sComment)
oNPlayers = oDOM.createElement("Players")
oDOM.appendChild(oNComment)
oDOM.appendChild(oNPlayers)

REM Access the record cells starting with row 1, as row 0 are the headers.
iRow = 1
Do While oSheet.getCellByPosition(0, iRow).Value <> 0
  REM With each new row a new "Player" element is created and added
  REM to the end of the list of children of the "Players" element.
  oNPlayer = oDOM.createElement("Player")
  oNPlayers.appendChild(oNPlayer)
  REM Create three elements and add them to the end
  REM of the list of children of the "Player" element.
  oNName = oDOM.createElement("Name")
  oNStats = oDOM.createElement("Statistics")
  oNGSlams = oDOM.createElement("GrandSlams")
  oNPlayer.appendChild(oNName)
  oNPlayer.appendChild(oNStats)
  oNPlayer.appendChild(oNGSlams)
  For iCol = 0 To 10
    sCellContent = Trim(oSheet.getCellByPosition(iCol, iRow).String)
    Select Case iCol
      Case 0, 1, 2 'The player's attributes ID, Gender, and Nationality
        REM Set the attribute with two arguments:
        REM name from the header, value from the current cell
        oNPlayer.setAttribute _
          (oSheet.getCellByPosition(iCol, 0).String, sCellContent)
    End Select
  Next iCol
Next iRow

```



```

Case 3
    REM Append a new text node with the player's name as value
    REM as the only child of the "Name" element.
    oNName.appendChild(oDOM.createTextNode(sCellContent))
Case 4, 5, 6    'The player's statistics
    REM Set the attribute with two arguments:
    REM name from the header, value from the current cell
    oNStats.setAttribute _
        (oSheet.getCellByPosition(iCol, 0).String, sCellContent)
Case 7, 8, 9, 10 'The player's Grand Slam victories.
    REM Create a new Grand Slam element and add it to the end
    REM of the list of children of the "GrandSlams" element.
    REM The header is taken as its tag name.
    oNGSlam = oDOM.createElement _
        (oSheet.getCellByPosition(iCol, 0).String)
    oNGSlams.appendChild(oNGSlam)
    REM Append a new text node with the Grand Slam years as value
    REM as the only child of the Grand Slam element.
    oNGSlam.appendChild(oDOM.createTextNode(sCellContent))
End Select
Next
iRow = iRow + 1
Loop
MakeXMLFromSheet = oDOM
End Function

```

The macro creates and adds a comment and the container element “Players” to the root node oDOM. For each new row (record), a “Player” element is created and added to the “Players” element. Each “Player” node contains the “Name”, “Statistics”, and “GrandSlams” child elements.

The macro then iterates over the columns. Columns 0, 1, and 2 are attributes of the “Player” element. The attributes are set from the cell content as value and the column header as name – the headers were purposely named to be valid XML names. Column 3, the person’s name, is added as a text node to the “Name” element. Columns 4, 5, and 6 are attributes of the “Statistics” element. They are set in the same way as columns 0, 1, and 2. Finally, columns 7 through 10 contain the years of victory in the four Grand Slam events. For each of the locations a new element is created using the header as its tag name and added to the “GrandSlams” element. The current cell content is added as a text node to this new Grand Slam element.

Table 219. *Methods implemented in the com.sun.star.xml.dom.XElement interface.*

Method	Description (taken from the API documentation)
getAttribute(name)	Retrieves an attribute value by name.
getAttributeNode(name)	Retrieves an attribute node by name.
getAttributeNodeNS(namespaceURI, name)	Retrieves an Attr node by local name and namespace URI.
getAttributeNS(namespaceURI, name)	Retrieves an attribute value by local name and namespace URI.
getElementsByTagName(name)	Returns a NodeList of all descendant Elements with a given tag name, in the order in which they are encountered in a preorder traversal of this Element tree.

Method	Description (taken from the API documentation)
getElementsByTagNameNS(namespaceURI, name)	Returns a NodeList of all the descendant Elements with a given local name and namespace URI in the order in which they are encountered in a preorder traversal of this Element tree.
getTagName()	The name of the element.
hasAttribute(name)	Returns true when an attribute with a given name is specified on this element or has a default value, false otherwise.
hasAttributeNS(namespaceURI, name)	Returns true when an attribute with a given local name and namespace URI is specified on this element or has a default value, false otherwise.
removeAttribute(name)	Removes an attribute by name.
removeAttributeNode(oldAttr)	Removes the specified attribute node.
removeAttributeNS(namespaceURI, name)	Removes an attribute by local name and namespace URI.
setAttribute(name, value)	Adds a new attribute.
setAttributeNode(newAttr)	Adds a new attribute node.
setAttributeNodeNS(newAttr)	Adds a new attribute.
setAttributeNS(namespaceURI, qualifiedName, value)	Adds a new attribute.

Table 219 shows the methods that are implemented by the XElement interface. Every XElement is also an XNode so the methods in Table 217 are also implemented.

The following code snippet demonstrates how to iterate through child nodes and identify the specific node types:

Listing 471. Processing child nodes.

```

If oNode.hasChildNodes Then
    oChildren = oNode.getChildNodes
    For i = 0 To oChildren.getLength - 1
        oChild = oChildren.item(i)
        If oChild.getNodeType = com.sun.star.xml.dom.NodeType.COMMENT_NODE Then
            'Comment nodes can contain text, but not attributes or child nodes.
            sCommentText = oChild.getNodeValue
        ElseIf Child.getNodeType = com.sun.star.xml.dom.NodeType.ELEMENT_NODE Then
            ...
        ElseIf Child.getNodeType = com.sun.star.xml.dom.NodeType.TEXT_NODE Then
            'Comment nodes can contain text, but not attributes or child nodes.
            sText = oChild.getNodeValue
    
```

Element nodes can contain attributes. Comments and text nodes cannot have attributes. Use getAttributes to access the attribute nodes as a NamedNodeMap. Null is returned if no attributes are present. The following code iterates through the attributes, getting the attribute name and value.

Listing 472. Iterate through each attribute for an element.

```

oAttList = oElement.getAttributes
If Not IsNull(oAttList) Then
    
```

```

For i = 0 To oAttList.getLength - 1
    oAtt = oAttList.item(i)
    sValue = oAtt.getNodeValue
    sName = oAtt.getNodeName

```

Element nodes can contain child nodes: multiple comments, multiple elements or one single text node. So the code inspects these child nodes recursively. The subroutine PrintDom is called for each element level.

Listing 473. Macro to write a DOM document as an XML text file.

REM Export the contents of a Calc sheet into an XML file.

```

Sub ExportSheetToXML
    REM Typically this subroutine will be very short:
    'Dim sURL
    'sURL = ChooseAFileName 'Function stored in this file's UNO module
    'WriteDomToFile(MakeXMLFromSheet(ThisComponent.Sheets(0)), sURL)
    REM But for this example a new Calc document is created first,
    REM some data are written into sheet 0,
    REM and at the end the stored XML file is opened for inspection.

    Dim oCalcDoc          'The source Calc spreadsheet document
    Dim oSheet             'The source sheet
    Dim sURL As String     'The URL of the output file

    sURL = GetWorkDir() & "/oome_xml_export.xml"
    oCalcDoc = StarDesktop.loadComponentFromURL( _
        "private:factory/scalc", "_blank", 0, Array())
    oSheet = oCalcDoc.Sheets(0)
    REM Fill the sheet with fancy data.
    CreateExampleSheet(oSheet)
    REM The following subroutine has to be devised using the specific sheet data.
    REM Typically it is placed in the document library.
    WriteDomToFile(MakeXMLFromSheet(oSheet), sURL)
    Wait 500              'Give the OS time to store the file to disk
    REM Load the XML file in a Writer document.
    REM LibreOffice requires that the FilterName be specified, AOO does not.
    Dim noArgs(1) As New com.sun.star.beans.PropertyValue
    noArgs(0).Name = "FilterName"
    noArgs(0).Value = "Text"
    StarDesktop.loadComponentFromURL(sURL, "_blank", 0, noArgs() )
End Sub

```

REM Writes the structure of the current DOM document
 REM into a valid and nicely formatted XML text file.
 REM This subroutine and the called subroutines and functions are not
 REM document specific and needn't be part of the document library.
 REM They can be stored in the "My Macros" library.

```

Sub WriteDomToFile(oDOM, sFilePath As String)
    Dim oSimpleFileAccess 'SimpleFileAccess service
    Dim oOutputStream      'Stream returned from SimpleFileAccess
    Dim oTextOutput        'TextOutputStream service
    Dim oNodes              'List of child nodes of the root node

```

```

Dim i%
Dim iIndentLevel As Integer 'Indentation level
Dim iIndentSpaces As Integer 'Number of spaces added to each indentation level

On Error Goto Catch

iIndentSpaces = 2
REM Set the output stream
sFilePath = ConvertToURL(sFilePath)
oSimpleFileAccess = createUnoService("com.sun.star.ucb.SimpleFileAccess")
With oSimpleFileAccess
    If .exists(sFilePath) Then .kill(sFilePath)
    oOutputStream = .openFileWrite(sFilePath)
End With

oTextOutput = createUnoService("com.sun.star.io.TextOutputStream")
With oTextOutput
    .OutputStream = oOutputStream
    .setEncoding("UTF-8")
    REM The first line is a processing instruction. It usually isn't part of
    REM the DOM tree. So we write it separately.
    .WriteString("<?xml version=""1.0"" encoding=""UTF-8""?>" & CHR$(10))

    REM A DOM tree can consist of zero, one or more child nodes on the
    REM root level. The child nodes are treated hierarchically.
    If oDOM.hasChildNodes Then
        oNodes = oDOM.getChildNodes
        For i = 0 To oNodes.getLength - 1
            PrintDom(oNodes.item(i), oTextOutput, iIndentLevel, iIndentSpaces)
        Next
    End If
    .closeOutput
End With
oOutputStream.closeOutput

Exit Sub

Catch:
Print "Error " & Err & " (" & Error(Err) & ")"
End Sub

REM Writes the elements of a DOM tree recursively line after line into a
REM text file. Mark to start with iLevel 0.
REM Indents lower levels by accumulating iIndent spaces, except for text nodes:
REM they are written in the same line directly after the element start tag
REM and are directly followed by the element end tag.
REM It is assumed that there are either one text node or one or more other
REM child nodes to an element, if any.
Sub PrintDom(oNode, oStream, iLevel As Integer, iIndent As Integer)
    Dim oElementChildren
    Dim oChild
    Dim sLine As String

```

```

Dim sAtt As String
Dim sIndent As String
Dim i%, iLen%
Dim sNodeName As String

sNodeName = oNode.getNodeName
sIndent = String(iLevel * iIndent, " ")

REM Only comments and elements are treated.
If oNode.getNodeType = com.sun.star.xml.dom.NodeType.COMMENT_NODE Then
    REM XML comment
    sLine = sIndent & "<!-- " & oNode.getNodeValue & " -->"
    oStream.writeString(sLine & CHR$(10))
Elseif oNode.getNodeType = _
    com.sun.star.xml.dom.NodeType.ELEMENT_NODE Then
    REM XML element
    sAtt = AttString(oNode.getAttributes)
    REM Check, if the element has data. Otherwise the element is skipped.
    If oNode.hasChildNodes OR sAtt <> "" Then
        oElementChildren = oNode.getChildNodes
        If HasContent(oElementChildren) Then
            sLine = sIndent & "<" & sNodeName & sAtt & ">" 'Start tag line
            iLen = oElementChildren.getLength
            If iLen = 1 Then
                REM Test for text node, assuming that there are no other
                REM sibling nodes besides a text node.
                oChild = oElementChildren.item(0)
                If oChild.getNodeType = com.sun.star.xml.dom.NodeType.TEXT_NODE Then
                    sLine = sLine & oChild.getNodeValue & "</" & sNodeName & ">"
                    REM Write the line: start tag plus text value plus end tag.
                    oStream.writeString(sLine & CHR$(10))
                    Exit Sub
                End If
            End If
            REM At this point there are child elements other than text nodes.
            REM Write the start tag line.
            oStream.writeString(sLine & CHR$(10))
            For i = 0 To iLen - 1
                REM Start the recursion, increment the indentation level
                PrintDom(oElementChildren.item(i), oStream, iLevel + 1, iIndent)
            Next
            sLine = sIndent & "</" & sNodeName & ">" 'End tag line
            REM Write the end tag line.
            oStream.writeString(sLine & CHR$(10))
        Else
            REM There are no child elements to be written.
            REM If there are attributes, the short notation is used.
            REM If there are not even attributes, no element tag is written.
            If sAtt <> "" Then
                sLine = sIndent & "<" & sNodeName & sAtt & " />"
                REM Write the element in a short notation line.
                oStream.writeString(sLine & CHR$(10))
            End If
        End If
    End If

```

```

        End If
    End If
End If
End Sub

```

REM Returns a string of all the attributes of an element.

```

Function AttString(oAttList) As String
    Dim oAtt                'A single attribute node
    Dim sValue As String    'A single attribute's value
    Dim sAtts As String     'The string containing all attributes
    Dim i%

    If Not IsNull(oAttList) Then
        For i = 0 To oAttList.getLength - 1
            oAtt = oAttList.item(i)
            sValue = oAtt.getNodeValue
            If sValue <> "" Then
                sAtts = sAtts & " " & oAtt.getNodeName & "=" & sValue & ""
            End If
        Next
    End If
    AttString = sAtts
End Function

```

REM Checks recursively, if there is some child node with content, be it text,
 REM comment or attribute value other than "". Returns True, if some content
 REM is found in the tree, otherwise False.

REM It is called by PrintDom, before the actual element start tag is written,
 REM so PrintDom can abstain from printing an empty element.

```

Function HasContent(oElementList) As Boolean
    Dim oChild                'A single child node of oElementList
    Dim oAttributes           'The attributes of oChild
    Dim oChildren             'The child nodes of oChild
    Dim i%, j%

    For i = 0 To oElementList.getLength - 1
        oChild = oElementList.item(i)
        If oChild.hasAttributes Then
            oAttributes = oChild.getAttributes
            For j = 0 To oAttributes.getLength - 1
                If oAttributes.item(j).getNodeValue <> "" Then
                    HasContent = True
                    Exit Function
                End If
            Next
        End If
    Next
End If

If oChild.getNodeType = com.sun.star.xml.dom.NodeType.TEXT_NODE Then
    If oChild.getNodeValue <> "" Then
        HasContent = True
        Exit Function
    End If
End If

```

```

Elseif oChild.getNodeType = com.sun.star.xml.dom.NodeType.COMMENT_NODE Then
    If oChild.getNodeValue <> "" Then
        HasContent = True
        Exit Function
    End If
Else
    oChildren = oChild.getChildNodes()
    If oChildren.getLength <> 0 Then
        REM Start the recursion
        If HasContent(oChildren) Then
            HasContent = True
            Exit Function
        End If
    End If
End If
Next
HasContent = False
End Function

```

The WriteDomToFile subroutine takes two arguments, the DOM document and the file URL. An output stream is obtained from the SimpleFileAccess, which is then used by a TextOutputStream to write a UTF-8 encoded text file one line at a time.

1. The first line of an XML file is the xml processing instruction that is not part of the DOM document.
2. The macro iterates over root nodes children and prints each using PrintDom. PrintDom calls itself to print children of the node with which it was called.
3. PrintDom accepts four arguments: the current node, the output stream, the indentation level, and the number of spaces used for indentation. The first level is not indented, for each lower level – that is for each consecutive call of PrintDom – two spaces are added.
4. If the current node is a comment, PrintDom writes the text of the comment in the XML comment notation: <!-- Text -->. Line feed characters within the text break the output into new lines. PrintDom terminates.
5. For element nodes, PrintDom uses the AttString function to print attributes.

Special attention is made to print the XML in a “pretty” way; for example, if possible, a node is printed as <Name/> rather than <Name></Name>.

15.13. Charts

Many things can be done with charts, but only a small portion of the capabilities are shown here in an attempt to provide an overview of what can be accomplished.

Although a chart is a document (it has its own document model), charts are always embedded in other OOo documents. Objects that support embedded charts (Calc sheets) implement the

getCharts method as defined by the XTableChartsSupplier interface, which returns a TableCharts service.

```
ThisComponent.Sheets(0).getCharts()
```

Table 220. Methods implemented in the *com.sun.star.table.XTableCharts* interface.

Method	Description
addNewByName (name, size, data, bCol, bRow)	Create and add the named chart.
createEnumeration()	Create an enumeration.
getByIndex(i)	Get chart by index.
getByName(name)	Get chart by name.
getCount()	Number of charts.
getElementNames()	Array of chart names.
hasByName (name)	Determine if a named chart exists.
hasElements ()	Determine if any charts exist.
removeByName (name)	Delete a named chart.

Sample data is needed to discuss charts.

Listing 474. Create sample chart data.

```
Function CreateCalcForChart
    Dim oCalcDoc          'The source Calc spreadsheet document
    Dim oSheet            'The source sheet
    Dim sDataRng$         'Where is the data

    oCalcDoc = StarDesktop.loadComponentFromURL( "private:factory/scalc", "_blank", 0, Array() )
    oSheet = oCalcDoc.Sheets(0)
    sDataRng = "A1:D6"
    Dim oData
    oData = Array( Array("A", "B", "C", "D"), _
        Array(3, 1, 5, 4), _
        Array(2, 2, 4, 1), _
        Array(1, 3, 3, 0), _
        Array(2, 4, 2, 1), _
        Array(3, 5, 1, 4))
    oSheet.getCellRangeByName( sDataRng ).setDataArray(oData)
    CreateCalcForChart = oCalcDoc
End Function
```

The sample data is shown below:

Table 221. Sample data for charts.

	A	B	C	D
1	A	B	C	D
2	3	1	5	4
3	2	2	4	1
4	1	3	3	0
5	2	4	2	1
6	3	5	1	4

Consider the simple example below that creates a simple chart. A Calc document is created with simple data (see Listing 474 and Table 221). If the chart does not exist, it is created 10 cm (about 4 inches) from the left edge and 1 cm from the top edge. The graph is 10 cm x 10 cm in size. An array of cell range addresses is passed as the third argument to `addNewByName` to define the data that is charted.

Listing 475. Create a simple chart.

```
Sub CreateCalcWithSimpleChart
    Dim oSheet      'Sheet containing the chart
    Dim oRect       'How big is the chart
    Dim oCharts     'Charts in the sheet
    Dim oChart      'Created chart
    Dim oAddress    'Address of data to plot
    Dim sName$      'Chart name
    Dim oChartDoc   'Embedded chart object
    Dim oTitle      'Chart title object
    Dim oDiagram    'Inserted diagram (data).
    Dim sDataRng$   'Where is the data
    Dim oCalcDoc

    oCalcDoc = CreateCalcForChart()

    sName = "ADP_Chart"
    sDataRng = "A1:D6"

    oSheet = oCalcDoc.sheets(0)
    oAddress = oSheet.getCellRangeByName( sDataRng ).getRangeAddress()
    oCharts = oSheet.getCharts()
    If NOT oCharts.hasByName(sName) Then
        oRect = createObject("com.sun.star.awt.Rectangle")
        oRect.X = 10000
        oRect.Y = 1000
        oRect.width = 10000
        oRect.Height = 10000

        ' The rectangle identifies the dimensions in 1/100 mm.
```

```

' The address is the location of the data.
' True indicates that column headings should be used.
' False indicates that Row headings should not be used.
oCharts.addNewByName(sName, oRect, Array(oAddress), True, False)
End If

oChart = oCharts.getByName( sName )
oChart.setRanges(Array(oAddress))
oChartDoc = oChart.getEmbeddedObject()
'oChartDoc.attachData(oAddress)
oTitle = oChartDoc.getTitle()
oTitle.String = "Andy - " & Now

' Create a diagram.
oDiagram = oChartDoc.createInstance( "com.sun.star.chart.LineDiagram" )
oChartDoc.setDiagram( oDiagram )
oDiagram = oChartDoc.getDiagram()
oDiagram.DataCaption = com.sun.star.chart.ChartDataCaption.VALUE
oDiagram.DataRowSource = com.sun.star.chart.ChartDataRowSource.COLUMNS
End Sub

```

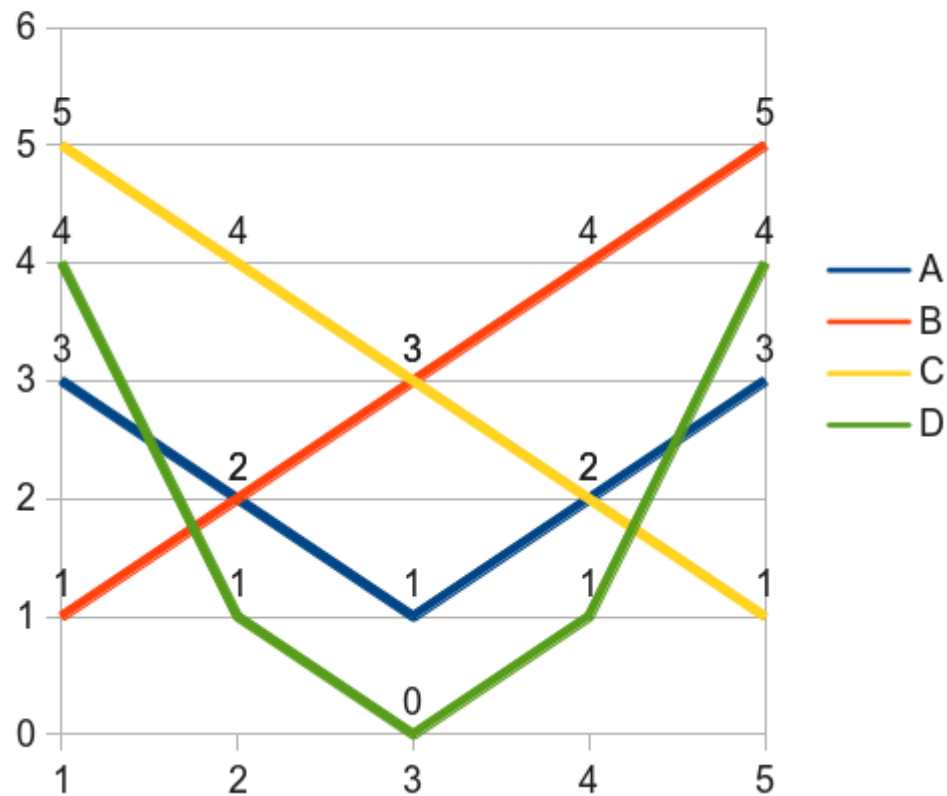


Figure 110. Simple chart created by Listing 475.

The chart can be modified after it is in the document. The following snippet changes the already inserted chart and uses three ranges that need not be contiguous.

Listing 476. Modify an existing chart.

```
sName = "ADP_Chart"
sDataRng = "A1:D6"

oSheet = ThisComponent.sheets(0)
oCharts = oSheet.getCharts()
If oCharts.hasByName(sName) Then
    oChart = oCharts.getByName(sName)
    oData = Array(oSheet.getCellRangeByName( "A1:A6" ).getRangeAddress(), _
        oSheet.getCellRangeByName( "C1:C6" ).getRangeAddress(), _
        oSheet.getCellRangeByName( "D1:D6" ).getRangeAddress())
    oChart.setRanges(oData)
End If
```

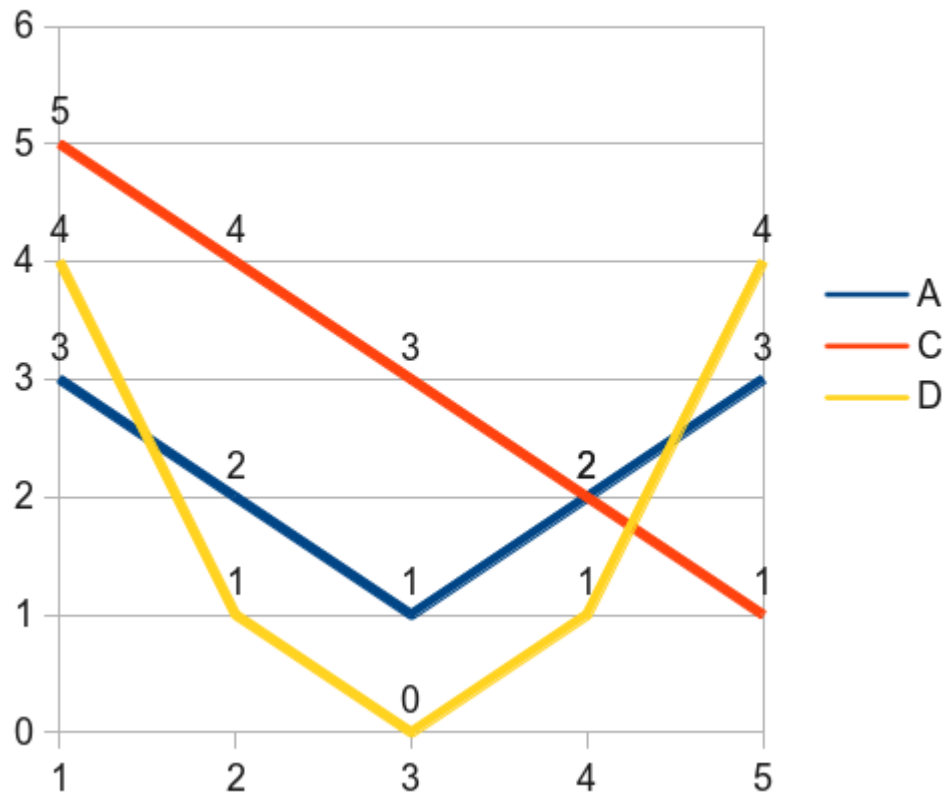


Figure 111. Modify the chart.

The chart supports the `com.sun.star.table.TableChart` service. The primary methods of interest are shown in Table 222. A `TableChart` object also implements methods to add and remove listeners.

Table 222. Main methods to control a `TableChart`.

Method	Description
<code>getEmbeddedObject()</code>	Get the chart model.
<code>getHasColumnHeaders()</code>	Are the cells of the topmost row of the source data used as column headers.
<code>getHasRowHeaders()</code>	Are the cells of the leftmost column of the source data used as row headers.
<code>getName()</code>	Get the chart name.
<code>getRanges()</code>	Get an array of ranges used in the chart.
<code>setHasColumnHeaders(b)</code>	Set if the cells of the topmost row of the source data are used as column headers.
<code>setHasRowHeaders (b)</code>	Set if the cells of the leftmost column of the source data are used as row headers.
<code>setName (name)</code>	Set the chart name.
<code>setRanges (ranges)</code>	Set the ranges used in the report.

The chart model has significant complexity – you do remember that it is an embedded document. There are multiple methods for dealing with chart titles. The chart title object supports backgrounds, bitmaps, text rotation, paragraph styles, and an array of formatted text objects that represent the text of the title. In our example, the title has a single text element, which can be changed as follows:

Listing 477. Set the chart title using `getTitleObject()`.

```
oChartDoc = oChart.getEmbeddedObject()  
Dim txtArray  
txtArray = oChartDoc.getTitleObject().getText()  
          (0).setString("New Title Text")
```

Formatted strings support properties related to formatting such as `CharColor`, `CharEmphasis`, `CharFontName`, and `CharFontStyleName`. I recommend object inspection to see what is supported. The chart doc also supports the `getTitle()` method which encapsulates the title as a single string and supports similar text formatting characteristics. The object returned by `getTitle` implements the `XShape` interface, so it can be used to change the position and size.

Listing 478. Set the chart title using `getTitle()`.

```
oChartDoc.getTitle().String = "one two three"
```

The subtitle, available using `getSubtitle()`, is also a `com.sun.star.comp.chart.Title` service, so, all methods that apply to the title, also apply to the subtitle.

Listing 479. Set the chart subtitle.

```
oChartDoc.getSubTitle().String = "I am the subtitle"
```

Use `getLegend()` to get the legend (`com.sun.star.comp.chart.Legend`), which you can use to change the font, size, position, and other formatting characteristics. The legend object does not allow you to change the text of the legend, however.

Changing the chart type involves asking the chart document for an appropriate diagram type, and then setting it (see Listing 475). Note that the `TypesDocCanCreate` macro in Listing 214 can also be used here.

Listing 480. List services the chart document can create.

```
Dim oWriteDoc  
Dim oText  
Dim sn  
oWriteDoc = StarDesktop.loadComponentFromURL("private:factory/swriter", "_blank", 0, Array())  
oText = oWriteDoc.getText()  
sn = oChartDoc.getAvailableServiceNames()  
MsgBox Join(sn, CHR$(10))  
oText.insertString ( oText.End, Join(sn, CHR$(10)), False )
```

The list returned by Apache OpenOffice 3.4 is as follows:

1. `com.sun.star.chart.AreaDiagram`
2. `com.sun.star.chart.BarDiagram`

3. `com.sun.star.chart.BubbleDiagram`
4. `com.sun.star.chart.DonutDiagram`
5. `com.sun.star.chart.FilledNetDiagram`
6. `com.sun.star.chart.LineDiagram`
7. `com.sun.star.chart.NetDiagram`
8. `com.sun.star.chart.PieDiagram`
9. `com.sun.star.chart.StockDiagram`
10. `com.sun.star.chart.XYDiagram`
11. `com.sun.star.document.ExportGraphicObjectResolver`
12. `com.sun.star.document.ImportGraphicObjectResolver`
13. `com.sun.star.drawing.BitmapTable`
14. `com.sun.star.drawing.DashTable`
15. `com.sun.star.drawing.GradientTable`
16. `com.sun.star.drawing.HatchTable`
17. `com.sun.star.drawing.MarkerTable`
18. `com.sun.star.drawing.TransparencyGradientTable`
19. `com.sun.star.xml.NamespaceMap`

The first 10 items are chart types. Changing the diagram type is trivial.

Listing 481. Set chart type to PieDiagram.

```
oChartDoc = oChart.getEmbeddedObject()  
oDiagram = oChartDoc.createInstance( "com.sun.star.chart.PieDiagram" )  
oChartDoc.setDiagram( oDiagram )
```

Use `getArea()` to get the `com.sun.star.comp.chart.Area` for the chart. Use the `Area` object to set visual characteristics such as colors, size, and position.

The actual plot is contained in the diagram, available using `getDiagram()`. To get a feel for some of the properties supported by the diagram object, look at Table 223. Numerous other properties and methods are also supported, but they are omitted due to space constraints. Inspect the object to find properties related to axis, colors, bitmaps, and grids.

Table 223. Properties used to set error bars.

Property	Description
ConstantErrorLow	Lower limit of the error range of a data row.
ConstantErrorHigh	Upper limit of the error range of a data row.
ErrorBarStyle	Use com.sun.star.chart.ErrorBarStyle constants to set the error bar style. Use: NONE, VARIANCE, STANDARD_DEVIATION, ABSOLUTE, RELATIVE, ERROR_MARGIN, STANDARD_ERROR, or FROM_DATA.
PercentageError	Percentage used to display error bars.
ErrorMargin	Percentage for the margin of errors.
ErrorIndicator	Use com.sun.star.chart.ChartErrorIndicatorType enum to specify how the error is indicated. Valid values include NONE, TOP_AND_BOTTOM, UPPER, or LOWER.
ErrorBarRangePositive	Cell range string for positive error bars – assuming the ErrorBarStyle is set to FROM_DATA.
ErrorBarRangeNegative	Cell range string for negative error bars – assuming the ErrorBarStyle is set to FROM_DATA.
MeanValue	Set to true to display the mean value for a data row as a line.
RegressionCurves	Use com.sun.star.chart.ChartRegressionCurveType enum to specify the type of regression for the data row values. Valid values include: NONE, LINEAR, LOGARITHM, EXPONENTIAL, or POWER.
DataCaption	Use com.sun.star.chart.ChartDataCaption constants to determine how points are captioned. Valid values include NONE, VALUE, PERCENT, TEXT, or SYMBOL. The chart in Figure 111 is set to VALUE, so the numeric values are shown at each data point.
DataRowSource	Use com.sun.star.chart.ChartDataRowSource enum values to determine if data is in rows or columns. Valid values include ROWS, or COLUMNS.

15.14. Conclusion

Calc documents are feature-rich and they support a wide variety of capabilities. In my opinion, Calc documents provide more functionality than any other document type supported by OOo. This chapter, therefore, is only the beginning of the wonderful things that you can do using Calc documents.

16. Draw and Impress Documents

This chapter introduces methods to manipulate and modify the content contained in Draw and Impress documents. Although the drawing functionality is the same in Draw and Impress, Impress contains extra functionality to facilitate presentations.

Draw and Impress are vector-oriented graphical applications. They can also display bitmap images, but their strength is not in photo editing. Vector-oriented applications represent many graphical objects as objects rather than as bit-mapped images. For example, lines, circles, rectangles, and text are each represented as special objects. One advantage of vector graphics is that you can independently manipulate and transform multiple overlapping elements without worrying about resolution and pixels.

Photo-editing graphics packages typically represent and manipulate images as bitmaps. A bitmap representing an image is characterized by the width and height of the image in pixels. Each pixel represents a single, colored dot in the image. The drawing capability in OpenOffice.org, however, is focused on vector operations.

Every Draw document supports the `com.sun.star.drawing.DrawingDocument` service and every Impress document supports the `com.sun.star.presentation.PresentationDocument` service. When I write a macro that must be user friendly and requires a specific document type, I verify that the document is the correct type by using the object method `supportsService` (see Listing 482).

Listing 482. *Check for an Impress document before checking for a Draw document.*

```
REM If it really matters, you should check the document type
REM to avoid a run-time error.
sDraw$ = "com.sun.star.drawing.DrawingDocument"
sImpress$ = "com.sun.star.presentation.PresentationDocument"
If ThisComponent.SupportsService(sImpress$) Then
    MsgBox "The current document is an Impress document", 0, "Impress Document"
ElseIf ThisComponent.SupportsService(sDraw$) Then
    MsgBox "The current document is a Draw document", 0, "Draw Document"
Else
    MsgBox "The current document is not the correct type", 48, "Error"
Exit Sub
End If
```

Warn

The `PresentationDocument` service implements the `DrawingDocument` service. This means that every presentation document looks like a drawing document. To distinguish between the two document types, you must first check for a presentation (Impress) document and then check for a drawing document.

The simple function below returns a new document, this is used in other macros for this chapter.

Listing 483. *Create a new document.*

```
REM Create a new empty document
REM consider document types of
REM scalc, swriter, sdraw, smath, and simpress
Function LoadEmptyDocument(docType$)
```



```

Dim noArgs()           'An empty array for the arguments
Dim sURL As String     'URL of the document to load
sURL = "private:factory/" & docType
LoadEmptyDocument = StarDesktop.LoadComponentFromUrl(sURL, "_blank", 0, noArgs())
End Function

```

16.1. Draw pages

The primary function of Draw and Impress documents is to contain graphical data, which is stored in draw pages. The primary draw-page functionality is implemented by a `GenericDrawPage`. There are two types of draw pages—the `MasterPage` and the `DrawPage`. Both draw-page types implement the `GenericDrawPage` and are therefore able to hold and manipulate the same content.

A master page acts as a common background for zero or more draw pages. Each draw page may link to one master page. Each master page is constrained as follows:

- A master page, unlike a regular draw page, may not link to a master page.
- A master page may not be removed from a document if any draw page links to it.
- Modifications made to a master page are immediately visible on every draw page that uses that master page.

The method `getMasterPages()` returns the document's collection of master pages. The method `getDrawPages()` returns the document's collection of draw pages. Both methods return the same object type; they differ only in how their contents are used (see Table 224).

Table 224. *Methods supported by the `com.sun.star.drawing.XDrawPages` interface.*

Method	Description
<code>duplicate(DrawPage)</code>	Duplicate the <code>DrawPage</code> and return the new <code>DrawPage</code> .
<code>getByIndex(Long)</code>	Get the specified draw page.
<code>getByName(String)</code>	Get the specified draw page.
<code>getCount()</code>	Return the number of contained objects as a Long Integer.
<code>hasByName(String)</code>	Return True if the named page exists.
<code>hasElements()</code>	Return True if there are draw pages to return.
<code>insertNewByIndex(Long)</code>	Create and insert a new draw page at the specified index.
<code>remove(DrawPage)</code>	Remove a specific draw page.

Each draw page has a name, which you can access by using the methods `getName()` and `setName()`. A draw page that is not a master supports the methods `getMasterPage()` and `setMasterPage()` to get and set the master page. Listing 484 demonstrates enumerating the document's draw pages and their corresponding master pages.

Listing 484. *Print information related to draw pages and master pages.*

```

Sub getPages()
    Dim s$
    Dim oDoc

```

```

oDoc = LoadEmptyDocument("simpres")
s = s & getPagesInfo(oDoc.getDrawPages(), "Draw Pages")
s = s & CHR$(10)
s = s & getPagesInfo(oDoc.getMasterPages(), "Master Pages")
MsgBox s, 0, "Pages"
End Sub

Function getPagesInfo(oDPages, sType$) As String
    Dim i%, s$
    Dim oDPage, oMPage
    s = s & "*** There are " & oDPages.getCount() & " " & sType & CHR$(10)
    For i = 0 To oDPages.getCount()-1
        oDPage = oDPages.getByIndex(i)
        s = s & "Page " & i & " = '" & oDPage.getName() & "'"
        If NOT oDPage.supportsService("com.sun.star.drawing.MasterPage") Then
            oMPage = oDPage.getMasterPage()
            s = s & " master = "
            If NOT IsNull(oMPage) AND NOT IsEmpty(oMPage) Then
                s = s & "'" & oMPage.getName() & "'"
            End If
        End If
    End For
    s = s & CHR$(10)
Next
getPagesInfo = s
End Function

```

Although you can get a draw page based on its name, you can have more than one draw page with the same name. If multiple draw pages use the same name and you retrieve the draw page based on the name, it is not specified which draw page is returned. The macro in Listing 485, which searches for a draw page with a specific name, is used in numerous places in this chapter.

Listing 485. Create a new page by name, avoiding multiple names.

```

Function createDrawPage(oDoc, sName$, bForceNew As boolean) As Variant
    Dim oPages 'All of the draw pages
    Dim oPage 'A single draw page
    Dim i% 'General index variable
    oPages = oDoc.getDrawPages()
    If oPages.hasByName(sName) Then
        REM If we require a new page then delete
        REM the page.
        If bForceNew Then
            oPages.remove(oPages.getByName(sName))
        Else
            REM Did not request a new page so return the found page
            REM and then get out of the function.
            createDrawPage = oPages.getByName(sName)
            Exit Function
        End If
    End If

    REM Did not find the page, or found the page and removed it.

```

```

REM Create a new page, set the name, and return the page.
oPages.insertNewByIndex(oPages.getCount())
oPage = oPages.getByIndex(oPages.getCount()-1)
oPage.setName(sName)
createDrawPage = oPage
End Function

```

16.1.1. Generic draw page

Both regular and master draw pages support the GenericDrawPage service. As its name implies, the GenericDrawPage service provides the primary generic drawing functionality. Writer and Calc documents provide specific styles for formatting entire pages. Draw pages, however, use the properties shown in Table 225.

Table 225. Properties defined by the *com.sun.star.drawing.GenericDrawPage* service.

Property	Description
BorderBottom	Bottom border in 1/100 mm, represented as a Long Integer.
BorderLeft	Left border in 1/100 mm, represented as a Long Integer.
BorderRight	Right border in 1/100 mm, represented as a Long Integer.
BorderTop	Top border in 1/100 mm, represented as a Long Integer.
Height	Height in 1/100 mm, represented as a Long Integer.
IsBackgroundDark	This is True if the averaged background fill color's luminance is below a specified threshold value.
NavigationOrder	Provides index access in navigation order for the top level shapes inside the page. By default this is equal to the index access of the slide itself, making the z-order the default navigation order for top level shapes.
Number	Draw page number as a Short Integer. This read-only value labels the first page 1.
Orientation	Page orientation as a <i>com.sun.star.view.PaperOrientation</i> enumeration. Two values are supported—PORTRAIT and LANDSCAPE.
UserDefinedAttributes	User-defined XML attributes.
Width	Width in 1/100 mm, represented as a Long Integer.

The primary purpose of a draw page is to contain shapes. The methods *addShape(Shape)* and *removeShape(Shape)* add and remove a shape from a document. Before a shape can be added to a draw page, it must be created by the document. Each line produced by Listing 486 and shown in Figure 112 is a separate, unrelated shape. You can independently manipulate each shapes.

Listing 486. Draw 20 lines in a Draw or Impress document.

```

Function draw20Lines()
    Dim oPage 'Page on which to draw
    Dim oShape 'Shape to insert
    Dim oPoint 'Initial start point of the line
    Dim oSize 'Width and height of the line
    Dim i% 'Index variable
    Dim n% 'Number of iterations to perform
    Dim nShift%'Shift the graphic down
    Dim oDoc

```

```

oDoc = LoadEmptyDocument("sdraw")
draw20Lines = oDoc
oPage = createDrawPage(oDoc, "Test Draw", True)
nShift = oPage.Height / 4

n = 20
For i = 0 To n
    oShape = oDoc.CreateInstance("com.sun.star.drawing.LineShape")
    oShape.LineColor = RGB( 255, 0, i+20 )
    oShape.LineWidth = 20
    oPoint = oShape.Position
    oPoint.X = oPage.Width / 4
    oPoint.Y = i * oPage.Height / n / 4 + nShift
    oShape.Position = oPoint
    oSize = oShape.Size
    oSize.Height = (oPage.Height - 2 * i * oPage.Height / n) / 4
    oSize.Width = oPage.Width / 2
    oShape.Size = oSize
    oPage.add(oShape)
Next
End Function

```

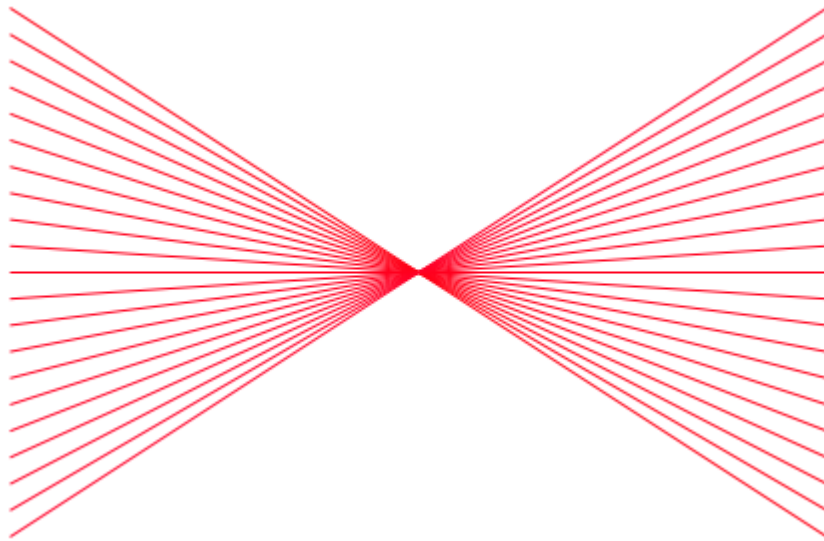


Figure 112. *Twenty lines in a Draw document.*

The macro in Listing 486 works with Draw and Impress. The 20 red lines are drawn on a page named “Test Draw”. If the page does not exist, it is created.

16.1.2. Combining shapes

The macro in Listing 486 creates 20 independent lines. You can group shapes together and then manipulate them as a single shape. The method `group(XShapes)` accepts a collection of shapes and turns them into a single group; an `XShapeObject` is returned. The macro in Listing 487 starts by calling Listing 486 and then it adds all of the lines to a single group.

Listing 487. *Group many shapes into a single shape.*

```
Sub groupShapes
    Dim oPage    'Page on which to draw
    Dim oShapes  'Shapes to group
    Dim i%       'Index variable
    Dim oDoc

    REM Create the shapes!
    oDoc = draw20Lines()
    oPage = oDoc.getDrawPages().getByName("Test Draw")

    REM Create a shape collection object to group the shapes
    oShapes = createUnoService("com.sun.star.drawing.ShapeCollection")
    For i = 0 To oPage.getCount() - 1
        oShapes.add(oPage.getByIndex(i))
    Next
    oPage.group(oShapes)
End Sub
```

When several shapes are grouped together using the group() method, the entire group is added as a single shape into the draw page, which you can retrieve by using oPage.getByIndex(). It's no longer possible to select a single line and independently manipulate it. To convert a group of shapes back to independent shapes, use the ungroup(XShapeObject) method. The ungroup() method removes the objects from the group and adds them back to the draw page as individual objects (see Listing 488).

Listing 488. *Convert a grouped shape back into multiple shapes.*

```
Sub unGroupShapes
    Dim oPage    'Page on which to draw
    Dim oShape    'Single shape
    Dim i%       'Index variable

    oPage = ThisComponent.getDrawPages().getByName("Test Draw")
    For i = 0 To oPage.getCount() - 1
        oShape = oPage.getByIndex(i)
        If oShape.supportsService("com.sun.star.drawing.GroupShape") Then
            oPage.ungroup(oShape)
        End If
    Next
End Sub
```

Although shapes that are grouped together are manipulated as one shape, they are really a collection of shapes. The combine(XShapes) method, on the other hand, converts each shape into a PolyPolygonShape and then combines them into one PolyPolygonShape. The new shape is added to the draw page, and the original shapes are removed and deleted. The split(XShape) method converts the shape to a PolyPolygonShape (if it is not already) and then the shape is split into several shapes of type PolyPolygonShape. The new shapes are added to the draw page and the original shape is removed and deleted.

Listing 489. Combine all shapes into a ShapeCollection.

```
Sub combineShapes
    Dim oPage, oShapes, i%, oDoc

    REM Create the shapes!
    oDoc = draw20Lines()
    oPage = oDoc.getDrawPages().getByName("Test Draw")

    oShapes = createUnoService("com.sun.star.drawing.ShapeCollection")
    For i = 0 To oPage.getCount() - 1
        oShapes.add(oPage.getByIndex(i))
    Next
    oPage.combine(oShapes)
End Sub
```

Combining a shape does not change how the shape appears so the output of Listing 489 is the same as Listing 486. Binding a shape, as shown in Listing 490, however, connects the lines together with a Bezier curve.

Listing 490. Bind shapes together.

```
Sub bindShapes()
    Dim oPage, oShapes, i%

    REM Create the shapes!
    draw20Lines()
    oPage = ThisComponent.getDrawPages().getByName("Test Draw")

    oShapes = createUnoService("com.sun.star.drawing.ShapeCollection")
    For i = 0 To oPage.getCount() - 1
        oShapes.add(oPage.getByIndex(i))
    Next
    oPage.bind(oShapes)
End Sub
```

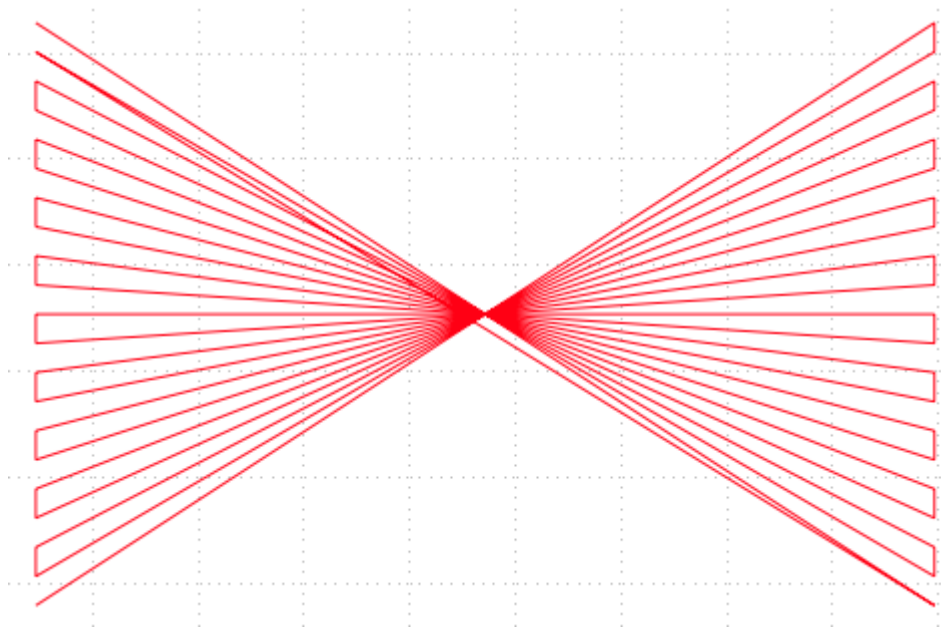


Figure 113. Using *bind* connects the lines with Bezier curves.

The `unbind(XShape)` method converts the shape to a `PolyPolygonShape` (if it is not already) and then each line segment is converted to a new `PolyPolygonShape`. The original shape is removed from the draw page and deleted. The shape produced by Listing 490, combines 21 lines into a one shape. Using `unbind` on the single shape does not change what the user sees, but each line is now its own shape, so there are 41 shapes rather than just one; the original 21 shapes and 20 lines connecting the original 21.

16.2. Shapes

Graphical content is expressed as a shape object. Shape objects are created by the document and then added to the draw page. To quickly find out what objects a document can create, use the `TypesDocCanCreate` macro in Listing 214 with a Draw document to see the objects that a Draw document can create. Filter the list a little more if you only want to see the shape types.

Listing 491. Shape services supported by Draw.

```
Sub SupportedDrawDocShapes
  Dim oDrawDoc
  ' Create a temporary draw document.
  oDrawDoc = LoadEmptyDocument("sdraw")
  ' Generate the list
  TypesDocCanCreate(oDrawDoc, "shape")
  ' Close the temporary draw document.
  oDrawDoc.Close(true)
End Sub
```

Use the same method to find shapes supported by Impress.

Listing 492. Shape services supported by Impress.

```
Sub SupportedImpressDocShapes
  Dim oDoc
```

```

' Create a temporary Impress document.
oDoc = LoadEmptyDocument("simplpress")
' Generate the list
TypesDocCanCreate(oDoc, "shape")
' Close the temporary Impress document.
oDoc.Close(true)
End Sub

```

The general drawing shapes are shown in Table 226. The shapes specific to a presentation document are shown in Table 227. Many of the shapes are either not published, or not documented. If a service or interface is not “published”, the developers expect the service or interface to change in the future, be removed, or to be unsuitable for public use for some reason. If a service is not documented, then no documentation is easily available, so it is probably not published. Use objects that are not documented or published at your own risk, they may change in the future. Sadly, even the documented shapes may not be well documented.

Tip If a service or interface is not “published”, the developers expect the service or interface to change in the future, be removed, or to be unsuitable for public use for some reason.

Some shape types are not created directly by the document; for example the PolyPolygonBezierShape. Sadly, these types are also poorly documented. Some of the types may be created indirectly. As a test, I used an Impress document to create an instance of CalcShape, but the returned object did not support the CalcShape service. Inserting the created service in the Impress document caused the shape to become an OLE2Shape service – and it still did not support the CalcShape service.

Table 226. Draw shapes (com.sun.star.drawing).

Shape Type	Description
AppletShape	Encapsulate a Java Applet.
CaptionShape	Rectangular drawing shape with an additional set of lines. It can be used as a description for a fixed point inside a drawing.
ClosedBezierShape	A series of Bezier shapes that are closed.
ClosedFreeHandShape	Closed Freehand shape drawn using the GUI.
ConnectorShape	Used to connect shapes or glue points.
ControlShape	A shape that displays a control such as a button.
CustomShape	Create a custom shape; but it is not published.
EllipseShape	Draw a circle, an ellipse, or an arc.
FrameShape	Undocumented, but seems to be used by other things as a frame.
GraphicObjectShape	Display a graphic object such as a bitmap image. There are separate types for presentation documents and drawing documents.
GroupShape	Represent multiple shapes as a single shape.
LineShape	A single line.
MeasureShape	A shape used for measuring in a diagram.
MediaShape	Undocumented.

Shape Type	Description
OLE2Shape	Display an OLE object in a Draw document.
OpenBezierShape	A series of Bezier lines.
OpenFreeHandShape	Open Freehand shape drawn using the GUI.
PageShape	Display a preview of another page in a Draw document.
PluginShape	Represent a media type that is not directly supported.
PolyLinePathShape	Undocumented.
PolyLineShape	A series of connected straight lines.
PolyPolygonBezierShape	Undocumented.
PolyPolygonPathShape	Undocumented.
PolyPolygonShape	A series of straight lines with the first and last points connected.
RectangleShape	Draw rectangles.
Shape3DCubeObject	Undocumented.
Shape3DExtrudeObject	Undocumented.
Shape3DLatheObject	Undocumented.
Shape3DPolygonObject	Undocumented.
Shape3DSceneObject	Undocumented.
Shape3DSphereObject	Undocumented.
TableShape	Table object in a Draw document.
TextShape	Box designed to hold text.

Table 227. Impress shapes (com.sun.star.presentation).

Shape Type	Description
CalcShape	Calc sheet in an Impress document.
ChartShape	Chart shape in an Impress document.
DateTimeShape	Not published.
FooterShape	Not published.
GraphicObjectShape	Display a graphic object such as a bitmap image in an Impress document.
HandoutShape	A drawing document PageShape for handouts in an Impress document.
HeaderShape	Not published.
MediaShape	Undocumented.
NotesShape	A TextShape for notes in an Impress document.
OLE2Shape	Display an OLE object in an Impress document.
OrgChartShape	Not documented.
OutlinerShape	A TextShape for outlines in an Impress document.
PageShape	Display a preview of another page in an Impress document.
SlideNumberShape	Not published.
SubtitleShape	A TextShape for subtitles in an Impress document.
TableShape	Table object in an Impress document.
TitleTextShape	A TextShape for titles in an Impress document.

16.2.1. Common attributes

A shape's position is stored in a com.sun.star.awt.Point structure, which contains two Long Integer values, X and Y, representing the upper-left corner in 1/100 mm. A shape's size is stored in a com.sun.star.awt.Size structure, which contains two Long Integer values, Width and Height, in 1/100 mm.

Table 228. Methods supported by Shape objects.

Method	Description
getPosition()	Get the shape's current position in 1/100 mm.
setPosition(Point)	Set the shape's current position in 1/100 mm.
getSize()	Get the shape's current size in 1/100 mm.
setSize(Size)	Set the shape's current size in 1/100 mm.
getGluePoints()	Get an object that provides indexed access to a set of glue points used internally by the object. Each glue point is a com.sun.star.drawing.GluePoint2 structure (see Table 242).
getShapeType()	String representing the shape's type.

Macros that deal with shape objects frequently require the Size and Point structures. The two methods in Listing 493 make it easier to create and set these structures.

Listing 493. *Create and return a Point or Size structure.*

```
Function CreatePoint(ByVal x As Long,ByVal y As Long) As com.sun.star.awt.Point
    Dim oPoint
    oPoint = createUnoStruct( "com.sun.star.awt.Point" )
    oPoint.X = x : oPoint.Y = y
    CreatePoint = oPoint
End Function

Function CreateSize(ByVal x As Long,ByVal y As Long) As com.sun.star.awt.Size
    Dim oSize
    oSize = createUnoStruct( "com.sun.star.awt.Size" )
    oSize.Width = x : oSize.Height = y
    CreateSize = oSize
End Function
```

Interfaces define methods and they may be derived from other interfaces. Services, on the other hand, implement interfaces and other services. Services also define properties. Some services are defined strictly to define a group of related properties. The properties defined by the Shape service are general and applicable to most shape types (see Table 229).

Table 229. *Properties defined by the com.sun.star.drawing.Shape service.*

Property	Description
ZOrder	Long Integer representing the ZOrder of this shape. This controls the drawing order of objects, effectively moving an object forward or backward.
LayerID	Short Integer identifying the layer that contains the shape.
LayerName	Name of the layer that contains the shape.
Printable	If True, the shape is included in printed output.
MoveProtect	If True, the shape cannot be moved interactively by the user.
Name	Shape name as a String.
SizeProtect	If True, the user may not change the shape's size.
Style	Shape's style as a com.sun.star.style.XStyle object.
Transformation	Transformation matrix of type com.sun.star.drawing.HomogenMatrix3 that can contain translation, rotation, shearing, and scaling.

OOo defines separate services that encapsulate properties and methods specific to lines, text, shadows, shape rotation, and filling area. Not all shape types support all of these services. For example, it makes no sense for a line shape to support the properties and methods related to filling areas. The services of interest for the common properties:

- com.sun.star.drawing.Text
- com.sun.star.drawing.LineProperties
- com.sun.star.drawing.FillProperties
- com.sun.star.drawing.ShadowProperties

- com.sun.star.drawing.RotationDescriptor

I used a macro to quickly determine which shapes support which common service. First, a list of services where the service name contains the word shape; for example, “com.sun.star.presentation.TableShape”.

Listing 494. *Return a filtered list of services supported by an object.*

```
Function GetListCanCreate(oObj, nameFilter$)
    Dim allNames    ' List of all the names.
    Dim s$
    Dim i%

    s = ""
    allNames = oObj.getAvailableServiceNames()

    For i = LBound(allNames) To UBound(allNames)
        If (InStr(allNames(i), nameFilter) > 0) Then
            If Len(s) > 0 Then
                s = s & "," & allNames(i)
            Else
                s = allNames(i)
            End If
        End If
    Next
    GetListCanCreate = Split(s, ",")
End Function
```

The following macro creates an Impress document. The impress document is asked which services it can create that contain the word “Shape”. Each shape is then inspected to see if it supports the categories of interest. Sadly, none of the categories of interest are supported by the shape until after the shape is inserted into the document. Each shape, therefore, is added to the Impress document, inspected to see what services it supports, and then that shape is disposed.

TIP	When a shape is created, it is not fully fleshed out and available until after it has been inserted into a document.

Listing 495. *Categorize shape types.*

```
Sub CategorizeShapeTypes
    Dim oSize as New com.sun.star.awt.Size
    Dim oPos as New com.sun.star.awt.Point

    Dim Categories ' Service names of interest
    Dim iRow%      ' Iterate over data rows.
    Dim iName%     ' Iterate over supported service names.

    Dim oShape     ' Created shapes.
    Dim oPage      ' Draw page where shapes are inserted.

    Dim oCalc      ' Calc document to hold results.
    Dim oImpress   ' Impress document on which to test.
    Dim oData      ' Holds temporary Cell data.
```

```

Dim oCellRange ' Range to hold the results in Calc.
Dim shapeNames ' List of supported shape names.
Dim oSheet      ' Calc sheet containing results.
Dim oRow        ' One row from the oData array.

REM Load an Impress document and a Calc document.
oImpress = LoadEmptyDocument("sImpress")
oCalc = LoadEmptyDocument("sCalc")

REM Get the list of supported shape names.
shapeNames = GetListCanCreate(oImpress, "shape")

Rem Make each shape the same size and position.
oSize.Width = 6000
oSize.Height = 6100
oPos.X = 6000
oPos.Y = 5000

REM First draw page to contain the created shapes.
oPage = oImpress.DrawPages.getByIndex(0)

REM Categories of interest.
Categories = Array("com.sun.star.drawing.Text", _
    "com.sun.star.drawing.LineProperties", _
    "com.sun.star.drawing.FillProperties", _
    "com.sun.star.drawing.ShadowProperties", _
    "com.sun.star.drawing.RotationDescriptor")

REM Get the cell range that will hold the results.
oSheet = oCalc.getSheets().getByIndex(0)
oCellRange = oSheet.getCellRangeByPosition(0, 0, _
    UBound(Categories) - LBound(Categories) + 1, _
    UBound(shapeNames) - LBound(shapeNames) + 1)

oData = oCellRange.getDataArray()

REM Set the row title
oRow = oData(0)
For iRow = 1 To UBound(oRow)
    oRow(iRow) = Categories(iRow - 1)
Next

REM For each shape name
For iName = 0 To UBound(shapeNames)
    REM Get the row for this shape name
    oRow = oData(iName + 1)
    oRow(0) = shapeNames(iName)

    REM Create the shape, add it to the draw page.
    oShape = oImpress.CreateInstance(shapeNames(iName))
    oPage.Add(oShape)
    oShape.setSize(oSize)
    oShape.setPosition(oPos)

```

```

REM Find out which services are supported by the current shape.
For iRow = LBound(Categories) To UBound(Categories)
    If oShape.supportsService(Categories(iRow)) Then
        oRow(iRow + 1) = "X"
    End If
Next

REM Get rid of the created shape.
oShape.dispose()
Next

REM Set the results in the Calc document.
oCellRange.setDataArray(oData)
End Sub

```

The Calc document created by Listing 495 is not sorted, and the full service names are used. The data is summarized here in two tables. , The com.sun.star.drawing shapes are shown in Table 230, and the com.sun.star.presentation shapes are in Table 231.

Table 230. Which Drawing shapes support which service.

Shape	Text	Line Properties	Fill Properties	Shadow Properties	Rotation Descriptor
AppletShape					
CaptionShape	X	X	X	X	X
ClosedBezierShape	X	X	X	X	X
ClosedFreeHandShape	X	X	X	X	X
ConnectorShape	X	X		X	X
ControlShape					
CustomShape	X	X	X		
EllipseShape	X	X	X	X	X
FrameShape					
GraphicObjectShape	X			X	X
GroupShape					
LineShape	X	X		X	X
MeasureShape	X	X		X	X
MediaShape					
OLE2Shape					
OpenBezierShape	X	X	X	X	X
OpenFreeHandShape	X	X	X	X	X
PageShape					
PluginShape					
PolyLinePathShape	X	X		X	X
PolyLineShape	X	X		X	X
PolyPolygonPathShape	X	X	X	X	X
PolyPolygonShape	X	X	X	X	X
RectangleShape	X	X	X	X	X
Shape3DCubeObject					
Shape3DExtrudeObject					
Shape3DLatheObject					
Shape3DPolygonObject					
Shape3DSceneObject					
Shape3DSphereObject					
TableShape					
TextShape	X	X	X	X	X

Table 231. Which Impress shapes support which service.

Shape	Text	Line Properties	Fill Properties	Shadow Properties	Rotation Descriptor
CalcShape					
ChartShape					
DateTimeShape	X	X	X	X	X
FooterShape	X	X	X	X	X
GraphicObjectShape	X			X	X
HandoutShape					
HeaderShape	X	X	X	X	X
MediaShape					
NotesShape	X	X	X	X	X
OLE2Shape					
OrgChartShape					
OutlinerShape	X	X	X	X	X
PageShape					
SlideNumberShape	X	X	X	X	X
SubtitleShape	X	X	X	X	X
TableShape					
TitleTextShape	X	X	X	X	X

Drawing text service

Any shape that supports the `com.sun.star.drawing.Text` service has the ability to contain text. The drawing text service supports the standard `com.sun.star.text.XText` interface and a special set of drawing text properties. Besides character and paragraph properties, the drawing text properties service defines properties specifically designed for shape objects (see Table 232).

Table 232. Properties defined by the `com.sun.star.drawing.TextProperties` service.

Property	Description
IsNumbering	If True, numbering is ON for the text in this shape.
NumberingRules	Describes the numbering levels as a sequence of <code>com.sun.star.style.NumberingRule</code> .
TextAnimationAmount	Number of pixels the text is moved in each animation step.
TextAnimationCount	Number of times the text animation is repeated.
TextAnimationDelay	Delay, in thousandths of a second, between each animation step.
TextAnimationDirection	Enumerated value of type <code>com.sun.star.drawing.TextAnimationDirection</code> : LEFT, RIGHT, UP, and DOWN.

Property	Description
TextAnimationKind	Enumerated value of type com.sun.star.drawing.TextAnimationKind: - NONE – No animation. - BLINK – Continuously switch the text between visible and invisible. - SCROLL – Scroll the text. - ALTERNATE – Scroll the text from one side to the other and back. - SLIDE – Scroll the text from one side to the final position and stop.
TextAnimationStartInside	If True, the text is visible at the start of the animation.
TextAnimationStopInside	If True, the text is visible at the end of the animation.
TextAutoGrowHeight	If True, the shape height changes automatically when text is added or removed.
TextAutoGrowWidth	If True, the shape width changes automatically when text is added or removed.
TextContourFrame	If True, the text is aligned with the left edge of the shape.
TextFitToSize	Enumerated value of type com.sun.star.drawing.TextFitToSizeType: - NONE – The text size is defined by the font properties. - PROPORTIONAL – Scale the text if the shape is scaled. - ALLLINES – Like PROPORTIONAL, but the width of each row is also scaled. - RESIZEATTR – If the shape is scaled, scale the font attributes.
TextHorizontalAdjust	Enumerated value of type com.sun.star.drawing.TextHorizontalAdjust: - LEFT – The left edge of the text is adjusted to the left edge of the shape. - CENTER – The text is centered inside the shape. - RIGHT – The right edge of the text is adjusted to the right edge of the shape. - BLOCK – The text extends from the left to the right edge of the shape.
TextLeftDistance	Distance from the left edge of the shape to the text as a Long Integer.
TextLowerDistance	Distance from the lower edge of the shape to the text as a Long Integer.
TextMaximumFrameHeight	Limit a shape's height as it grows automatically as you enter text.
TextMaximumFrameWidth	Limit a shape's width as it grows automatically as you enter text.
TextMinimumFrameHeight	Limit a shape's minimum height as it grows automatically as you enter text.
TextMinimumFrameWidth	Limit a shape's minimum width as it grows automatically as you enter text.
TextRightDistance	Distance from the right edge of the shape to the text as a Long Integer.
TextUpperDistance	Distance from the upper edge of the shape to the text as a Long Integer.
TextVerticalAdjust	Enumerated value of type com.sun.star.drawing.TextVerticalAdjust: - TOP – The top edge of the text is adjusted to the top edge of the shape. - CENTER – The text is centered inside the shape. - BOTTOM – The bottom edge of the text is adjusted to the bottom edge of the shape. - BLOCK – The text extends from the top to the bottom edge of the shape.
TextWritingMode	Enumerated value of type com.sun.star.text.TextWritingMode: - LR_TB – Text is written left to right and top to bottom. - RL_TB – Text is written right to left and top to bottom. - TB_RL – Text is written top to bottom and lines are placed right to left.

MeasureShape

The default behavior of a MeasureShape (see Figure 114) is to display the actual length of the shape. The macro in Listing 496 creates two measure shapes and changes the text of one of them to “Width”. To help illustrate setting the properties in Table 232, the TextAnimationKind property is set to SCROLL so that the text continuously scrolls from right to left; or at least it would if it were in an Impress document.

Listing 496. Draw a MeasureShape.

```
Sub drawMeasureShape()  
    Dim oPage 'Page on which to draw  
    Dim oShape 'Shape to insert  
    Dim oStart As new com.sun.star.awt.Point  
    Dim oEnd As new com.sun.star.awt.Point  
    Dim oDrawDoc 'Temporary draw document.  
  
    oDrawDoc = LoadEmptyDocument("sdraw")  
  
    oPage = createDrawPage(oDrawDoc, "Test Draw", True)  
    oShape = oDrawDoc.createInstance("com.sun.star.drawing.MeasureShape")  
    oPage.add(oShape)  
  
    REM The following values MUST be set AFTER the object is inserted.  
    oStart.X = oPage.Width / 4 : oEnd.X = oPage.Width / 2  
    oStart.Y = oPage.Height/4 : oEnd.Y = oPage.Height/4  
    oShape.StartPosition = oStart  
    oShape.EndPosition = oEnd  
    oShape.setString("Width")  
    oShape.TextAnimationKind = com.sun.star.drawing.TextAnimationKind.SCROLL  
  
    oShape = oDrawDoc.createInstance("com.sun.star.drawing.MeasureShape")  
    oPage.add(oShape)  
    oStart.X = oPage.Width / 5 : oEnd.X = oPage.Width / 5  
    oStart.Y = oPage.Height/4 : oEnd.Y = oPage.Height/2.5  
    oShape.StartPosition = oStart  
    oShape.EndPosition = oEnd  
End Sub
```

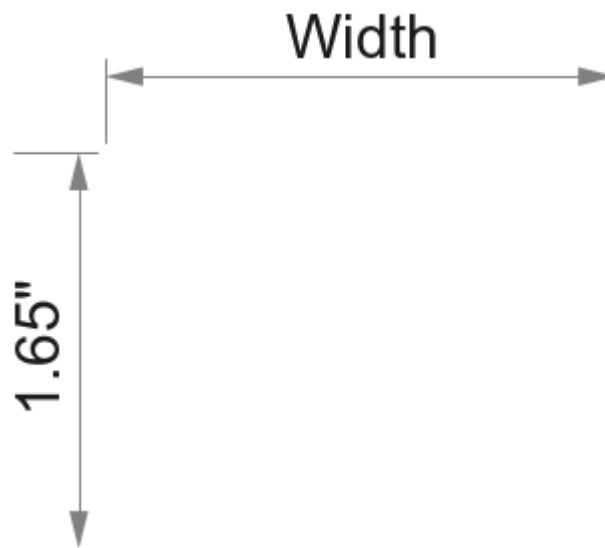


Figure 114. By default, measure shapes show the actual size—you can override this.

Drawing line properties

Shapes that support the `com.sun.star.drawing.LineProperties` service can influence how lines are drawn. Most shapes support line properties because most shapes contain lines of some sort. The specific properties dealing with line endpoints and start points are supported only by shapes with open ends.

Table 233. *Properties defined by the `com.sun.star.drawing.LineProperties` service.*

Property	Description
LineColor	Line color as a Long Integer.
LineDash	Enumerated value of type <code>com.sun.star.drawing.LineDash</code> that defines how dashed lines are drawn. - Style – Enumerated value of type <code>com.sun.star.drawing.DashStyle</code>: RECT, ROUND, RECTRELATIVE, and ROUNDRELATIVE. - Dots – Number of dots in this LineDash as a Long Integer. - DotLen – Length of a dot as a Long Integer. - Dashes – Number of dashes as a Short Integer. - DashLen – Length of a single dash as a Long Integer. - Distance – Distance between the dots as a Long Integer.
LineEnd	Line end in the form of a <code>PolyPolygonBezierCoords</code> .
LineEndCenter	If True, the line ends in the center of the polygon.
LineEndName	Name of the line end point's <code>PolyPolygonBezierCoords</code> .
LineEndWidth	Width of the line end polygon.
LineJoint	Enumerated value of type <code>com.sun.star.drawing.LineJoint</code> : - NONE – The joint between lines is not connected. - MIDDLE – The middle value between joints is used. - BEVEL – The edges are joined by lines. - MITER – Lines join at intersections.

Property	Description
	• ROUND – Lines are joined with an arc.
LineStart	Line start in the form of a PolyPolygonBezierCoords .
LineStartCenter	If True, the line starts from the center of the polygon.
LineStartName	Name of the line start point's PolyPolygonBezierCoords.
LineStartWidth	Width of the line start polygon.
LineStyle	Enumerated value of type com.sun.star.drawing.LineStyle: NONE (the line is hidden), SOLID, and DASH.
LineTransparence	Line transparency percentage as a Short Integer.
LineWidth	Line width in 1/100 mm as a Long Integer.

Filling space with a ClosedBezierShape

Shapes that support the com.sun.star.drawing.FillProperties service are able to control how open area in the shape is filled. In general, if the shape is closed, it can be filled.

Table 234. Properties defined by the com.sun.star.drawing.FillProperties service.

Property	Description
FillBackground	If True, the transparent background of a hatch-filled area is drawn in the current background color.
FillBitmap	If the FillStyle is BITMAP, this is the bitmap used.
FillBitmapLogicalSize	Specifies if the size is given in percentage or as an absolute value.
FillBitmapMode	com.sun.star.drawing.BitmapMode enum: • REAPEAT Repeate over the fill area. • STRETCH Stretch to fill the area. • NO_REPEAT One copy in the original size.
FillBitmapName	If the FillStyle is BITMAP, this is the name of the fill bitmap style used.
FillBitmapOffsetX	The horizontal offset where the tile starts. It is given as a percentage in relation to the width of the bitmap.
FillBitmapOffsetY	The vertical offset where the tile starts. It is given as a percentage in relation to the height of the bitmap.
FillBitmapPositionOffsetX	Every second line of tiles is moved the given percentage of the width of the bitmap.
FillBitmapPositionOffsetY	Every second row of tiles is moved the given percentage of the height of the bitmap.
FillBitmapRectanglePoint	The RectanglePoint specifies the position inside the bitmap to use as the top-left position for rendering.
FillBitmapSizeX	The width of the tile for filling.
FillBitmapSizeY	The height of the tile for filling.
FillBitmapURL	If the FillStyle is BITMAP, this is a URL to the bitmap used.
FillColor	Color to use if the FillStyle is SOLID.
FillGradient	If the FillStyle is GRADIENT, this describes the gradient used (see Table 235).
FillGradientName	If the FillStyle is GRADIENT, this is the name of the fill gradient style used.

Property	Description
FillHatch	If the FillStyle is HATCH, this describes the hatch used.
FillHatchName	If the FillStyle is GRADIENT, this is the name of the fill hatch style used.
FillStyle	com.sun.star.drawing.FillStyle enum: • NONE Do not fill. • SOLID Solid color. • GRADIENT Use a gradient. • HATCH Fill with a hatch. • BITMAP Use a bitmap such as a JPG.
FillTransparency	Transparency percentage if the FillStyle is SOLID.
FillTransparencyGradient	Defines the gradient with a com.sun.star.awt.Gradient structure (see Table 235).
FillTransparencyGradientName	Name of the gradient style to use; empty is okay.

Gradients are defined as follows:

Table 235. *com.sun.star.awt.Gradient structure.*

Property	Description
Style	Gradient style using an com.sun.star.awt.GradientStyle enum: • LINEAR Linear gradient. • AXIAL Axial gradient. • RADIAL Radial gradient. • ELLIPTICAL Gradient in the shape of an ellipse. • SQUARE Gradient in the shape of a square. • RECT Gradient in the shape of a rectangle.
StartColor	Gradient start color.
EndColor	Gradient end color.
Angle	Gradient angle in 1/10 degree.
Border	Per cent of the total width where just the start color is used.
XOffset	X-coordinate, where the gradient begins.
YOffset	Y-coordinate, where the gradient begins.
StartIntensity	Intensity at the start point of the gradient.
EndIntensity	Intensity at the end point of the gradient.
StepCount	Number of steps of change color.

The macro in Listing 497 draws a closed Bezier shape. The fill style is set to use a gradient, which means that the darkness of the shape changes over the shape. The resulting figure contains narrow bands of each color or intensity. You can smooth the appearance of the gradient by using the FillTransparencyGradient property as mentioned in Table 234.

Listing 497. *Draw a ClosedBezierShape with a gradient.*

Sub DrawClosedBezierShape

```

Dim oPage    'Page on which to draw
Dim oShape   'Shape to insert
Dim oCoords  'Coordinates of the polygon to insert
Dim oDrawDoc 'Temporary draw document.

oDrawDoc = LoadEmptyDocument("sdraw")

oCoords = createUnoStruct("com.sun.star.drawing.PolyPolygonBezierCoords")

REM Fill in the actual coordinates. The first and last points
REM are normal points and the middle points are Bezier control points.
oCoords.Coordinates = Array(_
    Array(_
        CreatePoint( 1000, 1000 ),_
        CreatePoint( 3000, 4000 ),_
        CreatePoint( 3000, 4000 ),_
        CreatePoint( 5000, 1000 )_
    )_
)
oCoords.Flags = Array(_
    Array(_
        com.sun.star.drawing.PolygonFlags.NORMAL,_
        com.sun.star.drawing.PolygonFlags.CONTROL,_
        com.sun.star.drawing.PolygonFlags.CONTROL,_
        com.sun.star.drawing.PolygonFlags.NORMAL _
    )_
)

oPage = createDrawPage(oDrawDoc, "Test Draw", True)
oShape = oDrawDoc.createInstance("com.sun.star.drawing.ClosedBezierShape")
oPage.add(oShape)
oShape.FillStyle = com.sun.star.drawing.FillStyle.GRADIENT
oShape.PolyPolygonBezier = oCoords
End Sub

Sub drawRectangleWithShadow()
    Dim oPage    'Page on which to draw
    Dim oShape   'Shape to insert
    Dim oDrawDoc 'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    oPage = createDrawPage(oDrawDoc, "Test Draw", True)
    oShape = oDrawDoc.createInstance("com.sun.star.drawing.RectangleShape")
    oPage.add(oShape)
    oShape.setPosition(createPoint(1000, 1000))
    oShape.setSize(createSize(4000, 1000))
    oShape.setString("box 1")
    oShape.Shadow = True

    oShape = oDrawDoc.createInstance("com.sun.star.drawing.RectangleShape")
    oPage.add(oShape)
    oShape.setPosition(createPoint(6000, 1000))

```

```

oShape.setSize(createSize(4000, 1000))
oShape.setString("box 2")
oShape.Shadow = True
oShape.ShadowXDistance = -150
oShape.CornerRadius = 100
End Sub

```

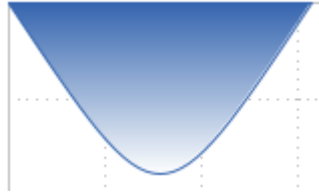


Figure 115. Bezier shape using a gradient fill.

Shadows and a RectangleShape

Shapes that support the ShadowProperties service can be drawn with a shadow. You can set the shadow location and color by using the properties in Table 236.

Table 236. Properties defined by the *com.sun.star.drawing.ShadowProperties* service.

Property	Description
Shadow	If True, the shape has a shadow.
ShadowColor	Color of the shadow as a Long Integer.
ShadowTransparency	Shadow transparency as a percentage.
ShadowXDistance	Horizontal distance of the left edge of the shape to the shadow.
ShadowYDistance	Vertical distance of the top edge of the shape to the shadow.

A common method for drawing shadows is to draw the shape at an offset location using a shadow color, and then draw the shape normally (see Figure 116). With this in mind, consider the properties ShadowXDistance and ShadowYDistance as the distance that the “shadow object” is shifted when it is drawn. The default values for ShadowXDistance and ShadowYDistance are positive, which shifts the shadow right and down. A negative shadow distance shifts the shadow left and up. The macro in Listing 498 draws two boxes; the first box uses a standard shadow that is shifted right and down, and the second box has the shadow shifted left and down (see Figure 116).

Listing 498. Draw rectangles with text, rounded corners, and drop shadows.

```

Sub drawRectangleWithShadow()
    Dim oPage 'Page on which to draw
    Dim oShape 'Shape to insert
    Dim oDrawDoc 'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    oPage = createDrawPage(oDrawDoc, "Test Draw", True)
    oShape = oDrawDoc.createInstance("com.sun.star.drawing.RectangleShape")
    oPage.add(oShape)

```

```

oShape.setPosition(createPoint(1000, 1000))
oShape.setSize(createSize(4000, 1000))
oShape.setString("box 1")
oShape.Shadow = True

oShape = oDrawDoc.createInstance("com.sun.star.drawing.RectangleShape")
oPage.add(oShape)
oShape.setPosition(createPoint(6000, 1000))
oShape.setSize(createSize(4000, 1000))
oShape.setString("box 2")
oShape.Shadow = True
oShape.ShadowXDistance = -150
oShape.CornerRadius = 100
End Sub

```



Figure 116. Notice the different shadows and that box 2 has rounded corners.

Rotation and shearing

The `com.sun.star.drawing.RotationDescriptor` provides the ability to rotate and shear a shape. Shear stretches a shape and would, for example, change a rectangle into a parallelogram. The `RotateAngle` property is a Long Integer measured in 1/100 degrees. The shape is rotated counterclockwise around the center of the shape's bounding box. The `ShearAngle` property is also a Long Integer measured in 1/100 degrees, but the shape is sheared clockwise around the center of the bounding box.

The macro in Listing 499 rotates a rectangle 20 degrees counterclockwise and shears a rectangle 25 degrees clockwise. This code also draws a normal rectangle with no rotation or shear to help you visualize the effects (see Figure 117).

Listing 499. Rotate and shear rectangles with text.

```

Sub drawRotateRectangle()
    Dim oPage 'Page on which to draw
    Dim oShape 'Shape to insert
    Dim oDrawDoc 'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    oPage = createDrawPage(oDrawDoc, "Test Draw", True)
    oShape = oDrawDoc.createInstance("com.sun.star.drawing.RectangleShape")
    oPage.add(oShape)
    oShape.setPosition(createPoint(1000, 1000))
    oShape.setSize(createSize(4000, 1500))
    oShape.setString("box 1")
    oShape.RotateAngle = 2000 '20 degrees

```



```

oShape = oDrawDoc.createInstance("com.sun.star.drawing.RectangleShape")
oPage.add(oShape)
oShape.setPosition(createPoint(1000, 1000))
oShape.setSize(createSize(4000, 1500))
oShape.FillStyle = com.sun.star.drawing.FillStyle.NONE
oShape.LineStyle = com.sun.star.drawing.LineStyle.DASH

oShape = oDrawDoc.createInstance("com.sun.star.drawing.RectangleShape")
oPage.add(oShape)
oShape.setPosition(createPoint(6000, 1000))
oShape.setSize(createSize(4000, 1500))
oShape.setString("box 2")
oShape.ShearAngle = 2500 '25 degrees

oShape = oDrawDoc.createInstance("com.sun.star.drawing.RectangleShape")
oPage.add(oShape)
oShape.setPosition(createPoint(6000, 1000))
oShape.setSize(createSize(4000, 1500))
oShape.FillStyle = com.sun.star.drawing.FillStyle.NONE
oShape.LineStyle = com.sun.star.drawing.LineStyle.DASH
End Sub

```

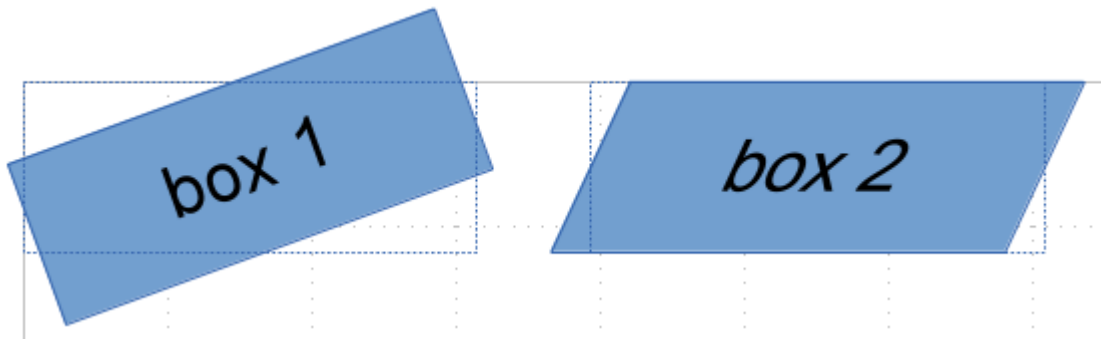


Figure 117. The rectangles with dashed lines are the original rectangles.

16.2.2. Shape types

OOo supports many different shape types, which build on each other. Most of the shape types are obvious from their names. For example, a LineShape is a line. I was initially confused, however, by the gratuitous use of the word “Poly” in shape names such as PolyLineShape and PolyPolygonShape. The prefix “Poly” comes from the Greek and it means “many.” So in OOo, a Polygon is a figure containing many angles, a PolyLineShape contains many line shapes, and a PolyPolygonShape contains many polygon shapes.

Simple lines

The purpose of the LineShape service is to draw a simple line. A LineShape requires an initial position and a size. The macro in Listing 500 draws a line from the point (1000, 1000) to the point (1999, 1999). The endpoint of the line is set by setting the shape’s size.

Listing 500. Draw a line.

```
Sub SimpleLine
    Dim oPage 'Page on which to draw
    Dim oShape 'Shape to insert
    Dim oDrawDoc 'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    oPage = createDrawPage(oDrawDoc, "Test Draw", True)
    oShape = oDrawDoc.CreateInstance("com.sun.star.drawing.LineShape")
    oPage.add(oShape)
    oShape.setPosition(CreatePoint(1000, 1000))
    oShape.setSize(CreateSize(1000, 1000))
End Sub
```

Although I have never seen it used, the LineShape service supports the service PolyPolygonDescriptor (see Table 237). The implication is that internally simple lines are represented as an open polygon that contains one line. The PolyPolygonDescriptor is used in other services as well.

Table 237. Properties in the com.sun.star.drawing.PolyPolygonDescriptor service.

Property	Description
PolygonKind	This read-only property identifies the polygon type (see Table 238).
PolyPolygon	Reference points for this polygon. This is an array of arrays. Each contained array is an array of com.sun.star.awt.Point structures. These points are used to draw the polygon and may have been transformed by a rotation or other transformation.
Geometry	These are the PolyPolygon points with no transformations.

The PolygonKind enumeration identifies the polygon type (see Table 238). The PolygonKind is a read-only property in the PolyPolygonDescriptor service (see Table 237). In other words, you can see what the type is, but you can't set it.

Table 238. Values defined by the com.sun.star.drawing.PolygonKind enumeration.

Value	Description
LINE	Identifies a LineShape.
POLY	Identifies a PolyPolygonShape.
PLIN	Identifies a PolyLineShape.
PATHLINE	Identifies an OpenBezierShape.
PATHFILL	Identifies a ClosedBezierShape.
FREELINE	Identifies an OpenFreeHandShape.
FREEFILL	Identifies a ClosedFreeHandShape.
PATHPOLY	Identifies a PolyPolygonPathShape.
PATHPLIN	Identifies a PolyLinePathShape.

The PolyPolygon property in Table 237 allows you to inspect the actual points used in the creation of the line. The code in Listing 501 assumes that oShape contains a LineShape object and it displays the two points in the line.

Listing 501. *Inspect the points used in a LineShape.*

```
x = oShape.PolyPolygon(0)
MsgBox "" & x(0).X & " and " & x(0).Y
MsgBox "" & x(1).X & " and " & x(1).Y
```

PolyLineShape

The LineShape service defines a single line, and the PolyLineShape service defines a series of lines. A LineShape is defined by setting its position and size. A PolyLineShape, however, is defined by the PolyPolygonDescriptor (see Table 237). Although it's easy to create a PolyLineShape when you know how, it isn't widely understood.

TIP The PolyPolygon property is an array of arrays that contain points.

The lines in the PolyLineShape are defined by the PolyPolygon property, which is an array that contains one or more arrays of points. Each array of points is drawn as a series of connected lines, but each array is not specifically connected to the other. The macro in Listing 502 generates two arrays of points (oPoints_1 and oPoints_2) and then the arrays are stored in another array.

Listing 502. *Draw a simple PolyLineShape.*

```
Sub SimplePolyLineShape
    Dim oPage      'Page on which to draw
    Dim oShape      'Shape to insert
    Dim oPoints_1  'First set of points to plot
    Dim oPoints_2  'Second set of points to plot
    Dim oDrawDoc    'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    oPoints_1 = Array(_
        CreatePoint( 1000, 1000 ),_
        CreatePoint( 3000, 2000 ),_
        CreatePoint( 1000, 2000 ),_
        CreatePoint( 3000, 1000 )_
    )

    oPoints_2 = Array(_
        CreatePoint( 4000, 1200 ),_
        CreatePoint( 4000, 2000 ),_
        CreatePoint( 5000, 2000 ),_
        CreatePoint( 5000, 1200 )_
    )

    oPage = createDrawPage(oDrawDoc, "Test Draw", True)
    oShape = oDrawDoc.createInstance("com.sun.star.drawing.PolyLineShape")
    oPage.add(oShape)
```

```

oShape.PolyPolygon = Array(oPoints_1, oPoints_2)
oShape.LineWidth = 50
End Sub

```



Figure 118. A single *PolyLineShape* produces two shapes that are not connected.

TIP The shape is added to the draw page before points are assigned to the PolyPolygon property.

The PolyPolygon property is an array of arrays. You can run the macro in Listing 502 with only one set of points, but the single array of points must still reside inside a second array.

Listing 503. The *PolyPolygon* property is an array of arrays of points.

```

oShape.PolyPolygon = Array(oPoints_1)

```

PolyPolygonShape

The PolyPolygonShape service defines a series of closed polygons that are not connected (see Figure 119). This service is essentially a closed-shape version of the PolyLineShape. Because it produces closed shapes, the PolyPolygonShape service supports fill properties.

The macro in Listing 504 uses the same set of points as the macro in Listing 502, but I moved things around to demonstrate different methods for creating an array of arrays of points. Both macros, however, create the shape and add it to the draw page before setting the properties.

Listing 504. Draw a simply PolyPolygonShape that is filled.

```

Sub SimplePolyPolygonShapeFilled
    Dim oPage      'Page on which to draw
    Dim oShape      'Shape to insert
    Dim oDrawDoc    'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    oPage = createDrawPage(oDrawDoc, "Test Draw", True)
    oShape = oDrawDoc.CreateInstance("com.sun.star.drawing.PolyPolygonShape")
    oPage.add(oShape)
    oShape.PolyPolygon = Array(_
        Array( CreatePoint( 1000, 1000 ), _
            CreatePoint( 3000, 2000 ), _
            CreatePoint( 1000, 2000 ), _
            CreatePoint( 3000, 1000 ) _
        ), _
        Array( CreatePoint( 4000, 1200 ), _
            CreatePoint( 4000, 2000 ), _
            CreatePoint( 5000, 2000 ), _
            CreatePoint( 5000, 1200 ) _
        ) _
    )

```

```

)

oShape.LineWidth = 50
End Sub

```



Figure 119. The PolyPolygonShape produces a closed-shape version of the PolyLineShape.

RectangleShape and TextShape

Externally, the RectangleShape and the TextShape are virtually identical. The two shape types support the same set of services (except for the defining service, of course) and they can be configured to produce the same output. The primary difference between the two shape types is their default values while producing output. In principle, properties can be adjusted from default values, so that each type could produce either output. The macro in Listing 505 creates a rectangle shape and a text shape next to each other (see Figure 120).

Listing 505. Draw a rectangle and a text object.

```

Sub SimpleRectangleShape2
    Dim oPage      'Page on which to draw
    Dim oShape     'Shape to insert
    Dim oDrawDoc   'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    oPage = createDrawPage(oDrawDoc, "Test Draw", True)
    oShape = oDrawDoc.createInstance("com.sun.star.drawing.RectangleShape")
    oPage.add(oShape)

    oShape.setPosition(createPoint(1000, 1000))
    oShape.setSize(createSize(6000, 1000))
    oShape.setString("rectangle")
    oShape.Shadow = True

    oShape = oDrawDoc.createInstance("com.sun.star.drawing.TextShape")
    oPage.add(oShape)

    oShape.setPosition(createPoint(8000, 1000))
    oShape.setSize(createSize(10000, 1000))
    oShape.setString("text")
    oShape.Shadow = True
End Sub

```

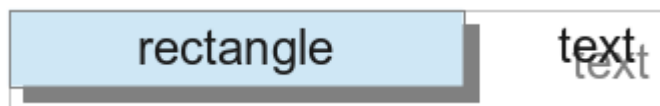


Figure 120. Two shapes that are drawn in the same way—but with different, nearly identical shape types that have different default values—produce different output.

The `RectangleShape` and the `TextShape` types both support the `CornerRadius` property. The corner radius is a Long Integer that indicates the radius of the circle used to produce the corners. This is demonstrated in Figure 116 as produced by Listing 498.

EllipseShape

A mathematician would say that an ellipse is a closed curve that is formed from two points (called the foci) in which the sum of the distances from any point on the curve to the two points is a constant. If the two foci are at the same point, the ellipse is a circle. In simpler terms, an ellipse is a circle or a squashed circle.

While drawing a rectangle, the position identifies the upper-left corner of the rectangle and then the size defines the width and height. If the same point and size is used to draw an ellipse, the ellipse will be contained inside the rectangle and will just barely touch the four sides of the rectangle. Mathematically, the sides of the rectangle are tangent to the ellipse at its principal axes, the maximum and minimum distances across the ellipse. The macro in Listing 506 starts by drawing four ellipse shapes. The final ellipse is rotated 30 degrees. The macro then draws a rectangle using the same position, size, and rotation as the last ellipse (see Figure 121). The final rectangle helps to illustrate the relationship between a rectangle and an ellipse.

Listing 506. Draw ellipse shapes.

```
Sub SimpleEllipseShapes
    Dim oPage      'Page on which to draw
    Dim oShape      'Shape to insert
    Dim i%          'Loop variable.
    Dim x           'One location point.
    Dim nLocs       'Locations array.
    Dim oDrawDoc    'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    nLocs = Array(_
        Array(CreatePoint(1000, 1000), createSize(1000, 1000)), _
        Array(CreatePoint(3000, 1000), createSize(1000, 1500)), _
        Array(CreatePoint(5000, 1000), createSize(1500, 1000)), _
        Array(CreatePoint(7000, 1000), createSize(1500, 1000)) _
    )

    oPage = createDrawPage(oDrawDoc, "Test Draw", True)
    For i = LBound(nLocs) To UBound(nLocs)
        oShape = oDrawDoc.CreateInstance("com.sun.star.drawing.EllipseShape")
        oPage.add(oShape)
        x = nLocs(i)
        oShape.setPosition(x(0))
        oShape.setSize(x(1))
        oShape.setString(i)
    Next
    oShape.RotateAngle = 3000
End Sub
```

```

REM Now draw a rectangle the same size as the last ellipse.
oShape = oDrawDoc.CreateInstance("com.sun.star.drawing.RectangleShape")
oPage.add(oShape)
oShape.setPosition(x(0))
oShape.setSize(x(1))
oShape.RotateAngle = 3000
oShape.FillStyle = com.sun.star.drawing.FillStyle.NONE
End Sub

```

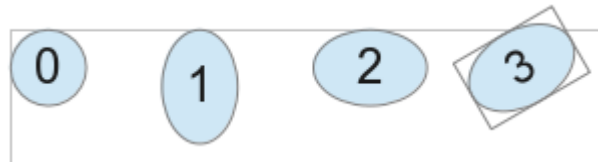


Figure 121. The size parameters determine shape; other parameters set position and orientation.

The `EllipseShape` service contains a property of type `CircleKind` that determines whether the entire ellipse should be drawn, or only a portion of it (see Table 239). In other words, you can draw an arc. The properties `CircleStartAngle` and `CircleEndAngle` define where the arc starts and ends. Each ellipse in Figure 121 uses a `FULL` `CircleKind`.

Table 239. Values defined by the `com.sun.star.drawing.CircleKind` enumeration.

Value	Description
<code>com.sun.star.drawing.CircleKind.FULL</code>	A full ellipse.
<code>com.sun.star.drawing.CircleKind.SECTION</code>	An ellipse with a cut connected by two lines.
<code>com.sun.star.drawing.CircleKind.CUT</code>	An ellipse with a cut connected by one line.
<code>com.sun.star.drawing.CircleKind.ARC</code>	An ellipse with an open cut.

Warn Both AOO and LO incorrectly swapped the meaning for `SECTION` and `CUT`.

The four different circle kinds are drawn by Listing 507 and shown in Figure 122.

Listing 507. Draw Arcs.

```

Sub ArcEllipseShapes
    Dim oPage      'Page on which to draw
    Dim oShape      'Shape to insert
    Dim i%
    Dim x
    Dim nLocs
    Dim oDrawDoc    'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    nLocs = Array(
        com.sun.star.drawing.CircleKind.FULL,
        com.sun.star.drawing.CircleKind.SECTION,
        com.sun.star.drawing.CircleKind.CUT,
        com.sun.star.drawing.CircleKind.ARC,
    )

```

```

)

oPage = createDrawPage(oDrawDoc, "Test Draw", True)
For i = LBound(nLocs) To UBound(nLocs)
    oShape = oDrawDoc.CreateInstance("com.sun.star.drawing.EllipseShape")
    oPage.add(oShape)
    oShape.setPosition(CreatePoint((i+1)*2000, 1000))
    oShape.setSize(CreateSize(1000, 700))
    oShape.setString(i)
    oShape.CircleStartAngle = 9000
    oShape.CircleEndAngle = 36000
    oShape.CircleKind = nLocs(i)
End Sub

```



Figure 122. Each supported CircleKind drawn in order.

Bezier curves

A Bezier curve is a smooth curve controlled by multiple points. Bezier curves connect the first and last points, but are only influenced by the other points. Mathematicians like Bezier curves because they are invariant under any affine mapping (any arbitrary combination of translation or rotation). Computer graphics professionals like Bezier curves because they are easy to manipulate and transform.

Bezier curves are controlled by a PolyPolygonBezierDescriptor (see Table 240), which is almost identical to the PolyPolygonDescriptor described in Table 237. The difference between the two descriptors is that each point in the Bezier curve is categorized based on how the point affects the curve.

Table 240. Properties in the *com.sun.star.drawing.PolyPolygonBezierDescriptor* service.

Property	Description
PolygonKind	This read-only property identifies the polygon type (see Table 238).
PolyPolygonBezier	Reference points for this Bezier curve. This is a PolyPolygonBezierCoords structure. The structure contains an array of points and an array of flags to categorize each point as to its function in the curve.
Geometry	This is the PolyPolygonBezierCoords with no transformations.

The PolyPolygonBezier property (see Table 240) is a PolyPolygonBezierCoords structure that contains two properties, Coordinates and Flags. The Coordinates property is an array of arrays of points that represent the control points for the Bezier curve. The Flags property is an array of arrays of PolygonFlags (see Table 241) that identifies how the corresponding point affects the curve.

Table 241. Values in the *com.sun.star.drawing.PolygonFlags* enumeration.

Value	Description
NORMAL	The curve travels through normal points.
SMOOTH	The point is smooth through the point.
CONTROL	Influence the curve.
SYMMETRIC	The point is symmetric through the point.

The macro in Listing 508 draws a small circle at each point in the Coordinates array. Drawing each point helps to visualize and understand how the different points affect the shape of a Bezier curve.

Listing 508. Draw a circle at each point in an array.

```
Sub DrawControlPoints(oCoords, oPage, oDoc, nWidth As Long)
    Dim oPoints 'One subarray of points
    Dim oPoint 'One point
    Dim oFlags 'One subarray of flags
    Dim oShape 'The circle to draw
    Dim nShape% 'Index into the oCoords arrays
    Dim i% 'General index variable

    For nShape = LBound(oCoords.Coordinates) To UBound(oCoords.Coordinates)
        oPoints = oCoords.Coordinates(nShape)
        oFlags = oCoords.Flags(nShape)
        For i = LBound(oPoints) To UBound(oPoints)
            oShape = oDoc.CreateInstance("com.sun.star.drawing.EllipseShape")
            oPage.add(oShape)
            oPoint = oPoints(i)
            REM To center the circle, I need to set the position
            REM as half width back and half width up.
            oShape.setPosition(CreatePoint(oPoint.X-nWidth/2, oPoint.Y-nWidth/2))
            oShape.setSize(CreateSize(nWidth, nWidth))
        Next
    Next
End Sub
```

Listing 509 draws two disconnected Bezier curves (see Figure 123). The second curve places two control points at the same location. The DrawControlPoints macro in Listing 508 is used to draw the control points along with the Bezier curve.

Listing 509. Draw an Open Bezier Curve.

```
Sub DrawOpenBezierCurves()
    Dim oPage 'Page on which to draw
    Dim oShape 'Shape to insert
    Dim i%
    Dim oCoords 'Coordinates of the polygon to insert
    Dim oDrawDoc 'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")
```

```

oCoords = createUnoStruct("com.sun.star.drawing.PolyPolygonBezierCoords")

REM Fill in the actual coordinates. The first and last points
REM are normal points and the middle points are Bezier control points.
oCoords.Coordinates = Array(
    Array(
        CreatePoint( 1000, 1000 ),_
        CreatePoint( 2000, 3000 ),_
        CreatePoint( 3000, 0500 ),_
        CreatePoint( 4000, 1000 ),_
    ),_
    Array(
        CreatePoint( 5000, 1000 ),_
        CreatePoint( 6500, 0200 ),_
        CreatePoint( 6500, 0200 ),_
        CreatePoint( 8000, 1000 ),_
    ),_
)
oCoords.Flags = Array(
    Array(
        com.sun.star.drawing.PolygonFlags.NORMAL,_
        com.sun.star.drawing.PolygonFlags.CONTROL,_
        com.sun.star.drawing.PolygonFlags.CONTROL,_
        com.sun.star.drawing.PolygonFlags.NORMAL,_
    ),_
    Array(
        com.sun.star.drawing.PolygonFlags.NORMAL,_
        com.sun.star.drawing.PolygonFlags.CONTROL,_
        com.sun.star.drawing.PolygonFlags.CONTROL,_
        com.sun.star.drawing.PolygonFlags.NORMAL,_
    )_
)

oPage = createDrawPage(oDrawDoc, "Test Draw", True)
oShape = oDrawDoc.createInstance("com.sun.star.drawing.OpenBezierShape")
oPage.add(oShape)
oShape.PolyPolygonBezier = oCoords
DrawControlPoints(oCoords, oPage, oDrawDoc, 100)
End Sub

```

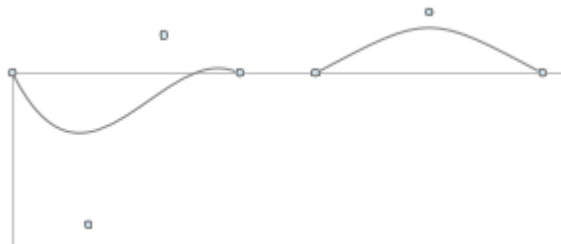


Figure 123. Notice how the control points influence the curve.

Not all combinations of points and flags are valid. A complete discussion of what constitutes a valid combination of points and flags is beyond the scope of this book. A run-time error occurs if you use an incorrect number of points or an unsupported sequence of control flags.

ConnectorShape

Use the ConnectorShape to provide a connection between two shapes. A “glue point” is a position inside a shape where the endpoint of a ConnectorShape can connect. Each glue point is defined by the GluePoint2 structure (see Table 242).

Table 242. Properties in the *com.sun.star.drawing.GluePoint2* structure.

Property	Description
Position	Glue-point position as a point structure.
IsRelative	If True, Position is given in 1/100 percent.
PositionAlignment	com.sun.star.drawing.Alignment enumerated value that specifies how the point is moved if the shape is resized. Valid values include: TOP_LEFT, TOP, TOP_RIGHT, LEFT, CENTER, RIGHT, BOTTOM_LEFT, BOTTOM, and BOTTOM_RIGHT.
Escape	com.sun.star.drawing.EscapeDirection enumerated value that specifies the escape direction for a glue point. Valid values include: SMART, LEFT, RIGHT, UP, DOWN, HORIZONTAL, and VERTICAL.
IsUserDefined	If False, this is a default glue point.

Each shape contains a default glue point at the top, right, bottom, and left of the shape. You can access a shape’s glue points by using the getGluePoints() method (see Table 228). The index of the default glue points are 0 (top), 1 (right), 2 (bottom), and 3 (left). You also can add new glue points to a shape’s default glue points (see Listing 511).

Connector shapes contain StartPosition and EndPosition properties, which identify the connector’s start and end positions. The start and end positions are used only if the corresponding properties StartShape and EndShape are empty. If the StartShape and EndShape properties are not empty, the connector shape connects to a glue point in the corresponding shape. Connector shapes reference other shapes’ glue points by index using the StartGluePointIndex and EndGluePointIndex properties.

Table 243. Properties in the *com.sun.star.drawing.ConnectorShape* service.

Property	Description
StartShape	Start shape, or empty if the start point is not connected to a shape.
StartGluePointIndex	Index of the glue point in the start shape.
StartPosition	Start point position in 1/100 mm. You can set the position only if the start point is not connected, but you can always read the point.
EndShape	End shape, or empty if the start point is not connected to a shape.
EndPosition	End point position in 1/100 mm. You can set the position only if the end point is not connected, but you can always read the point.
EndGluePointIndex	Index of the glue point in the end shape.
EdgeLine1Delta	Distance of line 1.
EdgeLine2Delta	Distance of line 2.
EdgeLine3Delta	Distance of line 3.
EdgeKind	Type of connector (see Table 244).

Four connector types are supported (see Table 244). The connector type determines how the line is drawn between the two points. The STANDARD type prefers to use three lines to connect the shapes, but it will use more lines if required.

Table 244. Properties in the *com.sun.star.drawing.ConnectorType* enumeration.

Value	Description
STANDARD	The ConnectorShape is drawn with three lines, with the middle line perpendicular to the other two.
CURVE	The ConnectorShape is drawn as a curve.
LINE	The ConnectorShape is drawn as one straight line.
LINES	The ConnectorShape is drawn with three lines.

The macro in Listing 510 draws four rectangles and then connects the rectangles using a ConnectorShape. Although the macro specifies the initial glue point, the end glue point is automatically chosen by OOo. If the macro did not explicitly set the initial glue point, it also would be automatically chosen. When a glue point is automatically chosen, it is done intelligently, as you can see in Figure 124.

Listing 510. Illustrate four connector types.

```
Sub DrawConnectorShapeNormal
    DrawConnectorShape(False, False)
End Sub

Sub DrawConnectorShape(doCustom As Boolean, useArrows as Boolean)
    Dim oPage      'Page on which to draw
    Dim oShapes     'Shapes to insert
    Dim oShape      'Shape to insert
    Dim nConTypes   'Array of connection types
    Dim i%
    Dim oGlue       'Custom Glue Shape
    Dim oStyles
    Dim oDrawDoc     'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    nConTypes = Array(_
        com.sun.star.drawing.ConnectorType.STANDARD,_
        com.sun.star.drawing.ConnectorType.CURVE,_
        com.sun.star.drawing.ConnectorType.LINE,_
        com.sun.star.drawing.ConnectorType.LINES,_
    )

    oShapes = Array(_
        oDrawDoc.CreateInstance("com.sun.star.drawing.RectangleShape"),_
        oDrawDoc.CreateInstance("com.sun.star.drawing.RectangleShape"),_
        oDrawDoc.CreateInstance("com.sun.star.drawing.RectangleShape"),_
        oDrawDoc.CreateInstance("com.sun.star.drawing.RectangleShape"),_
    )
```

```

)

REM Create the draw page and then add the shapes before manipulating them.
oPage = createDrawPage(oDrawDoc, "Test Draw", True)
For i = 0 To 3
    oPage.add(oShapes(i))
    oShapes(i).setSize(createSize(1300, 1000))
Next

oShapes(0).setPosition(createPoint(3000, 3500))
oShapes(1).setPosition(createPoint(6000, 3000))
oShapes(2).setPosition(createPoint(9000, 2500))
oShapes(3).setPosition(createPoint(6000, 4500))

For i = 0 To 3
    oShapes(i).setString(i)
    oShape = oDrawDoc.createInstance("com.sun.star.drawing.ConnectorShape")
    oPage.add(oShape)
    oShape.StartShape = oShapes(i)
    oShape.StartGluePointIndex = i
    oShape.EndShape = oShapes((i + 1) MOD 4)
    oShape.EdgeKind = nConTypes(i)

    If doCustom Then
        Rem Now create a glue point in the center of the shape.
        oGlue = createUnoStruct("com.sun.star.drawing.GluePoint2")
        oGlue.IsRelative = False
        oGlue.Escape = com.sun.star.drawing.EscapeDirection.SMART
        oGlue.PositionAlignment = com.sun.star.drawing.Alignment.CENTER
        oGlue.IsUserDefined = True
        oGlue.Position.X = oShapes(i).getPosition().X + 650
        oGlue.Position.Y = oShapes(i).getPosition().Y + 500
        oShape.StartGluePointIndex = oShapes(i).getGluePoints().insert(oGlue)
    End If

    If useArrows Then
        oStyles = oDrawDoc.getStyleFamilies().getByName("graphics")
        oShape.Style = oStyles.getByName("objectwitharrow")
    End If

Next
End Sub

```

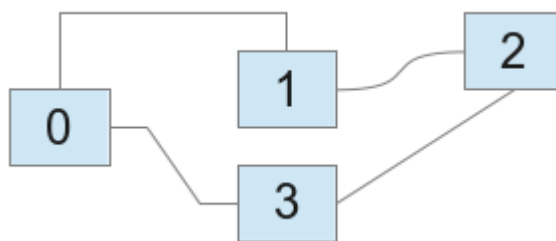


Figure 124. Notice the different connector types.

TIP

Many of a shape's properties are reset when the shape is added to a draw page. Therefore, you should set most properties after adding the shape to the draw page. It is also important to set properties in the correct order because setting some properties resets other properties. For example, setting a connector's StartShape resets the StartGluePointIndex.

Creating your own glue points

If you want to attach a connector to a shape at a location of your choosing, you must create a GluePoint2 structure and add it to the shape. Immediately after setting the EdgeKind property, create a glue point located at the center of the rectangle and then uses this point as the start point. Table 242 contains a description of the GluePoint2 structure.

Listing 511. Draw a connector shape from the center of the shape.

```
Sub DrawConnectorShapeCenter
    DrawConnectorShape(True, False)
End Sub
```

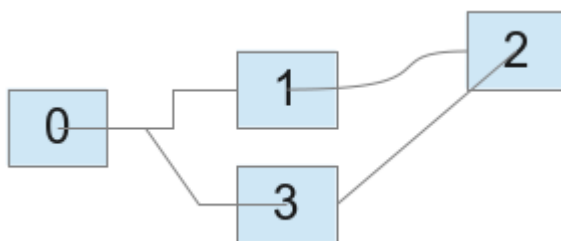


Figure 125. Custom glue points: Connectors start in the middle of the rectangles.

Adding arrows by using styles

You can set many properties for a shape by creating a style. If you frequently use certain fill styles and shadow styles, create a special style so you can quickly change the objects if required. Listing 512 displays the style families and the graphics styles supported by a draw document.

Listing 512. Print the supported graphics styles.

```
Sub PrintGraphicsStyles
    Dim oStyles
    Dim oStyleFamilies
    Dim oDrawDoc 'Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    oStyleFamilies = oDrawDoc.getStyleFamilies()
    MsgBox Join(oStyleFamilies.getElementNames(), CHR$(10)), 0, "Families"

    oStyles = oDrawDoc.getStyleFamilies().getByName("graphics")
    MsgBox Join(oStyles.getElementNames(), CHR$(10)), 0, "Graphics Styles"
    oDrawDoc.Close(True)
```

End Sub

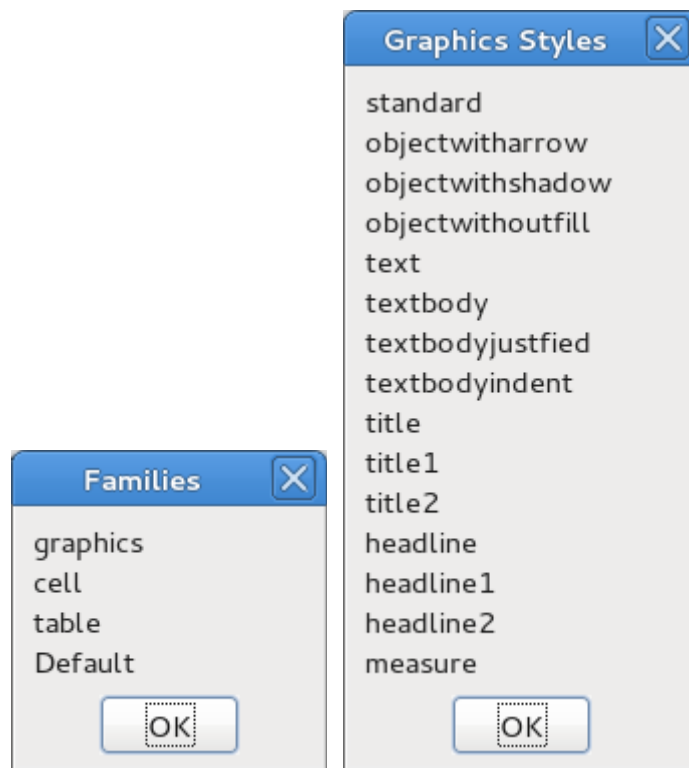


Figure 126. Graphics styles supported by Draw documents.

You can add arrows to a shape by setting the shape's style to “objectwitharrow”. If you don't like the default values in the “objectwitharrow” style, which produces very wide lines (see Figure 127), you can create your own style and use that instead.

Listing 513. Draw connectors with arrows.

```
Sub ConnectorWithArrows
    DrawConnectorShape(False, True)
End Sub
```

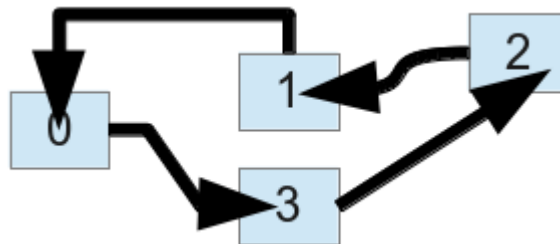


Figure 127. The default arrow style draws an arrow from the end shape to the start shape.

Insert a TableShape

The following macro inserts a TableShape into the first draw page and sets a value into a single cell. Finally, one cell is set to contain a graphic as the background fill style.

Listing 514. Insert a table into a Draw or Impress document.

```
Sub DrawTableShape
    Dim oSize as New com.sun.star.awt.Size
    Dim oPos as New com.sun.star.awt.Point
    Dim oPage      ' Draw page to contain the shape.
    Dim oTable     ' Table to insert.
    Dim oCell      ' A cell that will contain a value.
    Dim oDrawDoc   ' Temporary draw document.

    oDrawDoc = LoadEmptyDocument("sdraw")

    ' Set a size and table position.
    oSize.Width = 6000      : oSize.Height = 6100
    oPos.X = 6000           : oPos.Y = 5000

    ' Get the first draw page
    oPage = oDrawDoc.DrawPages.getByIndex(0)
    oTable = oDrawDoc.createInstance("com.sun.star.drawing.TableShape")
    oPage.add(oTable)

    ' Set the number of rows and columns to 5
    oTable.Model.Rows.InsertByIndex(1,4)
    oTable.Model.Columns.InsertByIndex(1,4)
    oTable.setSize(oSize)
    oTable.setPosition(oPos)

    ' Get the cell in the first column and second row and set to contain the text X.
    oCell = oTable.Model.getCellByPosition(0,1)
    oCell.getText().setString("X")

    ' Get the cell in the third row and column.
    oCell = oTable.Model.getCellByPosition(2,2)

    ' Fill the cell with a bitmap of an image.
    oCell.FillStyle = com.sun.star.drawing.FillStyle.BITMAP
    oCell.FillBitmapURL = _
        "http://www.openoffice.org/images/AOO_logos/OOo_Website_v2_copy.png"
End Sub
```

16.3. Forms

When a control is inserted into a document, it is stored in a form. Forms are contained in draw pages, which implement the `com.sun.star.form.XFormsSupplier` interface. The visible portion of the control—what you see—is stored in a draw page and it is represented by a `ControlShape`. The data model for the control is stored in a form and it is referenced by the `ControlShape`. The method `getForms()` returns an object that contains the forms for the draw page (see Table 245).

Table 245. Some methods supported by the *com.sun.star.form.Forms* service.

Method	Description
createEnumeration()	Create an enumeration of the forms.
getByIndex(Long)	Get a form by index.
getByName(String)	Get a form by name.
getCount()	Get the number of forms.
hasByName(String)	Return True if the form with the specified name exists.
hasElements()	Return True if the page contains at least one form.
insertByIndex(Long, Form)	Insert a form by index.
insertByName(String, Form)	Insert a form by name.
removeByIndex(Long)	Remove a form by index.
removeByName(String)	Remove the named form.
replaceByIndex(Long, Form)	Replace a form by index.
replaceByName(String, Form)	Replace a form by name.

The purpose of Listing 515 is to demonstrate how to add a form to a draw page. Forms are not interesting unless they contain a control, so Listing 515 adds a drop-down list box with some values to select. When I tested using AOO, the form was left in design mode, and you must exit design mode before you can test the control.

Listing 515. Add a form with a control to the first draw page.

```

Dim oPage           'Page on which to draw
Dim oShape           'Shape to insert
Dim oForm            'Individual form
Dim oControlModel    'Model for a control
Dim s (0 To 5) As String
Dim oDrawDoc         ' Temporary draw document.

oDrawDoc = LoadEmptyDocument("sdraw")

REM Data for the combo box!
s(0) = "Zero"       : s(1) = "One"       : s(2) = "Two"
s(3) = "Three"      : s(4) = "Four"      : s(5) = "Five"

oPage = oDrawDoc.DrawPages.getByIndex(0)

REM Create a shape for the control.
oShape = oDrawDoc.createInstance("com.sun.star.drawing.ControlShape")
oShape.Position = createPoint(3000, 4500)
oShape.Size = createSize(2500, 800)

REM Create a combo box model.
oControlModel = oDrawDoc.createInstance("com.sun.star.form.component.ComboBox")
oControlModel.Name = "NumberSelection"
oControlModel.Text = "Zero"
oControlModel.Dropdown = True

```

```

oControlModel.StringItemList = s()

REM Set the shape's control model!
oShape.Control = oControlModel

oForm = oDrawDoc.createInstance("com.sun.star.form.component.Form")
oForm.Name = "NumberForm"
oPage.Forms.insertByIndex( 0, oForm )

REM Add the control model to the first form in the collection.
oForm.insertByIndex( 0, oControlModel )
oPage.add( oShape )
End Sub

```

Regular forms, as created by Listing 515, group form components together. A DataForm, however, can connect to a database and display the results of SQL queries. An HTMLForm, on the other hand, contains controls specific to HTML pages.

Table 246. Control component that can be added to forms.

Component	Description
CheckBox	Check box control.
ComboBox	Provides text input or selection from a list of text values.
CommandButton	A clickable button.
CurrencyField	An edit field with a currency value.
DatabaseCheckBox	A data-aware check box that can be bound to a database field.
DatabaseComboBox	A data-aware combo box that can be bound to a database field.
DatabaseCurrencyField	A data-aware edit field with a currency value that can be bound to a database field.
DatabaseDateField	A data-aware date field that can be bound to a database field.
DatabaseFormattedField	A data-aware formatted field that can be bound to a database field.
DatabaseImageControl	A field for displaying images stored in a database.
DatabaseListBox	A data-aware list box that can be bound to a database field.
DatabaseNumericField	A data-aware numeric field that can be bound to a database field.
DatabasePatternField	A data-aware pattern field that can be bound to a database field.
DatabaseRadioButton	A data-aware radio button that can be bound to a database field.
DatabaseTextField	A data-aware text field that can be bound to a database field.
DatabaseTimeField	A data-aware time field that can be bound to a database field.
DateField	An edit field with a date value.
FileControl	An edit field for a file name.
FixedText	Display text that cannot be edited by the user.
FormattedField	An edit field that contains formatted text.
GridControl	Display data in a table-like way.
GroupBox	A control that can visually group controls.
HiddenControl	A control that is hidden.
ImageButton	A clickable button which is represented by an image.

Component	Description
ListBox	A control with multiple values from which to choose.
NumericField	An edit field with a numeric value.
PatternField	An edit field with text that matches a pattern.
RadioButton	A radio button.
TextField	A text-edit field that supports single-line and multi-line data.
TimeField	An edit field with a time value.

16.4. Presentations

The Presentation service contains properties (see Table 247) and methods that control a specific presentation. You can create multiple presentation objects for different types of presentations. For example, you might have one presentation that runs continuously at a trade show and one that requires manual intervention—for example, for a client sales visit. The document's `getPresentation()` method returns a new presentation object. After setting a presentation's properties as shown in Table 247, the methods `start()` and `end()` are used to start and stop a presentation. The method `rehearseTimings()` starts a presentation while displaying a running clock to help you determine the running time of your presentation.

Table 247. Properties defined by the *com.sun.star.presentation.Presentation* service.

Property	Description
AllowAnimations	If True, animations are enabled.
CustomShow	Name of a customized show to use for this presentation; an empty value is allowed.
FirstPage	Name of the first page in the presentation; an empty value is allowed.
IsAlwaysOnTop	If True, the presentation window is always the top window.
IsAutomatic	If True, page changes happen automatically.
IsEndless	If True, the presentation repeats endlessly.
IsFullScreen	If True, the presentation runs in full-screen mode.
IsLivePresentation	If True, the presentation runs in live mode.
IsMouseVisible	If True, the mouse is visible during the presentation.
Pause	Long Integer duration that the black screen is displayed after the presentation is finished.
StartWithNavigator	If True, the Navigator opens at the start of the presentation.
UsePen	If True, a pen appears during the presentation so that you can draw on the screen.

Listing 516. Start the current presentation.

```

Sub SimplePresentation()
    Dim oPres
    oPres = ThisComponent.getPresentation()
    oPres.UsePen = True
    REM This will start the presentation.
    REM Be ready to press the space bar to move through the slides.
    oPres.Start()
End Sub

```

A custom presentation can show the presentation's pages in any order. Pages can be shown multiple times or not at all. The `getCustomPresentations()` method returns a custom presentations object that contains all custom presentations (see Table 248).

Table 248. Some methods supported by the *XCustomPresentationSupplier* interface.

Method	Description
<code>createInstance()</code>	Create a custom presentation.
<code>getByName(String)</code>	Get a custom presentation by name.
<code>getElementNames()</code>	Array of custom presentation names.
<code>hasByName(String)</code>	Return True if the custom presentation with the specified name exists.
<code>hasElements()</code>	Return True if the page contains at least one custom presentation.
<code>insertByName(String, CustomPresentation)</code>	Insert a custom presentation by name.
<code>removeByName(String)</code>	Remove the named custom presentation.
<code>replaceByName(String, CustomPresentation)</code>	Replace a custom presentation by name.

A custom presentation is a container for draw pages that supports the *XNamed* and *XIndexedAccess* interfaces. Create the custom presentation, add the draw pages in the order that you want them to appear, and then save the custom presentation. A custom presentation is displayed in exactly the same way that regular presentations are displayed, using a *Presentation* object, but the *CustomShow* attribute is set to reference the custom presentation.

Listing 517. Creating a custom presentation.

```
Sub CustomPresentation()
    Dim oPres    'Presentations, both customer and regular
    Dim oPages   'Draw pages

    oPres = ThisComponent.getCustomPresentations().createInstance()
    If NOT ThisComponent.getCustomPresentations().hasByName("custom") Then
        oPages = ThisComponent.getDrawPages()

        REM Display pages 0, 2, 1, 0
        oPres.insertByIndex(0, oPages.getByIndex(0))
        oPres.insertByIndex(1, oPages.getByIndex(2))
        oPres.insertByIndex(2, oPages.getByIndex(1))
        oPres.insertByIndex(3, oPages.getByIndex(0))
        ThisComponent.getCustomPresentations().insertByName("custom", oPres)
    End If

    REM Now, run the customer presentation.
    oPres = ThisComponent.getPresentation()
    oPres.CustomShow = "custom"
    oPres.Start()
End Sub
```

16.4.1. Presentation draw pages

Draw pages in a presentation document are slightly different from those in a drawing document. The properties in Table 249 dictate how and when page transitions occur while showing presentations.

Table 249. *Properties defined by the com.sun.star.presentation.DrawPage service.*

Property	Description
Change	Long Integer that specifies what causes a page change. 0 – A mouse-click triggers the next animation or page change. 1 – The page change is automatic. 2 – Object effects run automatically, but the user must click on the page to change it.
Duration	Long Integer time in seconds the page is shown if the Change property is set to 1.
Effect	Effect used to fade in or out (see Table 250).
Layout	Index of the presentation layout page if this is not zero.
Speed	Speed of the fade-in effect using the com.sun.star.presentation.AnimationSpeed enumeration: SLOW, MEDIUM, or FAST.

Page transitions are governed by the Effect property of the presentation draw page (see Table 250).

Table 250. *Values defined by the com.sun.star.presentation.FadeEffect enumeration.*

Values	Values	Values
NONE	VERTICAL_STRIPES	HORIZONTAL_STRIPES
DISSOLVE	VERTICAL_CHECKERBOARD	HORIZONTAL_CHECKERBOARD
RANDOM	VERTICAL_LINES	HORIZONTAL_LINES
FADE_FROM_LEFT	MOVE_FROM_LEFT	UNCOVER_TO_LEFT
FADE_FROM_TOP	MOVE_FROM_TOP	UNCOVER_TO_UPPERLEFT
FADE_FROM_RIGHT	MOVE_FROM_RIGHT	UNCOVER_TO_TOP
FADE_FROM_BOTTOM	MOVE_FROM_BOTTOM	UNCOVER_TO_UPPERRIGHT
FADE_FROM_UPPERLEFT	MOVE_FROM_UPPERLEFT	UNCOVER_TO_RIGHT
FADE_FROM_UPPERRIGHT	MOVE_FROM_UPPERRIGHT	UNCOVER_TO_LOWERRIGHT
FADE_FROM_LOWERLEFT	MOVE_FROM_LOWERRIGHT	UNCOVER_TO_BOTTOM
FADE_FROM_LOWERRIGHT	MOVE_FROM_LOWERLEFT	UNCOVER_TO_LOWERLEFT
FADE_TO_CENTER	ROLL_FROM_LEFT	CLOSE_VERTICAL
FADE_FROM_CENTER	ROLL_FROM_TOP	CLOSE_HORIZONTAL
CLOCKWISE	ROLL_FROM_RIGHT	OPEN_VERTICAL
COUNTERCLOCKWISE	ROLL_FROM_BOTTOM	OPEN_HORIZONTAL
STRETCH_FROM_LEFT	WAVYLINE_FROM_LEFT	SPIRALIN_LEFT
STRETCH_FROM_TOP	WAVYLINE_FROM_TOP	SPIRALIN_RIGHT
STRETCH_FROM_RIGHT	WAVYLINE_FROM_RIGHT	SPIRALOUT_LEFT
STRETCH_FROM_BOTTOM	WAVYLINE_FROM_BOTTOM	SPIRALOUT_RIGHT

The following macro sets the transitions on all draw pages to RANDOM.

Listing 518. Set Transition Effects to RANDOM.

```
Sub SetTransitionEffects()
    Dim oPages 'Draw pages
    Dim i%

    oPages = ThisComponent.getDrawPages()
    For i = 0 To oPages.getCount() - 1
        With oPages.getByIndex(i)
            .Effect = com.sun.star.presentation.FadeEffect.RANDOM
            .Change = 1
            .Duration = 2
            .Speed = com.sun.star.presentation.AnimationSpeed.FAST
        End With
    Next
End Sub
```

16.4.2. Presentation shapes

Shapes contained in Impress documents differ from shapes in Draw documents in that they support the `com.sun.star.presentation.Shape` service. The presentation Shape service provides properties that define special behavior to enhance presentations (see Table 251).

Table 251. Properties defined by the `com.sun.star.presentation.Shape` service.

Property	Description
Bookmark	Generic URL string used if the <code>OnClick</code> property requires a URL.
DimColor	Color for dimming this shape if <code>DimPrevious</code> is <code>True</code> and <code>DimHide</code> is <code>False</code> .
DimHide	If <code>True</code> and <code>DimPrevious</code> is <code>True</code> , the shape is hidden.
DimPrevious	If <code>True</code> , the shape is dimmed after executing its animation effect.
Effect	Animation effect for this shape (see Table 252).
IsEmptyPresentationObject	<code>True</code> if this is the default presentation shape and it is empty.
IsPresentationObject	<code>True</code> if this is a presentation object.
OnClick	Specify an action if the user clicks the shape (see Table 253).
PlayFull	If <code>True</code> , the sound of this shape is played in full.
PresentationOrder	Long Integer representing the order in which the shapes are animated.
Sound	URL string for a sound file that is played while the shape's animation is running.
SoundOn	If <code>True</code> , sound is played during the animation.
Speed	Speed of the fade-in effect using the <code>com.sun.star.presentation.AnimationSpeed</code> enumeration: <code>SLOW</code> , <code>MEDIUM</code> , or <code>FAST</code> .
TextEffect	Animation effect for the text inside this shape (see Table 252).
Verb	Long Integer "ole2" verb if the <code>ClickAction</code> is <code>VERB</code> .

The animation effects supported by shapes (see Table 250) are similar to, but more plentiful than, the animation effects supported by draw pages (see Table 252).

Table 252. Values defined by the *com.sun.star.presentation.AnimationEffect* enumeration.

Property	Property	Property
NONE	DISSOLVE	CLOCKWISE
RANDOM	APPEAR	COUNTERCLOCKWISE
PATH	HIDE	
MOVE_FROM_LEFT	MOVE_TO_LEFT	MOVE_SHORT_TO_LEFT
MOVE_FROM_TOP	MOVE_TO_TOP	MOVE_SHORT_TO_TOP
MOVE_FROM_RIGHT	MOVE_TO_RIGHT	MOVE_SHORT_TO_RIGHT
MOVE_FROM_BOTTOM	MOVE_TO_BOTTOM	MOVE_SHORT_TO_BOTTOM
MOVE_FROM_UPPERLEFT	MOVE_TO_UPPERLEFT	MOVE_SHORT_TO_UPPERLEFT
MOVE_FROM_UPPERRIGHT	MOVE_TO_UPPERRIGHT	MOVE_SHORT_TO_UPPERRIGHT
MOVE_FROM_LOWERRIGHT	MOVE_TO_LOWERRIGHT	MOVE_SHORT_TO_LOWERRIGHT
MOVE_FROM_LOWERLEFT	MOVE_TO_LOWERLEFT	MOVE_SHORT_TO_LOWERLEFT
MOVE_SHORT_FROM_LEFT	LASER_FROM_LEFT	STRETCH_FROM_LEFT
MOVE_SHORT_FROM_TOP	LASER_FROM_TOP	STRETCH_FROM_UPPERLEFT
MOVE_SHORT_FROM_RIGHT	LASER_FROM_RIGHT	STRETCH_FROM_TOP
MOVE_SHORT_FROM_BOTTOM	LASER_FROM_BOTTOM	STRETCH_FROM_UPPERRIGHT
MOVE_SHORT_FROM_UPPERLEFT	LASER_FROM_UPPERLEFT	STRETCH_FROM_RIGHT
MOVE_SHORT_FROM_UPPERRIGHT	LASER_FROM_UPPERRIGHT	STRETCH_FROM_LOWERRIGHT
MOVE_SHORT_FROM_LOWERRIGHT	LASER_FROM_LOWERLEFT	STRETCH_FROM_BOTTOM
MOVE_SHORT_FROM_LOWERLEFT	LASER_FROM_LOWERRIGHT	STRETCH_FROM_LOWERLEFT
ZOOM_IN_FROM_LEFT	ZOOM_OUT_FROM_LEFT	FADE_FROM_LEFT
ZOOM_IN_FROM_TOP	ZOOM_OUT_FROM_TOP	FADE_FROM_TOP
ZOOM_IN_FROM_RIGHT	ZOOM_OUT_FROM_RIGHT	FADE_FROM_RIGHT
ZOOM_IN_FROM_BOTTOM	ZOOM_OUT_FROM_BOTTOM	FADE_FROM_BOTTOM
ZOOM_IN_FROM_CENTER	ZOOM_OUT_FROM_CENTER	FADE_FROM_CENTER
ZOOM_IN_FROM_UPPERLEFT	ZOOM_OUT_FROM_UPPERLEFT	FADE_FROM_UPPERLEFT
ZOOM_IN_FROM_UPPERRIGHT	ZOOM_OUT_FROM_UPPERRIGHT	FADE_FROM_UPPERRIGHT
ZOOM_IN_FROM_LOWERRIGHT	ZOOM_OUT_FROM_LOWERRIGHT	FADE_FROM_LOWERLEFT
ZOOM_IN_FROM_LOWERLEFT	ZOOM_OUT_FROM_LOWERLEFT	FADE_FROM_LOWERRIGHT
		FADE_TO_CENTER
ZOOM_IN	VERTICAL_CHECKERBOARD	VERTICAL_STRIPES
ZOOM_IN_SMALL	HORIZONTAL_CHECKERBOARD	HORIZONTAL_STRIPES
ZOOM_IN_SPIRAL	HORIZONTAL_ROTATE	VERTICAL_LINES
ZOOM_OUT	VERTICAL_ROTATE	HORIZONTAL_LINES
ZOOM_OUT_SMALL	HORIZONTAL_STRETCH	
ZOOM_OUT_SPIRAL	VERTICAL_STRETCH	
WAVYLINE_FROM_LEFT	SPIRALIN_LEFT	CLOSE_VERTICAL
WAVYLINE_FROM_TOP	SPIRALIN_RIGHT	CLOSE_HORIZONTAL
WAVYLINE_FROM_RIGHT	SPIRALOUT_LEFT	OPEN_VERTICAL
WAVYLINE_FROM_BOTTOM	SPIRALOUT_RIGHT	OPEN_HORIZONTAL

Shapes contained in presentation documents support special actions by setting the shape's **OnClick** property.

Table 253. *Values defined by the com.sun.star.presentation.ClickAction enumeration.*

Property	Description
NONE	No action is performed.
PREVPAGE	Jump to the previous page.
NEXTPAGE	Jump to the next page.
FIRSTPAGE	Jump to the first page.
LASTPAGE	Jump to the last page.
BOOKMARK	Jump to a bookmark specified by the Bookmark property in Table 251.
DOCUMENT	Jump to another document specified by the Bookmark property in Table 251.
INVISIBLE	The object becomes invisible.
SOUND	Play a sound specified by the Bookmark property in Table 251.
VERB	An OLE verb is performed as specified by the Verb property in Table 251. An OLE object supports actions called verbs. An OLE object that displays a video clip might support the verb “play,” for example.
VANISH	The object vanishes.
PROGRAM	Execute another program specified by the Bookmark property in Table 251.
MACRO	Execute a Star Basic macro specified by the Bookmark property in Table 251.
STOPPRESENTATION	Stop the presentation.

16.5. Conclusion

Impress and Draw documents contain numerous features supporting drawing and graphics in documents. Both types of documents have many drawing and graphical presentation features in common, and Impress documents facilitate the construction of graphic presentations with support for manual or automatic page presentation. Although these documents support the display of bit-mapped images, their strength is vector drawings rather than photographic images. The results possible with Impress and Draw documents range from simple to quite complex. Consider this chapter as only a starting point for your explorations on the capabilities of these two document types.

17. Library Management

This chapter discusses how and where macro libraries are stored, along with methods you can use to manipulate them. This chapter also covers the subtleties of the macro organizer as well as using the UNO API to manipulate libraries and modules.

This chapter assumes a basic understanding of library containers, libraries, and modules. If the following summary is not familiar, review the material presented in Chapter 2, “Getting Started.”

- A library container contains zero or more libraries.
- Each library contains zero or more modules and dialogs.
- Each module contains zero or more macros.
- The application is a library container. Libraries stored in the application are globally available to all macros.
- Every document is a library container.
- The library named Standard is special; it always exists and cannot be overwritten. I recommend against using the Standard library.
- Always give meaningful names to the libraries and modules that you create. For example, Library1 and Module4 are not meaningful names, although AXONInvoiceForm1 might be more descriptive and helpful.

17.1. Accessing libraries using Basic

I packaged some of the macros that I use into an application-level library with the name “Pitonyak.” The next day, I started OpenOffice.org, and to my surprise, I was not able to use any macros in my new library. It turns out that libraries must be loaded before they are available.

TIP A library must be loaded before it is available.

To manually load a library, open the Macro Organizer dialog (Tools | Macros | Macro) and double-click the library that you want to load. You can also access libraries by using a macro. Basic provides the variable `GlobalScope` to manipulate application-level libraries (see Listing 519).

Listing 519. Use `GlobalScope` to load application-level libraries.

```
GlobalScope.BasicLibraries.loadLibrary("Pitonyak")
```

TIP The `GlobalScope` variable does not support the “dbg” properties. In other words, you cannot inspect the `GlobalScope` variable.

Libraries contain two things—dialogs and macros. The `GlobalScope` variable has two properties—`BasicLibraries` and `DialogLibraries`—that provide access to the Basic and dialog library containers in the application library container. The `BasicLibraries` and `DialogLibraries` properties both support the same set of interfaces for accessing the contained library containers (see Table 254).

Table 254. Methods supported by library container objects.

Method	Description
createLibrary(Name)	Create a new library with the given name.
createLibraryLink(Name, Url, ReadOnly)	Create a link to an “external” library. The Name and URL are both strings. If the ReadOnly flag is True, the library cannot be modified.
removeLibrary(Name)	Remove the named library. If the library is a link, only the link is removed, not the actual library.
isLibraryLoaded(Name)	True if the library is loaded, False otherwise.
loadLibrary(Name)	Load a library if it is not already loaded.
isLibraryLink(Name)	True if the library is a link to another library.
getLibraryLinkURL(Name)	Return the URL of a linked library. An error occurs if the library is not a link.
isLibraryReadOnly(Name)	Return True if the library is read-only.
setLibraryReadOnly(Name, ReadOnly)	Set the named library to read-only if ReadOnly is True.
renameLibrary(Name, NewName)	Rename a library. If the library is linked, only the link is renamed.
changeLibraryPassword(Name, Pass, NewPass)	Change a library’s password.
getByName(Name)	Get the named library.
getElementNames()	Get an array of library names.
hasByName(Name)	True, if the named library exists.
hasElements()	True, if at least one library exists.
isLibraryPasswordProtected(Name)	True, if the library is password protected.
isLibraryPasswordVerified(Name)	True, if the password has already been used to unlock a library.
verifyLibraryPassword(Name, Pass)	Unlock a password-protected library.

TIP

The GlobalScope variable is not available outside of Basic. It is, however, possible to create and use the undocumented UNO service `com.sun.star.script.ApplicationScriptLibraryContainer` to access the global Basic libraries.

The contained libraries are stored as XML strings inside named containers. Table 255 contains the methods supported by library objects.

Table 255. Methods supported by library objects.

Method	Description
getByName(Name)	Get the named module as a String.
getElementNames()	Get an array of module names.
hasByName(Name)	True, if the library contains the named module.
hasElements()	True, if the library contains at least one module.
insertByName(Name, Module)	Insert the named module into the library.
removeByName(Name)	Remove the named module.
replaceByName(Name, Module)	Replace the named module.

When you use the macro organizer to create a library, it automatically creates both a dialog library and a Basic library. The newly created Basic library contains the module named “Module1,” which contains an empty subroutine named “Main.” The following steps illustrate the process:

1. Open the Macro dialog using **Tools | Macros | Organize Macros | OpenOffice.org Basic**.
2. Click the **Organizer** button to open the *Basic Macro Organizer* dialog. The organizer dialog can also be opened using **Tools | Macros | Organize Dialogs**.
3. Click the *Libraries* tab.
4. Click the **New** button and give the library the name “TestLib” (see Figure 128).
5. Click **OK** to close the *New Library* dialog.
6. Click the **Close** button on the *Basic Macro Organizer* dialog.
7. Click the **Close** button on the *Basic Macros* dialog.

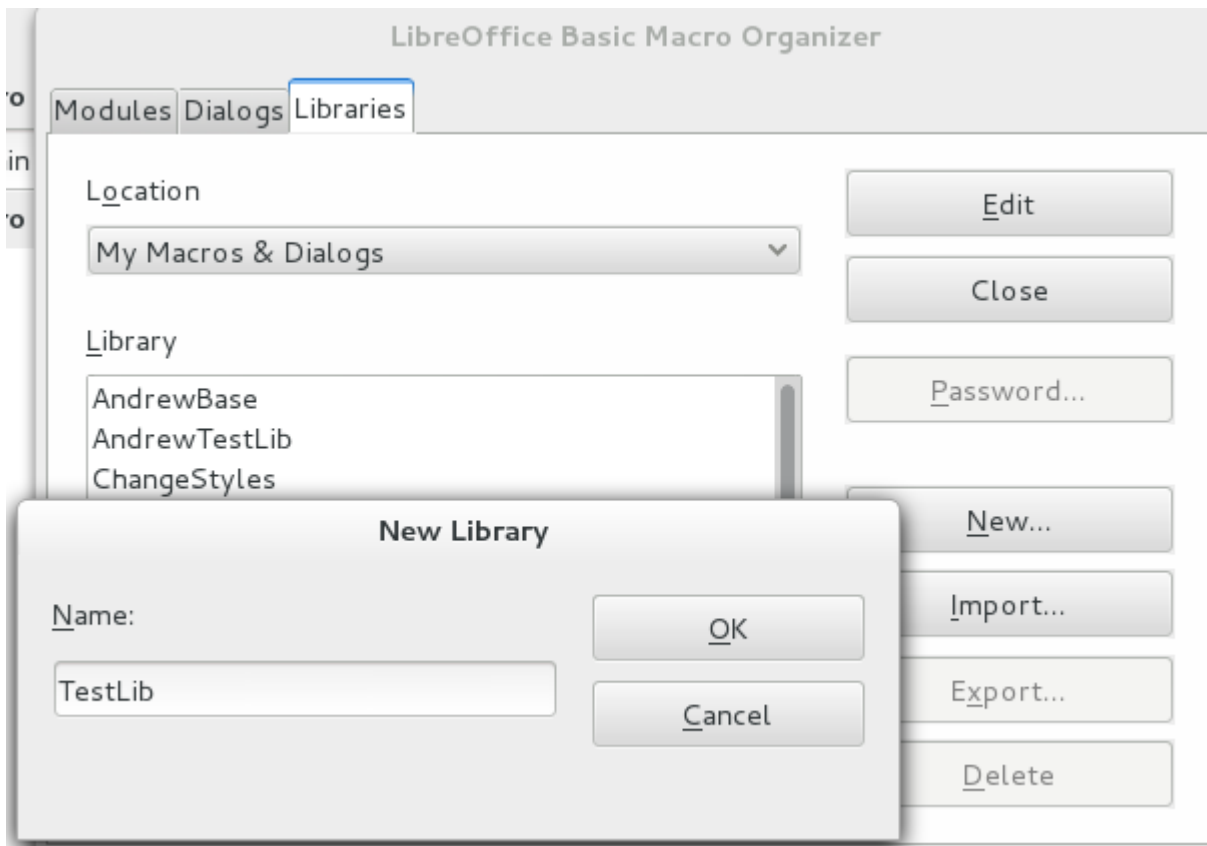


Figure 128. Creating a new library in the application library container.

To see the contents of the TestLib library, enter and run the macro in Listing 520. As seen in Figure 129, TestLib contains a Module1 with a single subroutine.

Listing 520. Print Module1 in the newly created TestLib library.

```
Sub PrintTestLib
    If GlobalScope.BasicLibraries.hasByName("TestLib") Then
        Dim oLib
        oLib = GlobalScope.BasicLibraries.getByName("TestLib")
        MsgBox oLib.getByName("Module1"), 0, "Module1 in TestLib"
```

```

Else
    MsgBox "Library 'TestLib' does not exist"
End If
End Sub

```

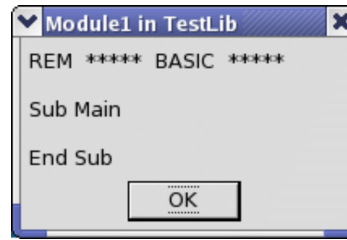


Figure 129. Module1 is automatically created by the dialog organizer.

You can use the code in Listing 521 to verify that both a Basic and a dialog library exist. Although the dialog library exists, it does not contain any dialogs. If the TestLib library does not exist, the returned library is null.

Listing 521. Obtain the Basic and the dialog library.

```

oLib = GlobalScope.BasicLibraries.getByName("TestLib")
oLib = GlobalScope.DialogLibraries.getByName("TestLib")

```

TIP

Libraries created using the methods in Table 255 do not automatically create both a dialog and a macro library.

Libraries created using the UNO API in Table 255 cause only one library to be created. The macro in Listing 522 creates a dialog library and a corresponding Basic library, adds Module1 to the code library, and adds a subroutine to Module1. See Figure 130.

Listing 522. Create A Global Library.

```

Sub CreateAGlobalLib
    Dim oLib
    Dim s$
    If GlobalScope.BasicLibraries.hasByName("TestLib") Then
        GlobalScope.BasicLibraries.removeLibrary("TestLib")
        GlobalScope.DialogLibraries.removeLibrary("TestLib")
        MsgBox "Deleted TestLib"
    Else
        GlobalScope.BasicLibraries.createLibrary("TestLib")
        GlobalScope.DialogLibraries.createLibrary("TestLib")
        oLib = GlobalScope.BasicLibraries.getByName("TestLib")
        s = "Sub Main" & CHR$(10) & _
            "  x = x + 1" & CHR$(10) & _
            "End Sub"
        oLib.insertByName("Module1", s)
        s = "=== Basic Libs ===" & CHR$(10)
        oLib = GlobalScope.BasicLibraries.getByName("TestLib")
        s = s & Join(oLib.getElementNames(), CHR$(10))
        oLib = GlobalScope.DialogLibraries.getByName("TestLib")
        s = s & CHR$(10) & CHR$(10) & "=== Dialog Libs ===" & CHR$(10)
    End If
End Sub

```

```

s = s & Join(oLib.getElementNames(), CHR$(10))
MsgBox s, 0, "Modules in the TestLib Library"
oLib = GlobalScope.BasicLibraries.getByname("TestLib")
MsgBox oLib.getByname("Module1"), 0, "Module1 in TestLib"
End If
End Sub

```

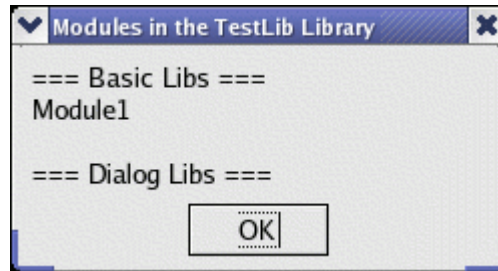


Figure 130. Although the dialog library exists, it contains no dialogs.

17.2. Libraries contained in a document

Basic defines the variables `BasicLibraries` and `DialogLibraries` to access libraries contained in a document. The behavior is the same as their `GlobalScope` counterparts except that they access the library container for the current document. Modify Listing 522 by removing all references to `GlobalScope`, and the macro acts on the current document.

Although the `BasicLibraries` and `DialogLibraries` variables provided by Basic provide an excellent method of accessing dialogs and libraries contained in a document, you may find code that uses the deprecated method `getLibraryContainer()`. Each document supports the method `getLibraryContainer()`. The returned object supports the method `getModuleContainer()`, which returns the contained Basic modules, and the method `getDialogContainer()`, which in my testing always returns a null object. Table 256 shows how to obtain a module using old and new methods.

Table 256. Obtain the module `aMod` from the library `TestLib` contained in a document.

Using <code>BasicLibraries</code>	Using <code>oDoc.getLibraryContainer()</code>
<pre> oLibs = BasicLibraries If oLibs.hasByName("TestLib") Then oLib = oLibs.getByname("TestLib") If oLib.hasByName("AMod") Then oMod = oLib.getByname("AMod") End If End If </pre>	<pre> oLibs = oDoc.getLibraryContainer() If oLibs.hasByName("TestLib") Then oLib = oLibs.getByname("TestLib") oMods = oLib.getModuleContainer() If NOT IsNull(oMods) Then If oMods.hasByName("AMod") Then oMod = oMods.getByname("AMod") End If End If End If </pre>

TIP

The `BasicLibraries` and `DialogLibraries` variables are only available if the currently running subroutine is contained in the document. Unfortunately, this leaves no way to obtain a dialog from a document except from code contained in the document. Write a dialog creation function that is contained in the document containing the dialog in question.

17.3. Writing an installer

I make no attempt at writing a full-fledged installer, but I do provide some clues here as to how to do it. The macro in Listing 523 copies a single library, whose name may be changed when it is copied. The process is simple and straightforward. The modules in the source library are copied one at a time into the destination library. The final argument determines if the destination library is cleared before the copy begins. If the destination library is not cleared first and it contains modules that do not exist in the source library, those modules will still exist in the destination library after the library is copied.

Listing 523. Copy a library.

```
REM sSrcLib is the name of the source library contained in oSrcLibs
REM sDestLib is the name of the destination library contained in oDestLibs
REM oSrcLibs is the source library container
REM oDestLibs is the destination library container
REM if bClearDest is True, then the destination library is cleared
Sub AddOneLib(sSrcLib$, sDestLib$, oSrcLibs, oDestLibs, bClearDest As Boolean)
    Dim oSrcLib    'The source library to copy
    Dim oDestLib    'The destination library to receive the modules in oSrcLib
    Dim sNames
    Dim i%

    REM If there is no destination library then simply return
    If IsNull(oDestLibs) OR IsEmpty(oDestLibs) Then
        Exit Sub
    End If

    REM Clear the destination library if requested
    If bClearDest AND oDestLibs.hasByName(sDestLib) Then
        oDestLibs.removeLibrary(sDestLib)
    End If

    REM If there is no source library, then there is nothing else to do
    If IsNull(oSrcLibs) OR IsEmpty(oSrcLibs) Then
        Exit Sub
    End If

    REM If the source library does not exist, then there is nothing else to do
    If NOT oSrcLibs.hasByName(sSrcLib) Then
        Exit Sub
    End If

    REM If the destination library does not exist, then create it
    If NOT oDestLibs.hasByName(sDestLib) Then
        oDestLibs.createLibrary(sDestLib)
    End If

    REM This is where the real fun begins!
    REM It may seem obvious, but the libraries must be loaded first.
    REM Common mistake to not load the libraries first!
    oSrcLibs.loadLibrary(sSrcLib)
```

```

oDestLibs.loadLibrary(sDestLib)

REM Get the source and destination libraries
REM Get all of the contained modules that should be copied
oSrcLib = oSrcLibs.getByName(sSrcLib)
oDestLib = oDestLibs.getByName(sDestLib)
sNames = oSrcLib.getElementNames()

REM For each module, either add it or replace it
For i = LBound(sNames) To UBound(sNames)
    If oDestLib.hasByName(sNames(i)) Then
        oDestLib.replaceByName(sNames(i), oSrcLib.getByName(sNames(i)))
    Else
        oDestLib.insertByName(sNames(i), oSrcLib.getByName(sNames(i)))
    End If
Next
End Sub

```

Assume that you want to copy a specific library from the application into a document so that you can send it to a friend. The fastest method is to simply use the macro organizer and append the library from the application into the document. Sure, you can write a macro, but some things are done faster manually. If, on the other hand, you need to copy libraries frequently, it makes sense to write a macro to do the work. The macro in Listing 524 accepts the name of a library contained in the application and then copies the library into the current document. Both the code library and the dialog library are copied. The analogous macro to copy a library from the document back to the application is trivial and nearly identical to Listing 524.

Listing 524. *Add a global library to to the current document.*

```

Sub AppLibToDocLib(sLibName$)
    Dim oGlobalLib
    oGlobalLib = GlobalScope.BasicLibraries
    AddOneLib(sLibName, sLibName, oGlobalLib, BasicLibraries, True)
    oGlobalLib = GlobalScope.DialogLibraries
    AddOneLib(sLibName, sLibName, oGlobalLib, DialogLibraries, True)
End Sub

```

17.4. Conclusion

Although sometimes it's quicker to manipulate libraries by hand, it's easy to manipulate and copy them using macros. The methods outlined in this chapter will allow you to perform most of the operations that you need for libraries.

18. Dialogs and Controls

This chapter discusses how to create and use dialogs and the controls that they contain. It focuses on the Basic IDE as the primary method of creating dialogs. This chapter covers each of the different controls and provides examples for most of the control types. It also includes a method to build dialogs and controls at run time, rather than using the Basic IDE.

In computer terms, a window is an area shown on the computer screen in which information is displayed. In most applications, it isn't always easy to tell the difference between a dialog and a window. Although the word "dialog" has multiple definitions, the definition usually specifies an exchange of information between two entities—usually people. In OpenOffice, a dialog is a window that is displayed to interact with a user.

A "modal window" blocks its parent application (or window) from further activity until the window has completed executing its code and has been closed successfully. In other words, when you open a "modal" window, it has the focus; you can't just move it out of the way and use the window underneath it. If a window is modal, it's usually considered to be a dialog.

TIP

While a modal dialog is displayed, you can access the dialog but not the document window. Dialogs are almost always modal, and general application windows, such as those displaying documents under user control, are almost never modal

The Find & Replace dialog in a Writer document is an example of a non-modal dialog. After opening the Find & Replace dialog, you can still access and edit the document. Another criterion for a window being classified as a dialog is based on function. The primary display window for a program is typically not a dialog. The information displayed or requested in a dialog is usually secondary compared to the function of the program. For example, configuration information and error notices are frequently displayed in a dialog, but the primary text of a Writer document is not. Although the difference between a window and a dialog is somewhat arbitrary, you can create and use your own dialogs in Basic, but you can't open and start a general application window. This is reasonable, because the usual purpose of a dialog is to communicate some details relevant to the configuration of an application, object, or document that is already active.

18.1. My first dialog

To create a dialog in the IDE, you must first create an empty dialog. You can use one of two methods to add an empty dialog to a library. The first method is to use the Macro Organizer dialog. The advantage of using the Macro Organizer is that you can immediately provide a meaningful name for the dialog.

You can also create dialogs and modules directly in the Basic IDE by right-clicking the Module tab. The tabs shown at the bottom of the Basic IDE identify the opened modules and dialogs. Place the cursor on any tab, right-click and select **Insert | BASIC Dialog** (see Figure 131).

A new dialog is created with the name Dialog1. Right-click the Dialog1 tab and choose Rename to change the name of the dialog from Dialog1 to HelloDlg. An empty dialog appears in the middle of the IDE page. This dialog, which is currently empty, is the one that will be displayed when you are finished. To select the dialog, click in its lower right corner. After you've selected the dialog, you can use the mouse to change its size or location. If you had selected the dialog by clicking anywhere in it, the more common activity of selecting controls contained in the dialog would be more difficult.

TIP When something is selected, green boxes appear around the selected item. The green boxes are called handles.

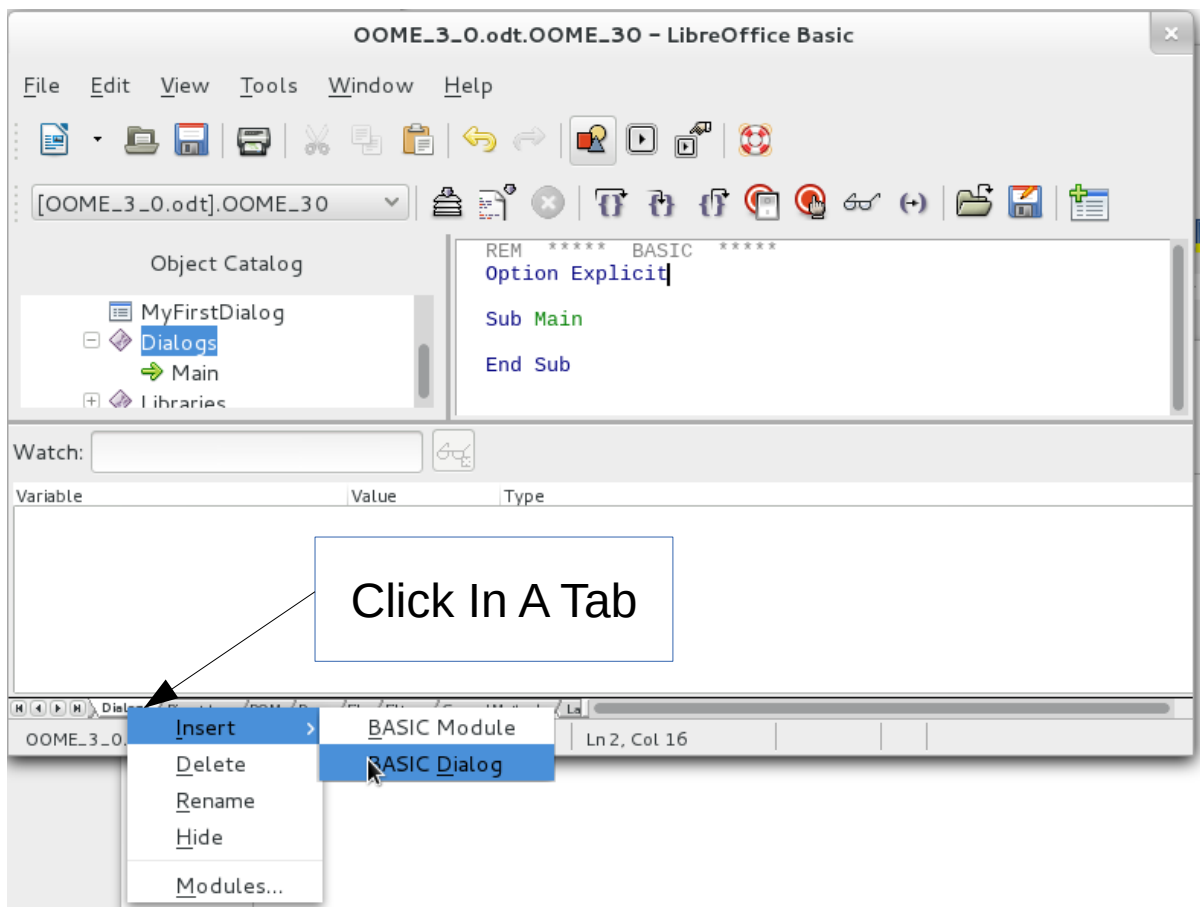


Figure 131. Right-click the module tab and select *Insert | BASIC Dialog*.

Click the Controls Dialog icon in the toolbar to open the Controls toolbar (see Figure 132). The Controls toolbar is non-modal, so you can leave it open and still manipulate the dialog. Click the lower right Test Mode button to open a copy of the dialog in test mode. This allows you to see how the dialog looks. To close the dialog after opening it in test mode, click the Close icon in the upper right corner of the dialog.

TIP The Controls toolbar is sometimes referred to as the “design tools window” or “Toolbox”, differs in appearance and what it contains between LO and AOO.

Most of the icons in the Controls toolbar are used to add controls to your dialog. If you click any of the control icons in the Controls toolbar, the cursor changes from an arrow-shaped selection cursor to a cross-shaped cursor. You can click the Select icon to change the cursor back to the selection shape rather than insert a new control.

Table 257. Dialog controls.

AOO	LO	Description
		Select
		Manage Language
		Test mode on / off.
		Properties
		Button
		Image control
		Check box
		Radio button
		Label field
		Text box
		List box
		Combo box
		Vertical scroll bar
		Horizontal scroll bar
		Group box
		Progress bar
		Horizontal line
		Vertical line
		Date field
		Time field

		Numeric field
		Currency field
		Formatted field
		Pattern field
		File selection
		Tree control
		Spin button is missing (by default) on the Toolbox bar in AOO and it is not usable in an AOO dialog anyway (I think).

TIP

Other controls do exist, but they are not shown in the Toolbox bar by default; for example, database controls such as a Table control. Just as well since they are not usable in a dialog.

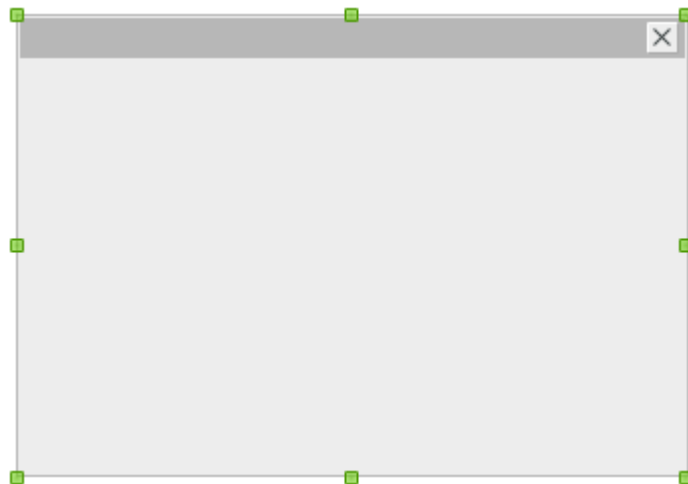


Figure 132. Click the lower right corner to enable the green handles.

18.1.1. The Properties dialog

Much of what a dialog or control does is controlled by properties, which you can set at both run time and design time. To set properties at design time, first select the object. After the object is selected, either right-click the object and select Properties, or click the Properties button in the Controls dialog (see Figure 133).

The Name property (see Figure 133) sets the name used to retrieve the control from the dialog. For a dialog, there is no reason to change the name, because the dialog is not retrieved in the same way that a control is retrieved. The Title property exists for dialogs (and only for dialogs), and it

sets the value of the dialog's title—I set the title for my example dialog to “Hello World.” Although the Properties dialog supports setting the position and size, this is usually done using the mouse.

The screenshot shows the 'General' tab of a 'Properties' dialog box. The 'Name' is 'HelloDlg'. The 'Title' is empty. 'Enabled' is 'Yes'. 'Page (step)', 'Tab order', 'Scroll width', 'Scroll height', 'Scroll top', and 'Scroll left' are all '0'. 'PositionX' is '9' and 'PositionY' is '80'. 'Width' is '120' and 'Height' is '83'. 'Font' is '(Default)' with a browse button (...). 'Background color' is 'Default' with a dropdown and a browse button (...). 'Scrollbars' is 'None' with a dropdown. 'Graphics' is empty with a dropdown and a browse button (...). 'Additional information...', 'Help text', and 'Help URL' are empty text boxes. 'With title bar' is 'Yes' with a dropdown.

Property	Value
Name	HelloDlg
Title	
Enabled	Yes
Page (step)	0
Tab order	0
Scroll width	0
Scroll height	0
Scroll top	0
Scroll left	0
PositionX	9
PositionY	80
Width	120
Height	83
Font	(Default)
Background color	Default
Scrollbars	None
Graphics	
Additional information	
Help text	
Help URL	
With title bar	Yes

Figure 133. Set the properties for the dialog.

TIP

The properties shown in the Properties dialog change based on the selected object

Use the following steps to set up the dialog in this example.

1. Use **View > Toolbars > Form Controls** to open the Controls toolbar.

2. Select the dialog by clicking the lower right corner of the dialog (see Figure 132).
3. Click the Properties icon on the Controls toolbar to open the Properties dialog (see Figure 133).
4. Set the dialog's name to HelloDlg—I like to name things.
5. Set the dialog's title to “Hello World.” When the cursor leaves the Title property field, the title is displayed in the dialog's title bar.
6. Make certain that the Properties dialog is not covering the Hello World dialog.
7. Click the Button icon in the Controls toolbar to add a button to the Hello World dialog. When the cursor moves over the dialog, it changes from the selection arrow to a cross. Position the cursor where you want the button, and then click the mouse button and drag to define the appropriate size.
8. When the button is created, it defaults to the name CommandButton1. Use the Properties dialog to change the button's name to OKButton and the button's label to “OK”.
9. Use the Properties dialog to change the button type to OK so that the button will automatically close the dialog. If you don't do this, the button will do nothing unless a handler is registered for the button—event handlers are set from the Events tab in the Properties dialog.
10. In the “Help text” field, enter the text “Click Here” so that some help text is displayed when the cursor rests over the button.

TIP

OpenOffice.org includes numerous excellent examples of dialog and control manipulation in the module named ModuleControls contained in the Tools library.

18.1.2. Starting a dialog from a macro

Dialogs can be stored in the document or in a global library (see Chapter 16, “Library Management”). Regardless of where the dialog is stored, it must be loaded before it can be used. Remember that you can access the dialog libraries stored in the current document by using DialogLibraries, and the libraries stored in the application by using GlobalScope.DialogLibraries. Use the following sequence of steps to load, create, and execute a dialog:

1. Use the loadLibrary() method to load the library containing the dialog. Although you can skip this step if the library is already loaded, it is probably not prudent to assume this.
2. Use the getByName() method to retrieve the library from DialogLibraries.
3. Use the method getByName() to retrieve the dialog from the retrieved library.
4. Create the dialog using CreateUnoDialog().
5. Execute the dialog.

Listing 525 loads and executes a dialog stored in the current document.

Listing 525. Load and run the test dialog.

```
Dim oHelloDlg      'The run-time dialog that is created
Sub RunHelloDlg
    Dim oLib        'Library that contains the dialog
    Dim oLibDlg     'Dialog as stored in the library

    REM First, load the library.
```

```

REM If the dialog is stored in an application-level library rather than
REM in the document, then GlobalScope.DialogLibraries must be used.
DialogLibraries.loadLibrary("OOME_40")

REM Obtain the entire library now that it has been loaded.
oLib = DialogLibraries.getByName("OOME_40")

REM Obtain the dialog as stored in the library.
REM I usually think of this as the definition of the dialog.
oLibDlg = oLib.getByName("HelloDlg")

REM Create a dialog that can be used.
oHelloDlg = CreateUnoDialog(oLibDlg)

REM Now start the dialog.
REM The execute() method is really a function that returns 1 if OK
REM is used to close the dialog and 0 if Cancel is used to close the
REM dialog. Clicking the Close button in the upper right corner
REM of the dialog is considered the same as clicking Cancel.
oHelloDlg.execute()
End Sub

```

TIP

The variable that contains the dialog is declared outside of the subroutine that creates the dialog so that the dialog can be accessed in event handlers, which are implemented as subroutines.

18.1.3. Assign an event handler

Controls and dialogs can call external event handlers. You can assign individual subroutines as event handlers by using the Events tab of the Properties dialog (see Figure 134). A quick look at the title shows that the events in Figure 134 are for a command button. The supported events can change based on the selected object.

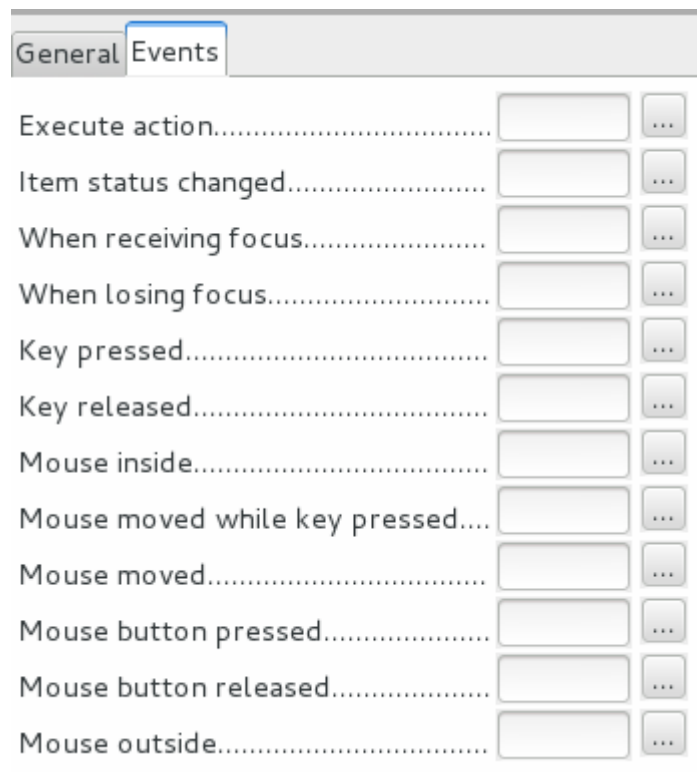


Figure 134. Events for a command button.

You can use an event handler to close the Hello World dialog. Use the Properties dialog to change the Button type from OK to default; the dialog will no longer close on its own. You might want to do this, for example, if you want to validate input before closing a dialog. Now, write the event handler that will close the dialog (see Listing 526).

Listing 526. Close the hello dialog.

```
Sub ExitHelloDlg
    oHelloDlg.endExecute()
End Sub
```

To assign the “Execute action” event to call the ExitHelloDlg macro, open the Properties dialog for the button and click the Events tab. Click the three dots to the right of the desired event (see Figure 134) to open the Assign action dialog (see Figure 135).

Warn

I have seen different behaviors depending on the OOo variant and version. In LO 4.0.2.2, if I click the dots for Execute action (see Figure 134) and then assign a macro to Item Status changed (see Figure 135), nothing was assigned. In other versions (not LO) I have assigned multiple actions and different actions and they worked.

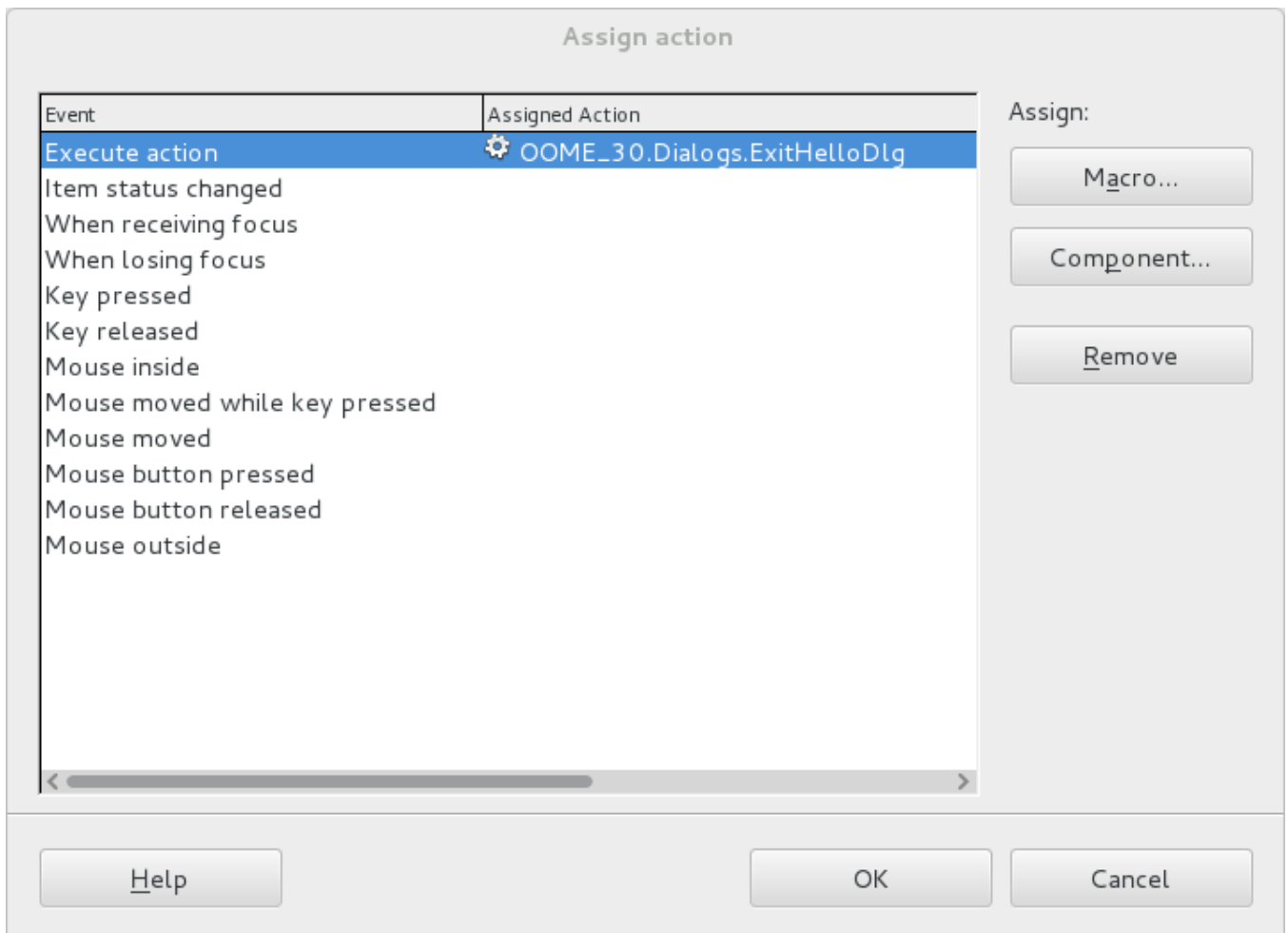


Figure 135. Assign macros to event handlers.

Warn To assign an event handler, you must click the Assign button and not the OK button. The only purpose for the OK button is to close the Assign Macro dialog. It is a common and frustrating mistake to select an event handler for an event, and then click OK rather than Assign.

Highlight event and click the **Macro** button to open the *Macro Selector* dialog.

TIP If the dialog will be used with an `XDialogEventHandler`, then use the Component button in the *Assign action* dialog (see Figure 135) and enter the name of the method to call.

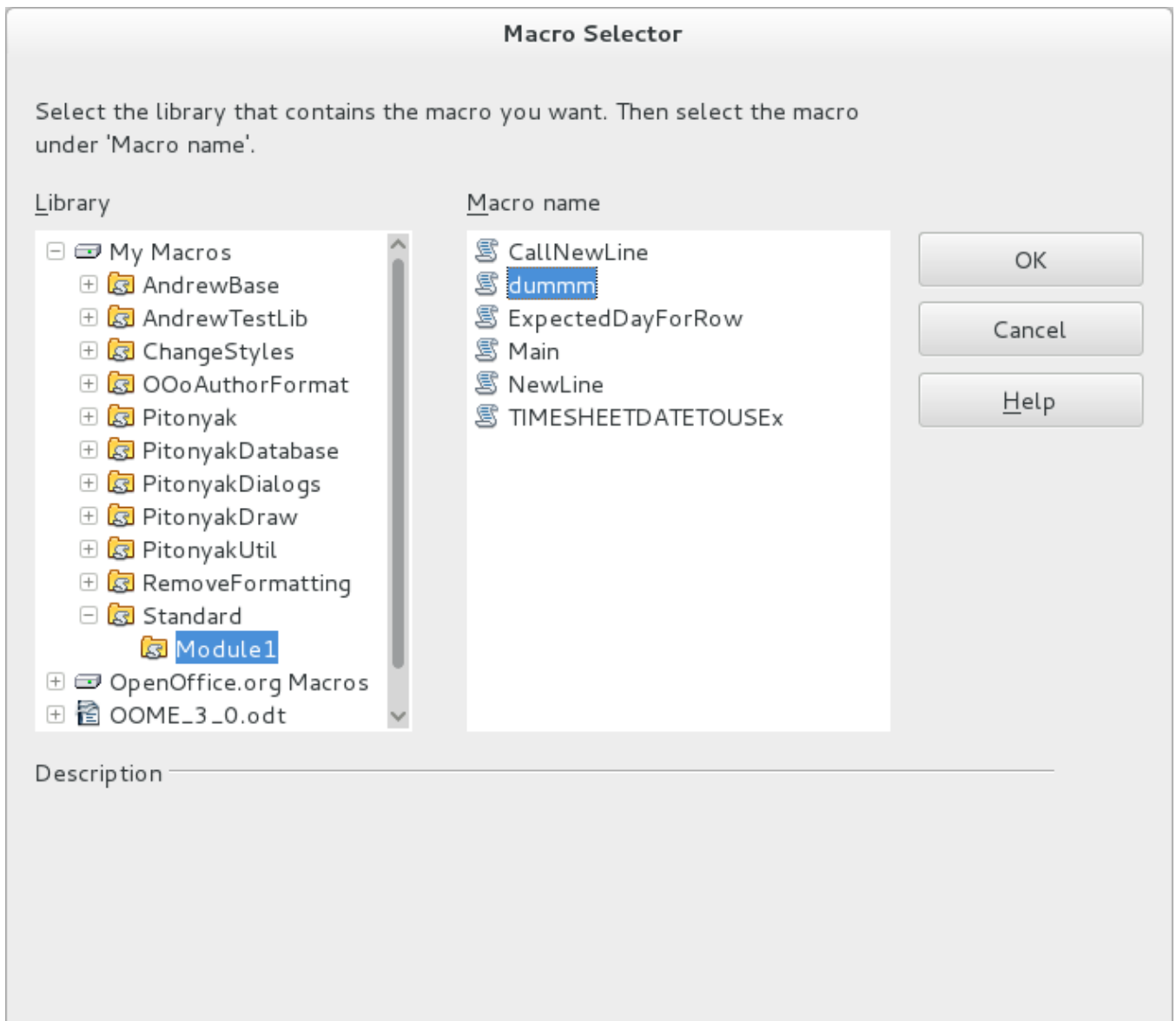


Figure 136. Assign macros to event handlers.

Use the *Library* tree control to select the macro library container (My Macros, application macros, or a document), the library, and module. Finally, select the desired macro in the *Library Name* list box.

18.2. Dialog and control paradigm

To understand how controls work, it is important that you understand the difference between a model, a view, and a controller. The model-view-control paradigm, sometimes referred to as MVC, is very widely used and accepted in the computer science community.

The *model* describes a control's appearance, data, and behavior. The model does not interact with a user and it is not able to display itself. When an object supports the MVC paradigm, the changes should be made to the object's model and then these changes will automatically be propagated to the views by the controller. Only one model can exist for an object.

The *view* is what the user sees. Although there can be only one model for an object, multiple views may exist to display an object.

The *controller* is what interacts with the user. When the user tells the controller to perform some action, a change is made to the model, and then the view is updated.

Note An OpenOffice.org document is distinguished from a non-document desktop component, such as the Basic IDE or the Help window, by the fact that it has a data model. In other words, OOO uses the MVC paradigm for documents.

The controls that are displayed in dialogs generally contain access methods in the view portion that update and affect the model. I recommend that you use the model rather than the view methods unless you have a compelling reason to do otherwise; I am told that it can lead to some inconsistencies. Apparently this problem is more pronounced with controls embedded in forms than in dialogs, but I have never observed the inconsistencies directly.

18.2.1. Dialog and control similarities

In some respects, a dialog is also a control. Dialogs support the MVC paradigm and share many similarities with controls. For example, dialogs support the UnoControl service, which is common to all controls. UNO controls support the XComponent interface, which is also supported by numerous other objects in OpenOffice.org (see Table 258).

Table 258. Methods defined by the com.sun.star.lang.XComponent interface.

Method	Description
dispose()	The object’s owner calls this method to free all the object’s resources. You won’t usually call this method.
addEventListener(XEventListener)	Add an event listener to the object.
removeEventListener(XEventListener)	Remove an event listener from the object’s listener list.

The XControl interface identifies an object as a control. All objects that support the UnoControl service also support the XControl interface. For Basic programmers, the most interesting method is probably getModel() because it returns the control’s data model (see Table 259).

Table 259. *Methods defined by the com.sun.star.awt.XControl interface.*

Method	Description
setContext(XInterface)	Set the control's context.
getContext()	Get the control's context.
createPeer(XToolkit, XWindowPeer)	Create a child window. If the peer is null, the desktop is the parent. This method is used to create a window to display a dialog that was created in a macro rather than in the IDE (see Listing 551).
getPeer()	Return the previously created or set peer XWindowPeer.
setModel(XControlModel)	Set the model's control; return True if successful.
getModel()	Return the control's XModel.
getView()	Return the control's XView.
setDesignMode(Boolean)	Turn the design mode on and off.
isDesignMode()	Return True if the control is in design mode; otherwise return False.
isTransparent()	Return True if the control is transparent; otherwise return False.

When you set a position and a size by using a single method (see Table 261), the PosSize constant group is used to control which portions are actually updated (see Table 260).

Table 260. *Constants defined by the com.sun.star.awt.PosSize constant group.*

Constant	Name	Description
1	X	Flag the X coordinate.
2	Y	Flag the Y coordinate.
4	WIDTH	Flag the width.
8	HEIGHT	Flag the height.
3	POS	Flag the X and Y coordinates.
12	SIZE	Flag the width and height.
15	POSSIZE	Flag everything.

A window is a rectangular region on an output device defined primarily by its position and size. The constants in Table 260 are used in the setPosSize() method shown in Table 261 to set the position and size. The XWindow interface defines the basic operations for a window component. The methods to get and set the position are demonstrated in Listing 543.

Table 261. *Methods defined by the com.sun.star.awt.XWindow interface.*

Method	Description
setPosSize (x, y, width, height, PosSize)	Set the window's outer bounds (see Table 260).
getPosSize()	Return the window's outer bounds as a rectangle.
setVisible(Boolean)	Show or hide the window.
setEnabled(Boolean)	Enable or disable the window.

Method	Description
setFocus()	Focus this window.
addWindowListener(listener)	Add the XWindowListener.
removeWindowListener(listener)	Remove the XWindowListener from the listener list.
addFocusListener(listener)	Add the XFocusListener.
removeFocusListener(listener)	Remove the XFocusListener from the listener list.
addKeyListener(listener)	Add the XKeyListener.
removeKeyListener(listener)	Remove the XKeyListener from the listener list.
addMouseListener(listener)	Add the XMouseListener.
removeMouseListener(listener)	Remove the XMouseListener from the listener list.
addMouseMotionListener(listener)	Add the XMouseMotionListener.
removeMouseMotionListener(listener)	Remove the XMouseMotionListener from the listener list.
addPaintListener(listener)	Add the XPaintListener.
removePaintListener(listener)	Remove the XPaintListener from the listener list.

The XView interface defines the methods required to display an object (see Table 262). These methods are used primarily by advanced users and are included here only for completeness.

Table 262. *Methods defined by the com.sun.star.awt.XView interface.*

Method	Description
setGraphics(XGraphics)	Set the output device.
getGraphics()	Return the XGraphics output device.
getSize()	Returns the size of the object in device units.
draw(x, y)	Draws the object at the specified position.
setZoom(x, y)	Set the zoom factor of the control's contents.

18.2.2. Dialog-specific methods

The methods defined in Table 258 through Table 262 are common to all control types. As expected, however, dialogs support methods and properties that are specific to dialogs (see Table 263).

Table 263. *Methods defined by the com.sun.star.awt.XDialog interface.*

Method	Description
setTitle()	Set the dialog's title.
getTitle()	Get the dialog's title.
execute()	Display the dialog.
endExecute()	Hide the dialog and cause the execute() method to return.

All controls act as a window: They are rectangular in shape, can receive focus, be enabled and disabled, and have listeners (see Table 261). Top-level windows add extra functionality because they are not contained in another window (see Table 264).

Table 264. *Methods defined by the `com.sun.star.awt.XTopWindow` interface.*

Method	Description
<code>addTopWindowListener(XTopWindowListener)</code>	Add a listener for this window.
<code>removeTopWindowListener(XTopWindowListener)</code>	Remove a listener so that it no longer receives events from this window.
<code>toFront()</code>	Move this window in front of all other windows.
<code>toBack()</code>	Move this window behind all other windows.
<code>setMenuBar(XMenuBar)</code>	Set a menu bar for this window.

The primary function of a dialog is to contain controls. The methods in Table 265 allow a dialog to add, remove, and get controls contained in a dialog. It is very common to extract a control from a dialog to read or set its value.

Table 265. *Methods defined by the `com.sun.star.awt.XControlContainer` interface.*

Method	Description
<code>setStatusText(String)</code>	Set the text in the container's status bar.
<code>getControls()</code>	Return all controls as an array of type <code>XControl</code> .
<code>getControl(name)</code>	Return the <code>XControl</code> with the specified name.
<code>addControl(name, XControl)</code>	Add the <code>XControl</code> to the container.
<code>removeControl(XControl)</code>	Remove the <code>XControl</code> from the container.

Controls are usually not added to or removed from the dialog. If you create a dialog using a macro, rather than designing the dialog in the IDE, you add control models to the dialog's model, rather than adding controls directly to the dialog. As usual, the work is done in the model and the changes are reflected in the view. The most-used method in Table 265 is `getControl(name)`.

18.2.3. The dialog model

You can obtain a control's model by using the method `getModel()` defined in Table 259. Dialogs support the `UnoControlDialogModel` service (see Table 266) that defines many of the properties that you can set from the Basic IDE (see Figure 133). You can set some properties directly from the dialog object—the `setTitle()` method in Table 263, for example—but it is better to modify the model directly.

Table 266. *Properties defined by the `com.sun.star.awt.UnoControlDialogModel` service.*

Property	Description
<code>BackgroundColor</code>	Dialog's background color as a Long Integer.
<code>Closeable</code>	If True, the dialog is closeable.

Property	Description
Enabled	If True, the dialog is enabled.
FontDescriptor	Structure that defines and describes the font for the text in the dialog's caption bar.
FontEmphasisMark	Constant that defines the type/position of the emphasis mark for text in dialog's caption bar.
FontRelief	Constant that specifies if the dialog's caption bar contains embossed or engraved text.
HelpText	Dialog's help text as a String.
HelpURL	Dialog's help URL as a String.
Moveable	If True, the dialog is moveable.
Sizeable	If True, the dialog is sizeable.
TextColor	Dialog's text color as a Long Integer.
TextLineColor	Dialog's text line color as a Long Integer.
Title	Text string displayed in the dialog's caption bar.

Controls that can be inserted into an UNO control dialog support the properties defined by the `UnoControlDialogElement` service (see Table 267).

Table 267. *Properties in the `com.sun.star.awt.UnoControlDialogElement` service.*

Property	Description
Height	Control's height.
Name	Control's name.
PositionX	Control's horizontal position.
PositionY	Control's vertical position.
Step	Control's step.
TabIndex	Control's tab index.
Tag	Control's tag.
Width	Control's width.

It is possible to create controls and dialogs by using a macro rather than designing them in the Basic IDE. The dialog's model supports the `createInstance()` method defined by the `com.sun.star.lang.XMultiServiceFactory` interface. When a macro creates a control to be inserted into a dialog, that control should be created by the dialog's model. A control that is created from the global service manager will not have the properties shown in Table 267. Creating controls by using a macro is an advanced topic that is discussed later in this chapter. The macro in Listing 527 shows the objects that a dialog model can create (see Figure 137).

Listing 527. *What services can the dialog model create?*

```
Sub ShowDialogServices
    Dim oLib          'Library that contains the dialog
    Dim oLibDlg       'Dialog as stored in the library
```

```

DialogLibraries.loadLibrary("OOME_40")
oLib = DialogLibraries.getByName("OOME_40")
oLibDlg = oLib.getByName("HelloDlg")
oHelloDlg = CreateUnoDialog(oLibDlg)
MsgBox Join(oHelloDlg.getModel().getAvailableServiceNames(), CHR$(10))
End Sub

```

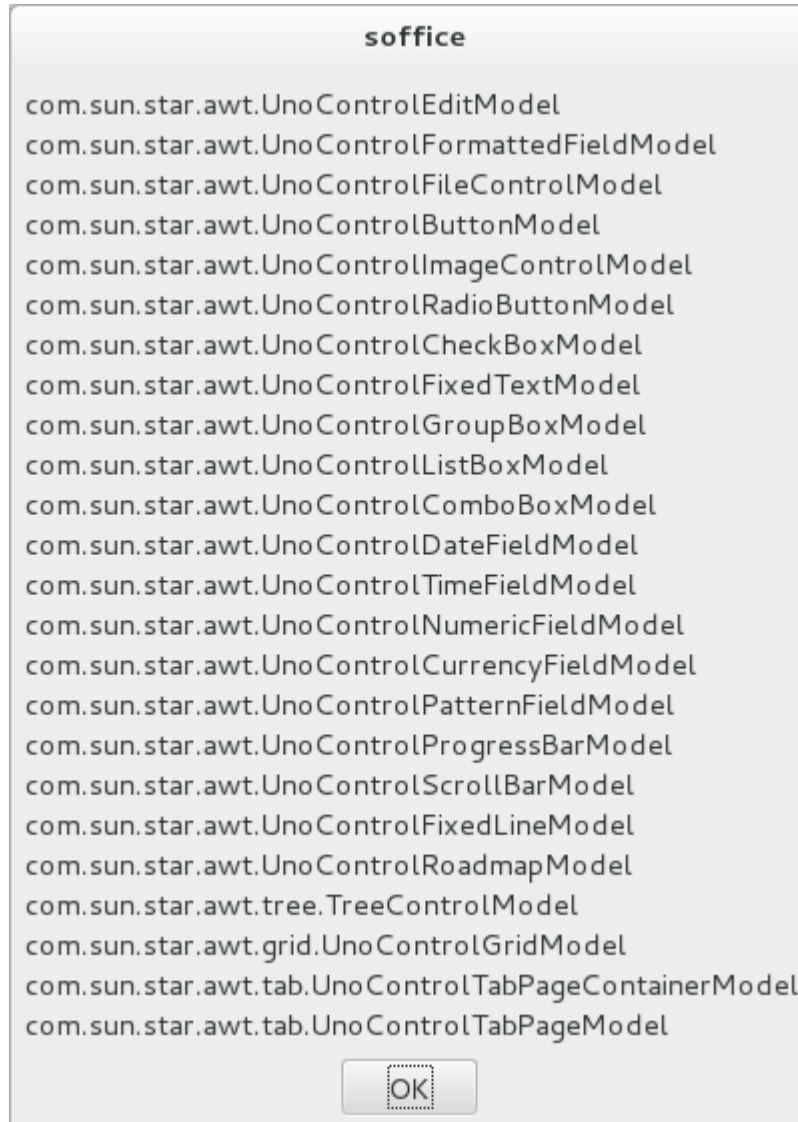


Figure 137. Services that the dialog model in AOO 3.4.1 can create.

18.3. Controls

You can obtain a control directly from a dialog by calling the `getControl()` method. You can obtain a control's model by calling the `getModel()` method on the control, or by calling `getByName()` on the model of the object in which it is contained. In other words, you can get a control's model from the dialog's model. The code in Listing 528 demonstrates the options available.

TIP

You can insert controls into dialogs and forms, but not all controls work in dialogs. Some controls require a data source, but a dialog is not able to contain a data source. This chapter deals only with controls that can be inserted into a dialog.

Listing 528. Get the button or the button's model.

```
oHelloDlg.getControl("OKButton")           'Get the UnoControlButton
oHelloDlg.getControl("OKButton").getModel() 'Get the UnoControlButtonModel
oHelloDlg.getModel().getByName("OKButton")  'Get the UnoControlButtonModel
```

The control types that can be inserted into a dialog are defined in the `com.sun.star.awt` module (see Table 268 and Figure 137). Notice the similarity between the service name of the control and the service name of the control's model.

Table 268. Controls and their models defined in the `com.sun.star.awt` module.

Control	Model	Description
TreeControl	TreeControlModel	
UnoControlButton	UnoControlButtonModel	Button
UnoControlCheckBox	UnoControlCheckBoxModel	Check box
UnoControlComboBox	UnoControlComboBoxModel	Combo box
UnoControlCurrencyField	UnoControlCurrencyFieldModel	Currency field
UnoControlDateField	UnoControlDateFieldModel	Date field
UnoControlDialog	UnoControlDialogModel	Dialog
UnoControlEdit	UnoControlEditModel	Text edit field
UnoControlFileControl	UnoControlFileControlModel	Select a file
UnoControlFixedLine	UnoControlFixedLineModel	Fixed line
UnoControlFixedText	UnoControlFixedTextModel	Display fixed
UnoControlFormattedField	UnoControlFormattedFieldModel	Formatted field
UnoControlGroupBox	UnoControlGroupBoxModel	Group box
UnoControlImageControl	UnoControlImageControlModel	Display an image
UnoControlListBox	UnoControlListBoxModel	List box
UnoControlNumericField	UnoControlNumericFieldModel	Numeric field
UnoControlPatternField	UnoControlPatternFieldModel	Pattern field
UnoControlProgressBar	UnoControlProgressBarModel	Progress bar
UnoControlRadioButton	UnoControlRadioButtonModel	Radio button
UnoControlRoadmap	UnoControlRoadmapModel	Roadmap control
UnoControlScrollBar	UnoControlScrollBarModel	Scroll bar
UnoControlTimeField	UnoControlTimeFieldModel	Time field

Note

To assist in the demonstration of the different dialog controls, create a dialog called `OOMESample`. As the control types are discussed, I will direct you to add controls to the `OOMESample` dialog. See Figure 138 for the final dialog. Start by creating the `OOMESample` dialog and making it large enough to fill most of the screen. ?? How BIG ??

18.3.1. Control button

A control button is a standard button that causes something to happen when it is “pressed.” A button’s type is controlled by the `PushButtonType` property in a button’s model. A button can be set to automatically close or cancel a dialog without writing an event handler; a default event handler is provided automatically for these predefined actions (see Table 269).

Table 269. Values defined by the `com.sun.star.awt.PushButtonType` enumeration.

Enumeration	Description
STANDARD	The default value. You must write your own event handler.
OK	Act like an OK button by closing the dialog and returning 1 from the <code>execute()</code> function.
CANCEL	Act like a Cancel button by closing the dialog and returning 0 from the <code>execute()</code> function.
HELP	Act like a Help button and open the help URL.

A control button can display a text label set by the `Label` property or a graphic image set by the `ImageURL` property. If the button displays an image rather than a label, the image is aligned based on a value defined by the `ImageAlign` constant group (see Table 270).

Table 270. Constants defined by the `com.sun.star.awt.ImageAlign` constant group.

Constant	Name	Description
0	LEFT	Align the image to the left.
1	TOP	Align the image to the top.
2	RIGHT	Align the image to the right.
3	BOTTOM	Align the image to the bottom.

TIP	Although a button can contain an image, it crops the image to fit. Use an <code>Image</code> control, rather than a button, if you need to scale the image.

The button model supports many of the same properties as the dialog model but it also supports some button-specific properties (see Table 271).

Table 271. Properties defined by the `com.sun.star.awt.UnoControlButtonModel` service.

Property	Description
Align	Optional, specifies the horizontal alignment of the text in the control.
BackgroundColor	Dialog’s background color as a Long Integer.
DefaultButton	If True, this button is the default button.
Enabled	If True, the dialog is enabled.

Property	Description
FontDescriptor	FontDescriptor structure for the text in the control.
FontEmphasisMark	FontEmphasis for the text in the dialog's caption bar.
FontRelief	FontRelief for the text in the dialog's caption bar.
HelpText	Dialog's help text as a String.
HelpURL	Dialog's help URL as a String.
ImageAlign	Specify how an image is aligned in the button (see Table 270).
ImageURL	URL of the image to display on the button. If the image is too large, it is cropped.
Label	String label displayed on the control.
MultiLine	Optional, specifies that the text may be displayed on more than one line.
Printable	If True, the button is printed when the document is printed.
PushButtonType	The default button action (see Table 269).
State	If 1, the button is activated; otherwise, the state is 0.
Tabstop	If True, the control can be reached by using the Tab key.
TextColor	Dialog's text color as a Long Integer.
TextLineColor	Dialog's text line color as a Long Integer.
Toggle	Optional, specifies if the button should toggle in a single operation.
VerticalAlign	Optional, specifies the vertical alignment of the text in the control. Supported values are from com.sun.star.style.VerticalAlignment with supporting values of TOP, MIDDLE, or BOTTOM.

Note Add an OK button to close the OOMESample dialog. Add a second button, named ClickButton, that calls OOMEDlgButton in Listing 529.

Add two buttons to the OOMESample dialog. Set one button to close the dialog when it is selected; label the button “Close” and set the button type to OK. The close button will automatically close the dialog when it is selected. Name the second button ClickButton and set the label to “Click Here”. Enter the code in Listing 529 and set the “Execute action” event of the ClickButton to call the OOMEDlgButton subroutine. When the ClickButton is selected, its label is changed to indicate the number of times that the button has been clicked.

Listing 529. Run the sample OOME Dialog.

Option Explicit

```

REM Global variables
Dim oOOMEDlg      'The run-time dialog that is created
Dim nClickCount% 'Number of times the button was clicked

Sub RunOOMEDlg
    REM Declare the primary variables
    Dim oLib        'Library that contains the dialog

```

```

Dim oLibDlg      'Dialog as stored in the library

REM Load the library and the dialog
DialogLibraries.loadLibrary("OOME_40")
oLib = DialogLibraries.getByName("OOME_40")
oLibDlg = oLib.getByName("OOMESample")
oOOMEDlg = CreateUnoDialog(oLibDlg)

REM Initial setup code here
nClickCount = 0
oOOMEDlg.getModel.Title = "OpenOffice.org Macros Explained Sample Dialog"

REM This can be used with the check box to be added.
REM oOOMEDlg.getModel().getByName("CheckBox").TriState = True

REM Run the dialog
oOOMEDlg.execute()
End Sub

Sub OOMEDlgButton
    Dim oButtonModel
    nClickCount = nClickCount + 1
    oButtonModel = oOOMEDlg.getModel().getByName("ClickButton")
    oButtonModel.Label = "Click " & nClickCount
End Sub

```

18.3.2. Check box

Although a check box is usually used to indicate a single state of yes or no, the check box in OOo can be used as a tristate check box. The default behavior is for a check box to support the states 0 (not checked) and 1 (checked); this is the State property in the model. If the TriState property is True, however, a third state of 2 is allowed. The third state is displayed as checked, but the check box is dimmed. The properties in Table 272 are supported by the check box button model.

Table 272. Properties defined by *com.sun.star.awt.UnoControlCheckBoxModel*.

Property	Description
Enabled	If True, the dialog is enabled.
FontDescriptor	FontDescriptor structure for the text in the dialog's caption bar.
FontEmphasisMark	FontEmphasis for the text in the dialog's caption bar.
FontRelief	FontRelief for the text in the dialog's caption bar.
HelpText	Dialog's help text as a String.
HelpURL	Dialog's help URL as a String.
Label	String label displayed on the control.
Printable	If True, the button is printed when the document is printed.
State	If 0, the button is not checked; if 1, the button is checked; 2 is the "don't know" state.
Tabstop	If True, the control can be reached by using the Tab key.
TextColor	Dialog's text color as a Long Integer.

Property	Description
TextLineColor	Dialog's text line color as a Long Integer.
TriState	If True, the button supports the "don't know" state of 2.

Note Add a check box to the OOMESample dialog and name it CheckBox. The "When initiating" event will call OOMEDlgCheckBox in Listing 530.

Add a check box to the sample dialog and set the button name to CheckBox. In the RunOOMEDlg routine, add code to the initialization section to set the check box to be a tristate check box. The TriState property can also be enabled in the Properties dialog for the control in the Basic IDE.

```
oOOMEDlg.getModel().getByName("CheckBox").TriState = True
```

Add the code in Listing 530 and then set the "When initiating" event to call the routine in Listing 530. When the check box is selected, its label is updated to show its state.

Listing 530. *Change the label on the checkbox to indicate state.*

```
Sub OOMEDlgCheckBox
    Dim oModel
    oModel = oOOMEDlg.getModel().getByName("CheckBox")
    oModel.Label = "State " & oModel.state

    ' If oModel.state = 0 Then
    '     oOOMEDlg.getModel().getByName("OBClear").state = 1
    ' ElseIf oModel.state = 1 Then
    '     oOOMEDlg.getModel().getByName("OBSet").state = 1
    ' ElseIf oModel.state = 2 Then
    '     oOOMEDlg.getModel().getByName("OBUnknown").state = 1
    ' End If

End Sub
```

18.3.3. Radio button

Radio buttons are usually used to select one item from a small list. (For a large list, it's common to use a combo box or a list box.) Radio buttons are always used in a group, and only one radio button in each group is active at a time. When a radio button is selected, OOo automatically marks all other radio buttons in the same group as not active.

Every control supports the UnoControlDialogElement service (see Table 267). The TabIndex property specifies the order in which controls are visited while repeatedly pressing the Tab key. The TabIndex property is usually set in the IDE at design time by setting the control's "order" property. Unfortunately, a different name is used in the IDE than in the defining service.

In OOo, a visual grouping that uses an UnoControlGroupBox has no effect other than to draw a frame around the controls. In other words, it provides a visual reminder that the radio buttons should be grouped, but it does not group them. Radio buttons are in the same group if (and only if) they have a sequential tab order; in Visual Basic, radio buttons are grouped based on the group

box. Create all of the radio buttons that will exist in a single group in sequence. By creating the buttons sequentially, they will automatically have sequential tab stops and will therefore all be in the same group. If you then create a group box to place around the radio buttons, the group box will break the tab-stop sequence so that you can safely create a new sequence of radio buttons. See Table 273.

Table 273. *Properties in the com.sun.star.awt.UnoControlRadioButtonModel.*

Property	Description
Enabled	If True, the dialog is enabled.
FontDescriptor	FontDescriptor structure for the text in the dialog's caption bar.
FontEmphasisMark	FontEmphasis for the text in the dialog's caption bar.
FontRelief	FontRelief for the text in the dialog's caption bar.
HelpText	Dialog's help text as a String.
HelpURL	Dialog's help URL as a String.
Label	String label displayed on the control.
Printable	If True, the button is printed when the document is printed.
State	If 0, the button is not checked; if 1, the button is checked.
Tabstop	If True, the control can be reached by using the Tab key.
TextColor	Dialog's text color as a Long Integer.
TextLineColor	Dialog's text line color as a Long Integer.

Note Add three radio buttons to the OOMESample dialog.

Add three radio buttons to the sample dialog and name them OBSet, OBClear, and OBUnknown. The three radio buttons will influence the state of the check box, so set an appropriate label as well (see Figure 138). In other words, the OBSet button will set the check box, the OBClear radio button will clear the check box, and the OBUnknown radio button will set the check box to the “don't know” state of 3. Enter the macro in Listing 531 and then set the “When initiating” event to call the OOMEDlgRadioButton routine for each of the three radio buttons. Each of the radio buttons is inspected to see which one is currently selected.

Listing 531. *Handle the sample radio buttons.*

```
REM When a radio button is selected, set the state of the check box
REM based on which radio button is selected. Set the check box
REM label to indicate why the check box was set to that state.
Sub OOMEDlgRadioButton
    Dim oModel
    Dim i As Integer
    Dim sBNames (0 To 2) As String
    REM These are ordered so that the button names are in the order
    REM that will set the check box state.
    sBNames(0) = "OBClear" : sBNames(1) = "OBSet" : sBNames(2) = "OBUnknown"
    For i = 0 To 2
```

```

oModel = oOOMEDlg.getModel().getByName(sbNames(i))
If oModel.State = 1 Then
    oOOMEDlg.getModel().getByName("CheckBox").State = i
    oOOMEDlg.getModel().getByName("CheckBox").Label = "Set by " & sbNames(i)
    REM oOOMEDlg.getModel().getByName("RBFrame").Label = sbNames(i)
Exit For
End If
Next
End Sub

```

Change the code for the checkbox as follows

18.3.4. Group box

The purpose of a group box is to draw a visual frame—a useful visual cue to the user that a group of controls are related in some way. The group box model supports the properties Enabled, FontDescriptor, FontEmphasisMark, FontRelief, HelpText, HelpURL, TextColor, TextLineColor, Label, and Printable.

Note Draw a group box around the three radio buttons in the OOMESample dialog.

Draw a group box around the three radio buttons and name the group box RBFrame. The macro in Listing 531 contains one line of code that will set the frame label to the name of the currently selected radio button. Remove the comment mark (REM) from the front of this line or the frame label will not update.

18.3.5. Fixed line control

The fixed line control, like the group box control, provides a nice visual separator for the user. Its only purpose is to draw a separating line in the dialog. The only interesting property in the fixed-line-control model is the Orientation property, which specifies if the line is drawn horizontally (0) or vertically (1).

18.3.6. Combo box

A combo box, sometimes called a “drop-down” control, is an input field with an attached list box. The combo box control is used to select a single value. The attached list box facilitates selecting a value from a predefined list. The selected text is available from the Text property in the control’s model. See Table 274.

Table 274. Properties defined by *com.sun.star.awt.UnoControlComboBoxModel*.

Property	Description
Autocomplete	If True, automatic completion of text is enabled.
Border	Specify no border (0), a 3-D border (1), or a simple border (2) as an Integer.
Dropdown	If True, the control contains a drop-down button.
LineCount	Maximum line count displayed in the drop-down box.
MaxTextLen	Maximum character count. If 0, the text length is unlimited.

Property	Description
Printable	If True, the button is printed when the document is printed.
ReadOnly	If True, the Text entry cannot be modified by the user.
StringItemList	Array of strings that identify the items in the list.
Tabstop	If True, the control can be reached by using the Tab key.
Text	Text that is displayed in the input field section of the box.
BackgroundColor	Dialog's background color as a Long Integer.
Enabled	If True, the dialog is enabled.
FontDescriptor	FontDescriptor structure for the text in the dialog's caption bar.
FontEmphasisMark	FontEmphasis for the text in the dialog's caption bar.
FontRelief	FontRelief for the text in the dialog's caption bar.
HelpText	Dialog's help text as a String.
HelpURL	Dialog's help URL as a String.
TextColor	Dialog's text color as a Long Integer.
TextLineColor	Dialog's text line color as a Long Integer.

Although it is considered good practice to interact with a control through the control's model, the combo box control contains some useful utility methods that are not directly available from the model. The methods listed in Table 275 essentially provide a shortcut to writing your own array utility functions to add and remove items from the StringItemList array in Table 274.

Table 275. *Extra utility methods supported by the combo box control.*

Method	Description
addItem(String, position)	Add an item at the specified position.
addItems(StringArray, position)	Add multiple items at the specified position.
removeItems(position, count)	Remove a number of items at the specified position.

Note Add a combo box to the OOMESample dialog and name it ColorBox.

Add a combo box to the dialog with a name of "ColorBox". Select the "List entries" property and enter the three values "Red", "Green", and "Blue". In earlier versions of OO, you entered multiple values by separating them with Ctrl+Enter. In AOO 3.4.1, use Shift+Enter. To place each entry on a separate line, type the word Red and press Shift+Enter, type the word Green and press Shift+Enter, and so on—it took me longer than it should have to figure out how to do this. It's easy, of course, to simply set these values directly into the model by setting the StringItemList property shown in Table 274.

Enter the code shown in Listing 532 and then set the "Text modified" event to call the OOMDdlgColorBox subroutine in Listing 532. The "Text modified" event causes the subroutine to be called every time that the text in the input line changes. In other words, while typing text, the

subroutine is called with every keystroke. If the “Item status changed” event is used, then the subroutine is called when a new selection is chosen from the drop-down list.

The macro in Listing 532 changes the background color of the drop-down list box based on the value of the entered text. The text comparison is case sensitive, so the text in the drop-down box must match exactly.

Listing 532. *Modify drop down list box background color.*

```
Sub OOMEDlgColorBox
    Dim oModel
    Dim s As String
    oModel = oOOMEDlg.getModel().getByName("ColorBox")
    s = oModel.Text
    If s = "Red" Then
        REM Set to Red
        oModel.BackgroundColor = RGB(255, 0, 0)
    ElseIf s = "Green" Then
        REM Set to Green
        oModel.BackgroundColor = RGB(0, 255, 0)
    ElseIf s = "Blue" Then
        REM Set to Blue
        oModel.BackgroundColor = RGB(0, 0, 255)
    Else
        Rem set back to white
        oModel.BackgroundColor = RGB(255, 255, 255)
    End If
End Sub
```

18.3.7. Text edit controls

The UnoControlEdit service is a basic edit field that acts as the base service for the other edit fields. An edit control is used to hold regular text. It is possible to limit the length of the text and to even add scroll bars. The edit control model supports the properties shown in Table 276.

Table 276. *Properties defined by the com.sun.star.awt.UnoControlEditModel service.*

Property	Description
Align	Specify that the text is aligned left (0), center (1), or right (2).
BackgroundColor	Dialog’s background color as a Long Integer.
Border	Specify no border (0), a 3-D border (1), or a simple border (2).
EchoChar	A password field displays the character specified by this integer rather than the entered text.
Enabled	If True, the dialog is enabled.
FontDescriptor	FontDescriptor structure for the text in the dialog’s caption bar.
FontEmphasisMark	FontEmphasis for the text in the dialog’s caption bar.
FontRelief	FontRelief for the text in the dialog’s caption bar.
HardLineBreaks	If True, hard line breaks are returned by the control’s getText() method.
HelpText	Dialog’s help text as a String.
HelpURL	Dialog’s help URL as a String.

Property	Description
HScroll	If True, the text in the control can be scrolled in the horizontal direction.
MaxTextLen	Maximum number of characters the control may contain. If zero, the text length is not purposely limited.
MultiLine	If True, the text may use more than one line.
Printable	If True, the control is printed with the document.
ReadOnly	If True, the control's content cannot be modified by the user.
Tabstop	If True, the control can be reached by using the Tab key.
Text	The text that is displayed in the control.
TextColor	Dialog's text color as a Long Integer.
TextLineColor	Dialog's text line color as a Long Integer.
VScroll	If True, the text in the control can be scrolled in the vertical direction.

Although I encourage you to use the model for most things, some features are not available without going directly to the control. For example, the control supplies the capability of selecting some or all of the text and determining which portion of the text is currently selected. In a regular document, the current controller provides this capability, but controls in dialogs combine the controller's capabilities with the control itself. To obtain or set the selected range, use the Selection structure (see Table 277).

Table 277. *Properties in the com.sun.star.awt.Selection structure.*

Property	Description
Min	Lower limit of the range as a Long Integer.
Max	Upper limit of the range as a Long Integer.

The control uses the Selection structure to get and set the current text selection range. The control is also able to directly return the text that is selected in a text control. The purpose is to allow a user to select only a portion of the text and to determine which portion is selected. Table 278 contains the standard text methods supported by text type controls.

Table 278. *Methods defined by the com.sun.star.awt.XTextComponent interface.*

Method	Description
addTextListener(XTextListener)	Add a text listener.
getMaxTextLen()	Get the model's MaxTextLen property (see Table 276).
getSelectedText()	Return the currently selected text string.
getSelection()	Return a Selection object that identifies the selected text (see Table 275).
getText()	Get the model's Text property (see Table 276).
insertText(Selection, string)	Insert the string at the specified Selection location.
isEditable()	Return the model's ReadOnly flag (see Table 276).
removeTextListener(XTextListener)	Remove a text listener.

Method	Description
setEditable(boolean)	Set the model's ReadOnly flag (see Table 276).
setMaxTextLen(len)	Set the model's MaxTextLen property (see Table 276).
setSelection(Selection)	Set the text that is selected (see Table 275).
setText(string)	Set the model's Text property (see Table 276).

Currency control

The currency control is used to input and edit monetary values. The currency control is defined by the `UnoControlCurrencyField` service, which also implements the standard edit control service. Table 279 lists some properties that are unique to the currency model, all of which may also be set from the control's property dialog in the IDE. Pay special attention to the default values, and notice that the property dialog uses Yes and No rather than True and False. Besides the specific properties in Table 279, the currency control model also supports these properties: `BackgroundColor`, `Border`, `Enabled`, `FontDescriptor`, `FontEmphasisMark`, `FontRelief`, `HelpText`, `HelpURL`, `Printable`, `ReadOnly`, `Tabstop`, `TextColor`, and `TextLineColor`.

Table 279. Properties in the `com.sun.star.awt.UnoControlCurrencyFieldModel` service.

Property	Description	Default
CurrencySymbol	Currency symbol as a String.	\$
DecimalAccuracy	Decimal accuracy as an Integer.	2
PrependCurrencySymbol	If True, the currency symbol is prepended to the currency amount.	No
ShowThousandsSeparator	If True, the thousands separator is displayed.	No
Spin	If True, the control has a spin button.	No
StrictFormat	If True, the entered text is checked during the user input.	No
Value	The value displayed in the control as a floating-point Double.	0
ValueMax	The maximum value as a floating-point Double.	1000000
ValueMin	The minimum value as a floating-point Double.	-1000000.00
ValueStep	The amount that the spin button changes the entered values as a Double.	1

TIP The default values shown in Table 279 are for the United States. Local specific default values are used.

The `XCurrencyField` interface implemented by the currency control defines methods to get and set the properties defined in Table 279. The `XSpinField` interface, which is also supported by the currency control, defines methods to interact with the spin control (see Table 280).

Table 280. Methods defined by the `com.sun.star.awt.XSpinField` interface.

Method	Description
addSpinListener(XSpinListener)	Add a spin listener.

Method	Description
down()	Decreases the value by ValueStep (see Table 279).
enableRepeat(boolean)	Enable/disable automatic repeat mode.
first()	Set the value to ValueMin (see Table 279).
last()	Set the value to ValueMax (see Table 279).
removeSpinListener(XSpinListener)	Remove a spin listener.
up()	Increases the value by ValueStep (see Table 279).

TIP All of the edit controls that support spin buttons also support the methods in Table 280.

Numeric control

The numeric control is used to accept numeric input. The numeric control is almost identical to the currency control except that the numeric control does not support the properties `CurrencySymbol` or `PrependCurrencySymbol`—even the default values are the same. If you know how to use a currency control, you know how to use a numeric control.

Date control

The date control is a text-edit control that supports date input. The input formats supported by the date control are predefined and set based on a numeric constant (see Table 281). The descriptions show formats such as (DD/MM/YY), but in reality, the separator is locale specific. In other words, dates are shown and edited in a way that is appropriate for where you live. The last two formats are not locale specific because they are described by the ISO 8601 standard.

The date control (see Table 282 for its model properties) is similar to the currency control in that it supports a minimum value, a maximum value, and a spin control. The spin control works by incrementing or decrementing the portion of the date in which it resides. For example, if the cursor is in the month field, the spin control affects the month rather than the year or the day.

TIP The `DropDown` property allows for a drop-down calendar that greatly facilitates date entry.

Table 281. *Date input formats supported by a Date control.*

Value	Description
0	Default system short format.
1	Default system short format with a two-digit year (YY).
2	Default system short format with a four-digit year (YYYY).
3	Default system long format.
4	Short format (DD/MM/YY).
5	Short format (MM/DD/YY).
6	Short format (YY/MM/DD).
7	Short format (DD/MM/YYYY).
8	Short format (MM/DD/YYYY).
9	Short format (YYYY/MM/DD).
10	Short format for ISO 8601 (YY-MM-DD).
11	Short format for ISO 8601 (YYYY-MM-DD).

Table 282. *Properties in the com.sun.star.awt.UnoControlDateFieldModel service.*

Property	Definition
BackgroundColor	Dialog's background color as a Long Integer.
Border	Specify no border (0), a 3-D border (1), or a simple border (2).
Date	The date: – as a Long Integer (AOO and old versions of LO) – as a structure com.sun.star.util.Date (LO new, see Table 282).
DateFormat	Date format as an Integer (see Table 281).
DateMax	Maximum allowed date as a Long Integer.
DateMin	Minimum allowed date as a Long Integer.
DropDown	If True, a drop-down exists that shows a calendar for choosing a date.
Enabled	If True, the dialog is enabled.
FontDescriptor	FontDescriptor structure for the text in the dialog's caption bar.
FontEmphasisMark	FontEmphasis for the text in the dialog's caption bar.
FontRelief	FontRelief for the text in the dialog's caption bar.
HelpText	Dialog's help text as a String.
HelpURL	Dialog's help URL as a String.
Spin	If True, spin buttons are drawn.
StrictFormat	If True, the date is checked during the user input.
TextColor	Dialog's text color as a Long Integer.
TextLineColor	Dialog's text line color as a Long Integer.

Note Add a date input control to the OOMESample dialog and name it MyDateField.
Use the code in Listing 535 to initialize the date control.

The Date data type that is used by Basic is not compatible with the Date property used in a date control. It depends from the OpenOffice application used which data format you have to choose for a date control.

In older versions of LO and in AOO (still at least in the latest version 4.1.1) the date control uses a Long that represents the date in the form YYYYMMDD. For example, “February 3, 2004” is represented as the long integer 20040203. The Basic function CDateToIso() converts a Basic Date variable to the required format. Add a date input control and name it MyDateField. The code in Listing 533 sets the date control to the current day using the old method.

Listing 533. *Initialize a date control to today’s date (AOO and old LO).*

```
REM Set the initial date to TODAY
oOOMEDlg.getModel().getByName("MyDateField").Date = CdateToIso(Date())
```

LibreOffice made a change to use a structure, so the above code will fail on LO. You must instead create a com.sun.star.util.Date object (see Table 283) and then set the date with the new object.

Table 283. *Properties in the com.sun.star.util.Date structure*

Property	Description
Day	Day of month. Integer (1-31 or 0 for a void date).
Month	Month of year. Integer (1-12 or 0 for a void date).
Year	Year. Integer.

Listing 534. *Initialize a date control to today’s date. (new/current versions of LO)*

```
Dim x As Date
Dim x2 as New com.sun.star.util.Date
....
x = Date()          'Start with a date, then populate the struct:
x2.Year = Year(x)    ' Year
x2.Month = Month(x)  ' Month
x2.Day = Day(x)      ' Day
oOOMEDlg.getModel().getByName("MyDateField").Date = x2      'Set the model Date
```

Sadly there seems to be no workaround to work for both variations. If you need a date control in a macro that runs on AOO as well as on all LO versions you can control it by an error handler (see Listing 534). If you forgot how this works, recall section 3.10 Error handling using On Error. You can also call the GetProductName() macro in the Misc module in the Tools library, which is included with both AOO and LO.

Listing 535. *Initialize a date control to today’s date (works with all versions of AOO and LO).*

```
REM Initialize the date control.
REM Set the initial date to TODAY.
```

```

Dim tToday As Date
tToday = Date()

On Error Goto LO
REM Try to initialize the control with a Long Integer.
REM This is the correct format for AOO and for older versions of LO.
oOOMEDlg.getModel().getByName("MyDateField").Date = CdateToIso(tToday)
On Error Goto 0 'Clear the error handler.
Goto GoOn

LO:
REM On Error the control is initialized using a structure.
REM This is the correct format for newer versions of LO.
On Error Goto 0 'Clear the error handler to prevent an infinite loop,
                'in case the macro runs into some other runtime error.
Dim oToday As New com.sun.star.util.Date
oToday.Year = Year(tToday)
oToday.Month = Month(tToday)
oToday.Day = Day(tToday)
oOOMEDlg.getModel().getByName("MyDateField").Date = oToday

GoOn:

```

Although the control implements methods that get and set properties in the model, the more interesting methods are related to the user interface. While using a date field, press the Page Up key to cause the control to jump to the “last” date. By default, the last date is the same as the DateMax property. The last date is set by the control and not by the model, and it may differ from the DateMax. Analogously, the pressing the Page Down key causes the control to jump to the “first” date. The first and last dates are set using the setFirst() and setLast() methods shown in Table 284. Use the method isEmpty() to determine if the control contains no value, and use the method setEmpty() to cause the control to contain no value.

Table 284. *Methods in the com.sun.star.awt.UnoControlDateField interface.*

Method	Description
setDate(long) [LO old and AOO] setDate(com.sun.star.util.Date) [LO new]	Set the model’s Date property.
getDate()	Get the model’s Date property.
setMin(long)	Set the model’s DateMin property.
getMin(long)	Get the model’s DateMin property.
setMax(long)	Set the model’s DateMax property.
getMax(long)	Get the model’s DateMax property.
setFirst(long)	Set the date used for the Page Down key.
getFirst()	Get the date used for the Page Down key.
setLast(long)	Set the date used for the Page Up key.
getLast()	Get the date used for the Page Up key.
setLongFormat(boolean)	Set to use the long date format.

Method	Description
isLongFormat()	Return True if the long date format is used.
setEmpty()	Set the empty value.
isEmpty()	Return True if the current value is empty.
setStrictFormat(boolean)	Set the model's StrictFormat property.
isStrictFormat()	Get the model's StrictFormat property.

Time control

The time control is very similar to the date control. Time values may indicate either a time on a clock or a length of time (duration). The supported formats are shown in Table 285.

Table 285. Time input formats supported by a time control.

Value	Description
0	Short 24-hour format.
1	Long 24-hour format.
2	Short 12-hour format.
3	Long 12-hour format.
4	Short time duration.
5	Long time duration.

The properties supported by the time-control model are shown in Table 286. Modify Table 282 by changing the word Date to Time, and by removing the DropDown property, and Table 282 becomes Table 286. In other words, the date and time models are almost identical.

Table 286. Properties in the *com.sun.star.awt.UnoControlTimeFieldModel* service.

Property	Definition
Time	The time displayed in the control: – as a Long Integer (LO old and AOO) – as a structure <i>com.sun.star.util.Time</i> (LO new, see Table 286).
TimeFormat	Time format as an Integer (see Table 285).
TimeMax	Maximum allowed date as a Long Integer.
TimeMin	Minimum allowed date as a Long Integer.
Spin	If True, spin buttons are drawn.
StrictFormat	If True, the date is checked during the user input.

As expected, the methods supported by the date control are similar to the methods supported by the time control (see Table 284). There are two differences: The time control has *getTime()* and *setTime()* rather than *getDate()* and *setDate()*, and the time control does not support the methods *setLongFormat()* and *isLongFormat()*.

In older versions of LO and in AOO (still at least in the latest version 4.1.1) the time control uses a Long Integer to store and return the entered value. The rightmost two digits represent fractions of a second, the next two digits represent seconds, the next two digits represent minutes, and the leftmost two digits represent hours. Leading zeros are, of course, not returned because the value is really a Long Integer. The good news, however, is that the values are easy to extract. Assuming that the dialog contains a time field named MyTimeField, the code in Listing 536 obtains the time value and extracts the hours, minutes, seconds, and portions of a second.

Listing 536. *Extracting values from a time control (AOO and old versions of LO).*

```
n = oOOMEDlg.getModel().getByName("MyTimeField").Time
h = n / 1000000 MOD 100 REM Get the hours
m = n / 10000 MOD 100 REM Get the minutes
s = n / 100 MOD 100 REM Get the seconds
ss = n MOD 100 REM portions of a second
```

The new versions of LO use the com.sun.star.util.Time structure for the time control, see Table 287.

Table 287. *Properties in the com.sun.star.util.Time structure..*

Property	Description
HundredthSeconds [LO old and AOO]	Portions of a second: Integer (0 – 99)
NanoSeconds [LO new]	Long Integer (0 – 999 999 999)
Seconds	Integer (0 – 59).
Minutes	Integer (0 – 59).
Hours	Integer (0 – 23).
IsUTC [only LO new]	True: time zone is UTC, False: unknown time zone.

Reading a time value from a time control in newer versions of LO needs a variable of the com.sun.star.util.Time structure type, see Listing 537.

Listing 537. *Extracting values from a time control (LO new).*

```
Dim oTime As New com.sun.star.util.Time
oTime = oOOMEDlg.getModel().getByName("MyTimeField").Time
h = oTime.Hours REM Get the hours.
m = oTime.Minutes REM Get the minutes.
s = oTime.Seconds REM Get the seconds.
ss = oTime.NanoSeconds REM Get the portions of a second.
```

If you need a time control in a macro that runs with AOO as well as with all LO versions you can control it with an error handler like the one for the date control in Listing 535.

Formatted control

The formatted control is a generic number input field. Numbers can be entered as a formatted number, percent, currency, date, time, scientific, fraction, or Boolean number. Unfortunately, this level of control comes with a price. The formatted control requires a numeric formatter provided

by OpenOffice.org to function. The easiest method to associate a formatted control to a numeric format object is to use the Properties dialog in the IDE. To use a number formatter, the following sequence of events is likely to occur:

- Choose a numeric format.
- Use the methods in Table 290 to search the FormatsSupplier (see Table 29) using the constants in Table 289 to find or create a suitable number formatter.
- Set the FormatKey (see Table 288) to reference the desired format contained in the FormatsSupplier.

Table 288. *Properties in the com.sun.star.awt.UnoControlFormattedFieldModel service.*

Property	Description
Align	Specify that the text is aligned left (0), center (1), or right (2).
BackgroundColor	Dialog's background color as a Long Integer.
Border	Specify no border (0), a 3-D border (1), or a simple border (2).
EffectiveDefault	The default value of the formatted field. This is a number or a string based on TreatAsNumber.
EffectiveMax	The maximum value as a Double if TreatAsNumber is True.
EffectiveMin	The minimum value as a Double if TreatAsNumber is True.
EffectiveValue	The current value as a Double or a String based on the value of TreatAsNumber.
Enabled	If True, the dialog is enabled.
FontDescriptor	FontDescriptor structure for the text in the dialog's caption bar.
FontEmphasisMark	FontEmphasis for the text in the dialog's caption bar.
FontRelief	FontRelief for the text in the dialog's caption bar.
FormatKey	Long Integer that specifies the format for formatting.
FormatsSupplier	XNumberFormatsSupplier that supplies the formats used by this control.
HelpText	Dialog's help text as a String.
HelpURL	Dialog's help URL as a String.
MaxTextLen	Maximum number of characters the control may contain. If zero, the text length is not purposely limited.
Printable	If True, the control is printed with the document.
ReadOnly	If True, the control's content cannot be modified by the user.
Spin	If True, a spin control is used.
StrictFormat	If True, the text is checked during the user input.
Tabstop	If True, the control can be reached by using the Tab key.
Text	The text that is displayed in the control.
TextColor	Dialog's text color as a Long Integer.
TextLineColor	Dialog's text line color as a Long Integer.
TreatAsNumber	If True, the text is treated as a number. This property controls how the other

Property	Description
	values are treated.

The number formatter that controls the formatting in the input box must be created by the number format supplier in the `FormatsSupplier` property. In other words, you can't simply take a number formatter created by the document or some other number formatter. When the formats supplier is searched to determine what numeric formats it contains, the constants in the `NumberFormat` constant group are used (see Table 289).

Table 289. Constants in the `com.sun.star.util.NumberFormat` constant group.

Constant	Value	Description
0	ALL	All number formats.
1	DEFINED	User-defined number formats.
2	DATE	Date formats.
4	TIME	Time formats.
8	CURRENCY	Currency formats.
16	NUMBER	Decimal number formats.
32	SCIENTIFIC	Scientific number formats.
64	FRACTION	Number formats for fractions.
128	PERCENT	Percentage number formats.
256	TEXT	Text number formats.
6	DATETIME	Number formats that contain date and time.
1024	LOGICAL	Boolean number formats.
2048	UNDEFINED	Used if no number format exists.

The formatted control model contains a number format supplier in the `FormatsSupplier` property. The supplier identifies its contained numeric formats based on a numeric key. The `FormatKey` property in Table 288 must be set with the key obtained from the `FormatsSupplier`. Use the methods in Table 290 to find or create the numeric format for the formatted control.

Table 290. Properties in the `com.sun.star.awt.UnoControlFormattedFieldModel` service.

Method	Description
<code>getByKey(key)</code>	Return the <code>com.sun.star.util.NumberFormatProperties</code> based on the numeric key.
<code>queryKeys(NumberFormat, Locale, boolean)</code>	Return an array of Long Integer keys that identify numeric formats of the specified type (see Table 289) for the specified locale. If the final Boolean parameter is <code>True</code> and if no formats exist, one will be created.
<code>queryKey(format, Locale, boolean)</code>	Return the Long Integer key for the specified format string.
<code>addNew(format, Locale)</code>	Add a number format and return the Long Integer key.
<code>addNewConverted(format, Locale,</code>	Add a number format, using a format string in a locale that is different than the

Method	Description
Locale)	desired locale.
removeByKey(key)	Remove a number format based on the Long Integer key.
generateFormat(key, Locale, bThousands, bRed, nDecimals, nLeading)	Return a format string based on the input values, but do not actually create the number format.

Note Add a formatted control to the OOMESample dialog.

Although it is not overly difficult to use the methods in Table 290 to find and create your own numeric formats, the easiest method to associate a formatted control to a numeric format object is to use the Properties dialog in the IDE. Add a formatted control to the dialog and name it MyFormattedField. The code in Listing 538 initializes the control to use the current date as its initial value and sets the desired date format. The date is formatted using the format “DD. MMM YYYY”.

Listing 538. *Set a formatted control to use a date as “DD. MMM YYYY”.*

```

REM Set the formatted field to a specific format.
Dim oFormats      'All of the format objects
Dim oSupplier      'All of the format objects
Dim nKey As Long  'Index of number format in the number formats
Dim oLocale as new com.sun.star.lang.Locale
Dim sFormat As String

sFormat = "DD. MMM YYYY"
oSupplier =oOOMEDlg.getModel().getByName("MyFormattedField").FormatsSupplier
oFormats  =oSupplier.getNumberFormats()

REM See if the number format exists by obtaining the key.
nKey = oFormats.queryKey(sFormat, oLocale, True)

REM If the number format does not exist, add it.
If (nKey = -1) Then
    nKey = oFormats.addNew(sFormat, oLocale)
    REM If it failed to add, and it should not fail to add, then use zero.
    If (nKey = -1) Then nKey = 0
End If

REM I can set the date using a string or the date.
REM ....EffectiveValue = "12. Apr 2004"
REM ...EffectiveValue = Date() fails but this works.
oOOMEDlg.getModel().getByName("MyFormattedField").EffectiveValue = CStr(Date())

REM Now, set the key for the desired date formatting.
REM I set this to treat the value as a number.
oOOMEDlg.getModel().getByName("MyFormattedField").FormatKey = nKey
oOOMEDlg.getModel().getByName("MyFormattedField").TreatAsNumber = True

```

There are a couple of interesting things to notice in Listing 538. First, the EffectiveValue property is used rather than the Text property (which I can use). The TreatAsNumber property is set to True because the spin control does not properly increment an initial date if the TreatAsNumber property is set to False. Although it is not shown, the formatted control uses the same numeric values for dates as the Basic Date data type—unlike the Date control. Finally, the spin button in a formatted control always increments and decrements the value by 1—unlike the Date control, which can independently increment and decrement each portion of the date.

Pattern control

Use a pattern control to enter formatted strings based on an edit mask and a literal mask. The literal mask is the string value that is initially displayed when the control starts. The user then edits the value in the control but is limited by the edit mask. The edit mask contains special characters that define what value can be entered at which location. For example, the character L (literal) indicates that the user cannot enter a value at the specified location, and the value N (number) indicates that the user can only enter the characters 0 through 9.

In the United States, each person is issued an identification number for tax purposes called a social security number. The social security number is composed of three numbers, a dash, two numbers, a dash, and four numbers. The edit mask is given as “NNNLNNLNNN”. This allows the user to enter numeric values at the appropriate locations, but prevents any modifications where the dashes are located. A literal mask of the form “__-__-__” provides the initial template for the user to see, as illustrated in Listing 539. The use of the character “_” is an arbitrary choice; you may use any value that you desire, such as a space character.

***Listing 539.** Edit mask and literal mask for a social security number.*

```
EditMask      = "NNNLNNLNNN"
LiteralMask = "__-__-__"
```

The number of characters that can be entered is determined by the length of the edit mask. It is considered an error if the edit mask and the literal mask are not the same length. Table 291 shows the edit mask characters supported by the pattern control. The pattern field supports the properties in Table 292.

***Table 291.** Edit mask characters supported by the pattern control.*

Character	Description
L	A text constant that is displayed but cannot be edited.
a	The characters a–z and A–Z can be entered.
A	The characters a–z and A–Z are valid, but the letters a–z are converted to uppercase.
c	The characters a–z, A–Z, and 0–9 can be entered.
C	The characters a–z, A–Z, and 0–9 are valid, but the letters a–z are converted to uppercase.
N	The characters 0–9 can be entered.
x	Any printable character can be entered.

Character	Description
X	Any printable character can be entered, but the letters a–z are converted to uppercase.

Table 292. *Properties in the com.sun.star.awt.UnoControlPatternFieldModel service.*

Property	Description
BackgroundColor	Dialog's background color as a Long Integer.
Border	Specify no border (0), a 3-D border (1), or a simple border (2).
EditMask	The edit mask.
Enabled	If True, the dialog is enabled.
FontDescriptor	FontDescriptor structure for the text in the dialog's caption bar.
FontEmphasisMark	FontEmphasis for the text in the dialog's caption bar.
FontRelief	FontRelief for the text in the dialog's caption bar.
HelpText	Dialog's help text as a String.
HelpURL	Dialog's help URL as a String.
LiteralMask	The literal mask.
MaxTextLen	Maximum number of characters the control may contain. If zero, the text length is not purposely limited.
Printable	If True, the control is printed with the document.
ReadOnly	If True, the control's content cannot be modified by the user.
StrictFormat	If True, the text is checked during the user input.
Tabstop	If True, the control can be reached by using the Tab key.
Text	The text that is displayed in the control.
TextColor	Dialog's text color as a Long Integer.
TextLineColor	Dialog's text line color as a Long Integer.

TIP If the StrictFormat property is False, the pattern control ignores the edit mask and the literal mask.

Fixed text control

Use the fixed text control to label other controls and areas in a dialog. In the IDE, the fixed text control is called a “label field.” The fixed text control is a very simple text edit field. The model supports the properties in Table 293.

Table 293. *Properties in the com.sun.star.awt.UnoControlFixedTextFieldModel service.*

Property	Description
Align	Specify that the text is aligned left (0), center (1), or right (2).
BackgroundColor	Control's background color as a Long Integer.
Border	Specify no border (0), a 3-D border (1), or a simple border (2).
Enabled	If True, the control is enabled.

Property	Description
FontDescriptor	FontDescriptor structure for the text in the control.
FontEmphasisMark	FontEmphasis for the text in the control.
FontRelief	FontRelief for the text in the control.
HelpText	Control's help text as a String.
HelpURL	Control's help URL as a String.
Label	Text label displayed in the control.
MultiLine	If True, the text may use more than one line.
Printable	If True, the control is printed with the document.
TextColor	Control's text color as a Long Integer.
TextLineColor	Control's text line color as a Long Integer.

File control

The file control is a text edit control that contains a command button, which opens the File dialog. The initial directory that is used is set by the Text property. Table 294 contains the properties defined by the file control model.

Table 294. *Properties in the com.sun.star.awt.UnoControlFileControlModel service.*

Property	Description
BackgroundColor	Control's background color as a Long Integer.
Border	Specify no border (0), a 3-D border (1), or a simple border (2).
Enabled	If True, the control is enabled.
FontDescriptor	FontDescriptor structure for the text in the control.
FontEmphasisMark	FontEmphasis for the text in the control.
FontRelief	FontRelief for the text in the control.
HelpText	Control's help text as a String.
HelpURL	Control's help URL as a String.
Printable	If True, the control is printed with the document.
Text	The text displayed in the control.
TextColor	Control's text color as a Long Integer.
TextLineColor	Control's text line color as a Long Integer.

The file control is easy to use, but unfortunately it doesn't support using filters. A frequent solution for this is to add a text input box with a button next to it. When the button is selected, it opens a file selection dialog with the desired filters. When the dialog is closed, it adds the selected file to the text control.

Note Add a text field named FileTextField to the OOMESample dialog. Add a button next to the text field that calls the macro in Listing 540.

Create a text field and name it FileTextField. Add a button next to the text field and set the “When initiating” event to call the macro in Listing 540. This macro opens a file selection dialog based on the directory or file stored in the FileTextField. If the text field is empty, the macro checks the configuration information to determine the starting directory. The interesting thing about this example is that it uses multiple file filters—something that isn’t possible if a file control is used.

Listing 540. Select a file.

```
Sub FileButtonSelected
    Dim oFileDialog          'File selection dialog
    Dim oSettings            'Settings object to get the path settings
    Dim sFile As String      'File URL as a string
    Dim oFileAccess          'Simple file access object
    Dim oFiles               'The File dialog returns an array of selected files

    REM Start with the value in the text field
    sFile = oOOMEDlg.getModel().getByName("FileTextField").Text
    If sFile <> "" Then
        sFile = ConvertToURL(sFile)
    Else
        REM The text field is blank, so obtain the path settings
        oSettings = CreateUnoService("com.sun.star.util.PathSettings")
        sFile = oSettings.Work
    End If

    REM Dialog to select a file
    oFileDialog = CreateUnoService("com.sun.star.ui.dialogs.FilePicker")

    REM Set the supported filters
    oFileDialog.AppendFilter( "All files (*.*)", "*.*)" )
    oFileDialog.AppendFilter( "JPG files (*.jpg)", "*.jpg" )
    oFileDialog.AppendFilter( "OOo Writer (*.sxw)", "*.sxw" )
    oFileDialog.AppendFilter( "OOo Calc (*.sxc)", "*.sxc" )
    oFileDialog.AppendFilter( "OOo Draw (*.sxd)", "*.sxd" )
    oFileDialog.AppendFilter( "OOo Impress (*.sxi)", "*.sxi" )
    oFileDialog.SetCurrentFilter( "All files (*.*)" )

    REM Determine if the "file" is a directory or a file!
    oFileAccess = CreateUnoService("com.sun.star.ucb.SimpleFileAccess")
    If oFileAccess.exists(sFile) Then
        REM I do not force the "display directory" to be a folder.
        REM I could do something fancy with this such as
        REM If NOT oFileAccess.isFolder(sFile) Then extract the folder...
        REM but I won't!
        oFileDialog.setDisplayDirectory(sFile)
    End If

    REM Execute the File dialog.
    If oFileDialog.execute() Then
        oFiles = oFileDialog.GetFiles()
        If UBound(oFiles) >= 0 Then
            sFile = ConvertFromURL(oFiles(0))
            oOOMEDlg.getModel().getByName("FileTextField").Text = sFile
        End If
    End If
End Sub
```

```

End If
End If
End Sub

```

18.3.8. Image control

An image control is used to display a graphic image. The image control is able to automatically scale an image to fit inside the image control. Use the ImageURL property to specify the image to load and the ScaleImage property to tell the control to scale the image automatically (see Table 295).

Table 295. *Properties in the com.sun.star.awt.UnoControlImageControlModel service.*

Property	Description
BackgroundColor	Control's background color as a Long Integer.
Border	Specify no border (0), a 3-D border (1), or a simple border (2).
Enabled	If True, the dialog is enabled.
HelpText	Control's help text as a String.
HelpURL	Control's help URL as a String.
ImageURL	URL of the image to load.
Printable	If True, the control is printed with the document.
ScaleImage	If True, the image is automatically scaled to the size of the control.
TabStop	If True, the Tab key can reach this control.

18.3.9. Progress control

A progress bar is a straight bar that fills in a solid color starting at the left and moving to the right. The idea is that at the start the progress bar is empty, and it fills as work is accomplished. A typical example is the "Saving document" progress bar at the bottom of OpenOffice when you click the Save icon to save an open document.

Use the ProgressValueMin and ProgressValueMax properties to tell the control the range of values that it will contain. When the ProgressValue property is at the minimum value, the control is empty—nothing is filled. When the ProgressValue is halfway between the minimum and the maximum values, the control is half filled. Finally, when the ProgressValue is at the maximum value, the control is completely filled. The progress bar is not limited to the values of empty, full, and half; these values are only examples. Table 296 contains the properties supported by the progress control model.

Table 296. *Properties in the com.sun.star.awt.UnoControlProgressBarModel service.*

Property	Description
BackgroundColor	Control's background color as a Long Integer.
Border	Specify no border (0), a 3-D border (1), or a simple border (2).
Enabled	If True, the dialog is enabled.

Property	Description
FillColor	Color used to fill the progress bar.
HelpText	Control's help text as a String.
HelpURL	Control's help URL as a String.
Printable	If True, the control is printed with the document.
ProgressValue	Current progress value as a Long Integer.
ProgressValueMax	Maximum progress value as a Long Integer.
ProgressValueMin	Minimum progress value as a Long Integer.

TIP You can set both BackgroundColor and FillColor.

18.3.10. List box control

A list box contains a list of items—one per line—from which the user may select zero or more items. In other words, a list box provides a mechanism to select multiple items from a list. The list box control automatically adds scroll bars if they are required. Table 297 contains a list of properties supported by the list box model.

Table 297. *Properties in the com.sun.star.awt.UnoControlListBoxModel service.*

Property	Description
BackgroundColor	Control's background color as a Long Integer.
Border	Specify no border (0), a 3-D border (1), or a simple border (2).
Dropdown	If True, the control has a drop-down button.
Enabled	If True, the control is enabled.
FontDescriptor	FontDescriptor structure for the text in the control.
FontEmphasisMark	FontEmphasis for the text in the control.
FontRelief	FontRelief for the text in the control.
HelpText	Control's help text as a String.
HelpURL	Control's help URL as a String.
LineCount	The maximum line count displayed in the drop-down box.
MultiSelection	If True, more than one entry can be selected.
Printable	If True, the control is printed with the document.
ReadOnly	If True, the user cannot modify the data in the control.
SelectedItems	Array of Short Integers that represent the index of the selected items.
StringItemList	Array of strings that represent the items in the list box.
Tabstop	If True, the control can be reached by using the Tab key.
TextColor	Control's text color as a Long Integer.
TextLineColor	Control's text line color as a Long Integer.

Although I recommend doing most manipulations through the model, many things are frequently done using methods defined for the control. The list box control supports the same special functions supported by the combo box control (see Table 275) to add and remove items at specific locations. The list box control supports other useful methods that are primarily related to the user interaction (see Table 298).

Table 298. *Methods defined by the `com.sun.star.awt.XListBox` interface.*

Method	Description
<code>addItemListener(XItemListener)</code>	Add a listener for item events.
<code>removeItemListener(XItemListener)</code>	Remove a listener for item events.
<code>addActionListener(XActionListener)</code>	Add a listener for action events.
<code>removeActionListener(XActionListener)</code>	Remove a listener for action events.
<code>addItem(String, position)</code>	Add an item at the specified position.
<code>addItems(StringArray, position)</code>	Add multiple items at the specified position.
<code>removeItems(position, count)</code>	Remove a number of items at the specified position.
<code>getItemCount()</code>	Get the number of items in the list box.
<code>getItem(position)</code>	Get the specified item.
<code>getItems()</code>	Return the <code>StringItemList</code> property (see Table 297).
<code>getSelectedItemPos()</code>	Return the position of one of the selected items. This is useful if only one item can be selected.
<code>getSelectedItemPos()</code>	Return the <code>SelectedItems</code> property (see Table 297).
<code>getSelectedItem()</code>	Return one of the selected items. This is useful if only one item can be selected.
<code>getSelectedItems()</code>	Return all selected items as an array of strings.
<code>selectItemPos(position, boolean)</code>	Select or deselect the item at the specified position.
<code>selectItemsPos(positions, boolean)</code>	Select or deselect multiple items at the positions specified by the array of positions.
<code>selectItem(item, boolean)</code>	Select or deselect the specified item string.
<code>isMultipleMode()</code>	Get the <code>MultiSelection</code> property (see Table 297).
<code>setMultipleMode(boolean)</code>	Set the <code>MultiSelection</code> property (see Table 297).
<code>getDropDownLineCount()</code>	Get the <code>LineCount</code> property (see Table 297).
<code>setDropDownLineCount(integer)</code>	Set the <code>LineCount</code> property (see Table 297).
<code>makeVisible(position)</code>	Scroll the items in the list box so that the specified item is visible.

18.3.11. Scroll bar control

The scroll bar control can be used to scroll a control or dialog that does not already contain or support a scroll bar. All of the work to control another object and synchronize the external object

with the scroll bar must be done using a macro. In other words, the scroll bar control does not automatically connect to or control another object, but requires a macro to implement the connection. To use the scroll bar control, write a macro that responds to the control's events and then manipulate another object based on what happens in the control. If you want the scroll bar to track changes in the controlled object, you must write handlers for the other object to update the scroll bar. As an input or an output, the scroll bar serves to indicate a relationship in another object; the relationship displayed and the corresponding action depends on the programming of a macro. See Table 142.

Table 299. *Properties in the com.sun.star.awt.UnoControlScrollBarModel service.*

Property	Description
BlockIncrement	Increment for a block move as a Long Integer.
Border	Specify no border (0), a 3-D border (1), or a simple border (2).
Enabled	If True, the control is enabled.
HelpText	Control's help text as a String.
HelpURL	Control's help URL as a String.
LineIncrement	Increment for a single-line move as a Long Integer.
Orientation	Scroll bar orientation with values com.sun.star.awt.ScrollBarOrientation.HORIZONTAL (0) and com.sun.star.awt.ScrollBarOrientation.VERTICAL (1).
Printable	If a scroll bar is embedded into a document, setting this to True causes the scroll bar to be printed when the document is printed.
ScrollValue	Current scroll value.
ScrollValueMax	Maximum scroll value of the control.
VisibleSize	Visible size of the scroll bar.

Note Add a horizontal scroll bar to the OOMESample dialog. Add the global variable oBPosSize to store a button's original position and size (see Listing 541).

To illustrate how the scroll bar functions, add a horizontal scroll bar to the sample dialog and name it ButtonScrollBar. All of the work done by a scroll bar control is done by a macro. A scroll bar doesn't know if something is horizontal or vertical, so it's just as easy to use a vertical scroll bar as a horizontal scroll bar for the same application—a person using a vertical scroll bar to move something horizontally might face ridicule by his peers, but the control doesn't care. I chose a horizontal scroll bar because I want to use it to move the Close button horizontally on the dialog. Add a global variable to store the initial position of the Close button (see Listing 541).

Listing 541. *Global variables for the Close button's initial position and size.*

```
Dim oBPosSize 'Button original position and size
```

In my sample dialog, there is nothing between the Close button and the right edge of the dialog (see Figure 138). I want the scroll bar to cause the Close button to move right and left.

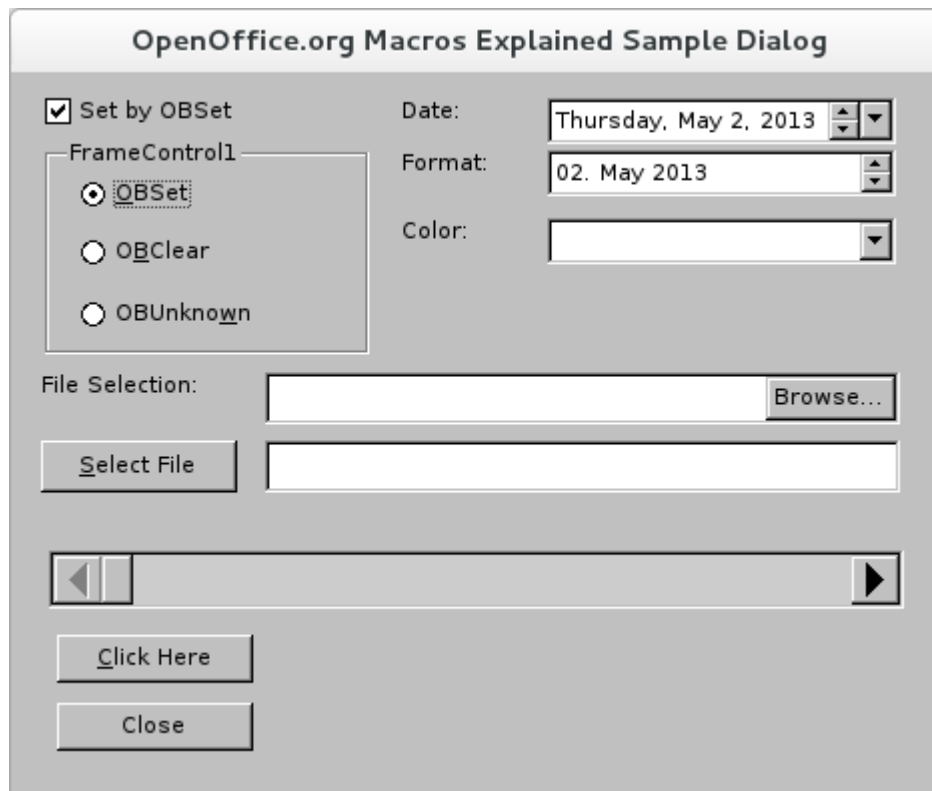


Figure 138. OOMESample dialog referred to throughout this chapter.

The maximum scroll bar value defaults to 100. This is usually used as a percentage to scroll something. In the sample dialog, however, I calculate the maximum distance that I can move the Close button, and then I set the maximum value that the scroll bar can return as this number. This allows me to use the scroll value directly, rather than calculating a position based on a percentage, which is the usual case.

In the initialization section, add code that saves the Close button’s initial position and size (see Listing 542); this information is used later as a reference point. Next, calculate how far the button can be shifted to the right based on the size and position of the button and the size of the dialog that contains the button.

Listing 542. Save the initial button position and set the scroll bar’s maximum value.

```
REM Get the Close button's original position and size.
REM Set the maximum value that the scroll bar can become
REM based on the maximum distance that I can move the Close button.
oBPosSize = oOOMEDlg.getControl("CloseButton").getPosSize()
oOOMEDlg.getModel().getByName("ButtonScrollBar").ScrollValueMax = _
    oOOMEDlg.getPosSize.Width - oBPosSize.X - oBPosSize.Width - 10
```

Set the scroll bar to call the ScrollButton method in Listing 543 for the “While Adjusting” event. The ScrollButton method uses the scroll value to position the Close button.

Listing 543. ScrollButton is in the DlgCode module.

```
Sub ScrollButton
    Dim oModel          'Scroll bar model
```

```

Dim oButton          'Close button
Dim newX As Long 'New X location for the Close button

oModel = oOOMEDlg.getModel().getByName("ButtonScrollBar")
oButton = oOOMEDlg.getControl("CloseButton")

REM The scroll bar was set so that its maximum value is
REM the maximum desired value.
newX = oBPosSize.X + oModel.ScrollValue
oButton.setPosSize(newX, 0, 0, 0, com.sun.star.awt.PosSize.X)
End Sub

```

Now you can move the Close button to the right and to the left by moving the scroll bar.

18.4. Using the Step property for multi-page AutoPilot dialogs

A multi-page dialog can display different sets of controls in the same dialog. Each set of controls is sometimes referred to as a “page.” OpenOffice.org frequently refers to multi-page dialogs as AutoPilot dialogs because they are frequently used to guide you through some process. At each step in the process, a different set of controls is displayed in the same dialog. For example, OOo comes with an ImportWizard macro library. The Main() subroutine in the Main module opens a dialog that assists in converting multiple documents. The first page (or step) allows you to select what document types will be imported. The second page (or step) allows you to select the directory that contains the documents to import.

The dialog model contains the Step property, which tells the dialog the current step in the process. If the dialog’s Step property is set to zero, the default value, then the dialog displays every control. The model for every control that the dialog contains, also supports the Step property. In a control, the Step property is used to determine at which step the control is displayed. If a control’s Step property is zero, the control is displayed for every step. A Cancel button is likely to be displayed at every step, for example.

18.5. The object inspector example

This section discusses an object inspector, that I use to inspect objects to understand what I can do with them. I am taking code that I have changed over the years to accomplish specific tasks, and significant improvements are possible; but this serves my needs.

The object inspector accepts an optional argument that identifies the object to inspect. If the object is a UNO object, then the object contains debug properties that list the supported properties, methods, services, and interfaces.

In this example, a dialog is created without using the Basic IDE. The inspection information is stored in a multi-line text edit control. I use a text edit control rather than a list box, because the text control allows me to copy the contained text to the clipboard and the list box control does not. I frequently inspect an object and then copy the supported properties, methods, and interfaces to the clipboard so that I can use them as a reference.

Radio buttons dictate what is displayed in the text control. The text control can display the supported properties, methods, interfaces and services, and general inspection information such as the object's data type (see Figure 139).

18.5.1. Utility subroutines and functions

Numerous utility methods are required to generically inspect an object in OO. The utility methods do not control, access, or update the dialog or any controls so I have separated them from the common functionality. The techniques and methods used here should be somewhat familiar, so I haven't discussed them in depth. All of the routines are contained in the Inspector module in this chapter's source code.

Identifying and removing white space from a string

A text character that appears as open space is called a "white space character." For example, in this book there is a single space between each word. The new-line character can create white space between lines. While inspecting an object, the resulting strings frequently contain leading and trailing white space of different types that must be removed. The functions in Listing 544 identify the different types of white space and remove them from a string.

Listing 544. Identify and remove white space from a string.

REM Utility Functions and Methods

```
*****
** Is the specified character whitespace? The answer is true if the
** character is a tab, CR, LF, space, or a non-breaking space character!
** These correspond to the ASCII values 9, 10, 13, 32, and 160
*****
Function IsWhiteSpace(iChar As Integer) As Boolean
    Select Case iChar
        Case 9, 10, 13, 32, 160
            IsWhiteSpace = True
        Case Else
            IsWhiteSpace = False
    End Select
End Function

*****
** Find the first character starting at location i% that is not whitespace.
** If there are none, then the return value will be greater than the
** length of the string.
*****
Function FirstNonWhiteSpace(ByVal i%, s$) As Integer
    If i <= Len(s) Then
        Do While IsWhiteSpace(Asc(Mid$(s, i, 1)))
            i = i + 1
        If i > Len(s) Then
            Exit Do
        End If
    Loop
```

```

End If
FirstNonWhiteSpace = i
End Function

'*****
'** Remove white space text from both the front and the end of a string.
'** This modifies the argument string and it returns the modified string.
'** This removes all types of white space, not just a regular space.
'*****
Function TrimWhite(s As String) As String
    s = Trim(s)
    Do While Len(s) > 0
        If Not IsWhiteSpace(ASC(s)) Then Exit Do
        s = Right(s, Len(s) - 1)
    Loop
    Do While Len(s) > 0
        If Not IsWhiteSpace(ASC(Right(s, 1))) Then Exit Do
        s = Left(s, Len(s) - 1)
    Loop
    TrimWhite = s
End Function

```

Convert a simple object to a string

The CStr() method converts most standard data types to a string. Not all variables are easily converted to a string using CStr(), so the ObjToString() method attempts to make an intelligent conversion. The most troublesome variable types are Object and Variant, which can contain the value Empty or Null—these two cases are identified and printed in Listing 545. Variables of type Object cannot be trivially converted to a String, so they are not. The standard types are converted to a String by Listing 545. The ObjToString() method truncates the string at 100 characters.

Listing 545. String representation of an object.

```

Function ObjToString(oInsObj, optional arraySep$, _
    Optional maxDepth%, Optional CRReplace As Boolean) As String

    Dim iMaxDepth% ' Maximum number of array elements to display
    Dim sArraySep$ ' Separator used for array elements
    Dim s$         ' Contains the results.
    Dim bCRReplace As Boolean

    ' Optional arguments
    If IsMissing(CRReplace) Then
        bCRReplace = False
    Else
        bCRReplace = CRReplace
    End If

    If IsArray(oInsObj) Then
        If IsMissing(arraySep) Then
            sArraySep = CHR$(10)
        Else
            sArraySep = arraySep
        End If
    End If

```

```

End If

If IsMissing(maxDepth) Then
    iMaxDepth = 100
Else
    iMaxDepth = maxDepth
End If

s = ObjArrayToString(oInsObj, sArraySep, iMaxDepth)
Else
    ' Not an array
    Select Case VarType(oInsObj)
        Case 0
            s = "Empty"
        Case 1
            s = "Null"
        Case 2, 3, 4, 5, 7, 8, 11, 16 To 23, 33 To 37
            s = CStr(oInsObj)
        Case Else
            If IsUnoStruct(oInsObj) Then
                s = "[Cannot convert an UnoStruct to a string]"
            Else
                Dim sImplName$ : sImplName = GetImplementationName(oInsObj)
                If sImplName = "" Then
                    s = "[Cannot convert " & TypeName(oInsObj) & " to a string]"
                Else
                    s = "[Cannot convert " & TypeName(oInsObj) & _
                        " (" & sImplName & ") to a string]"
                End If
            End If
        End Select
    End Select
    If bCRReplace Then
        s = Replace(s, CHR$(10), "|", 1, -1, 1)
        s = Replace(s, CHR$(13), "|", 1, -1, 1)
    End If
    If Len(s) > 100 Then s = Left(s, 100) & "..."
End If
ObjToString = s
End Function

```

Arrays are identified and converted separately. One and two dimensional arrays are supported.

Listing 546. Convert an array to a string.

```

'*****
'** Convert an array of objects to a string if possible.
'**
'** oInsObj    - Object to Inspect
'** sArraySep  - Separator character for arrays
'** maxDepth  - Maximum number of array elements to show
'*****
Function ObjArrayToString(oInsObj, sArraySep$, Optional maxDepth%) As String
    Dim iMaxDepth%
    Dim iDimensions%

```



```

If IsMissing(maxDepth) Then
    iMaxDepth = 100
Else
    iMaxDepth = maxDepth
End If

iDimensions = NumArrayDimensions(oInsObj)
If iDimensions = 1 Then
    ObjArrayToString = ObjArray_1_ToString(oInsObj, iMaxDepth%, sArraySep)
ElseIf iDimensions = 2 Then
    ObjArrayToString = ObjArray_2_ToString(oInsObj, iMaxDepth%, sArraySep)
Else
    ObjArrayToString = "[Unable to convert array with dimension " & iDimensions & "]"
End If
End Function

'*****
'** Convert a 1-D array of objects to a string.
'**
'** oInsObj      - Object to Inspect
'** iMaxDepth    - Maximum number of array elements to show
'** sSep         - Separator character for arrays
'*****
Function ObjArray_1_ToString(oInsObj, iMaxDepth%, sSep$) As String
    Dim iDim_1%
    Dim iMax_1%
    Dim iCounter%
    Dim s$
    Dim sTempSep$

    sTempSep = ""
    iDim_1 = LBound(oInsObj, 1)
    iMax_1 = UBound(oInsObj, 1)
    Do While (iCounter < iMaxDepth AND iDim_1 <= iMax_1)
        iCounter = iCounter + 1
        s = s & sTempSep & "[" & iDim_1 & "]" " _
            & ObjToString(oInsObj(iDim_1), ", ")
        sTempSep = sSep
        iDim_1 = iDim_1 + 1
    Loop
    ObjArray_1_ToString = s
End Function

'*****
'** Convert a 2-D array of objects to a string.
'**
'** oInsObj      - Object to Inspect
'** iMaxDepth    - Maximum number of array elements to show
'** sSep         - Separator character for arrays
'*****
Function ObjArray_2_ToString(oInsObj, iMaxDepth%, sSep$) As String
    Dim iDim_1%, iDim_2%

```

```

Dim iMax_1%, iMax_2%
Dim iCounter%
Dim s$
Dim sTempSep$

sTempSep = ""
iDim_1 = LBound(oInsObj, 1)
iMax_1 = UBound(oInsObj, 1)
iMax_2 = UBound(oInsObj, 2)
Do While (iCounter < iMaxDepth AND iDim_1 <= iMax_1)
    iCounter = iCounter + 1
    iDim_2 = LBound(oInsObj, 2)
    Do While (iCounter < iMaxDepth AND iDim_2 <= iMax_2)
        s = s & sTempSep & "[" & iDim_1 & ", " & iDim_2 & "]" " _
            & ObjToString(oInsObj(iDim_1, iDim_2), ", ")
        sTempSep = sSep
        iDim_2 = iDim_2 + 1
    Loop
    iDim_1 = iDim_1 + 1
Loop
ObjArray_2_ToString = s
End Function

'*****
'** Number of dimension for an array. a(4) returns 1 and a(3, 4) returns 2.
'**
'** oInsObj      - Object to Inspect
'*****
Function NumArrayDimensions(oInsObj) As Integer
    Dim i% : i = 0
    If IsArray(oInsObj) Then
        On Local Error Goto DebugBoundsError:
        Do While (i% >= 0)
            LBound(oInsObj, i% + 1)
            UBound(oInsObj, i% + 1)
            i% = i% + 1
        Loop
        DebugBoundsError:
        On Local Error Goto 0
    End If
    NumArrayDimensions = i
End Function

```

The object implementation name is useful to have, and special care must be taken for objects that do not support the method.

Listing 547. *Get the object's implementation name (if it is an UNO Service).*

```

'*****
'** Get an objects implementation name if it exists.
'**
'** oInsObj      - Object to Inspect
'*****
Function GetImplementationName(oInsObj) As String

```

```

On Local Error GoTo DebugNoSet
Dim oTmpObj
oTmpObj = oInsObj
GetImplementationName = oTmpObj.getImplementationName()
DebugNoSet:
End Function

```

Object inspection using Basic methods

The `ObjInfoString()` method (shown in Listing 548) creates a string that describes an object. This string includes detailed information about the object type. If the object is an array, the array dimensions are listed. If possible, even the UNO implementation name is obtained. No other UNO-related information—such as supported properties or methods—is obtained.

Listing 548. *Get information about an object.*

```

'*****
'** Return a string that contains useful information about an object.
'** This includes detailed information about the object type. If the
'** object is an array, then the array dimensions are listed.
'** If possible, even the UNO implementation name is obtained.
'*****
Function ObjInfoString(oInsObj) As String
    Dim s As String

    REM We can always get the type name and variable type.
    s = "TypeName = " & TypeName(oInsObj) & CHR$(10) & _
        "VarType = " & VarType(oInsObj) & CHR$(10)

    REM Check for NULL and EMPTY
    If IsNull(oInsObj) Then
        s = s & "IsNull = True"
    ElseIf IsEmpty(oInsObj) Then
        s = s & "IsEmpty = True"
    Else
        If IsObject(oInsObj) Then
            s = s & "Implementation Name = " & GetImplementationName(oInsObj)
            s = s & CHR$(10) & "IsObject = True" & CHR$(10)
        End If
        If IsUnoStruct(oInsObj) Then s = s & "IsUnoStruct = True" & CHR$(10)
        If IsDate(oInsObj) Then s = s & "IsDate = True" & CHR$(10)
        If IsNumeric(oInsObj) Then s = s & "IsNumeric = True" & CHR$(10)
        If IsArray(oInsObj) Then
            On Local Error Goto DebugBoundsError:
            Dim i%, sTemp$
            s = s & "IsArray = True" & CHR$(10) & "range = ("
            Do While (i% >= 0)
                i% = i% + 1
                sTemp$ = LBound(oInsObj, i%) & " To " & UBound(oInsObj, i%)
                If i% > 1 Then s = s & ", "
                s = s & sTemp$
            Loop
            DebugBoundsError:
            On Local Error Goto 0

```

```

        s = s & " ) " & CHR$(10)
    End If
End If

s = s & "Value : " & CHR$(10) & ObjToString(oInsObj) & CHR$(10)
ObjInfoString = s
End Function

```

Sort an array

Many objects contain so many properties and implement so many methods, that when they are displayed in the text box, it is difficult to find a specific entry. Sorting the output makes it easier to find individual entries.

Assume that I have two arrays. The first array contains a list of names and the second array contains a list of ages. The third item in the age array contains the age of the third item in the name array. In computer science, these are referred to as “parallel arrays.” Parallel arrays are used in the object browser. When dealing with properties, the property type is stored in the first array, and the property name is stored in the second array. The same thing is done while dealing with the supported methods.

While creating the data for inspection, parallel arrays are created and maintained. I want to sort the information, but I can’t sort one array unless I sort all of the arrays. A typical solution to this problem is to create an index array that contains integers, which are used to index into the array. The index array is sorted rather than the actual data. For example, sort the array oItems()=(“B”, “C”, “A”). The sorted index array is (2, 0, 1), implying that oItems(2) is the first item, oItems(0) is the second item, and oItems(1) is the last item. The function SortMyArray() in Listing 549 sorts an array by using an index array.

Listing 549. *Indirectly Sort an array by moving indexes.*

```

'*****
'*** Sort the oItems() array by arranging the iIdx() array.
'*** If oItems() = ("B", "C", "A") Then on output, iIdx() = (2, 0, 1)
'*** This means that the item that should come first is oItems(2), second
'*** is oItems(0), and finally, oItems(1)
'*****
Sub SortMyArray(oItems(), iIdx() As Integer)
    Dim i As Integer      'Outer index variable
    Dim j As Integer      'Inner index variable
    Dim temp As Integer    'Temporary variable to swap two values.
    Dim bChanged As Boolean 'Becomes True when something changes
    For i = LBound(oItems()) To UBound(oItems()) - 1
        bChanged = False
        For j = UBound(oItems()) To i+1 Step -1
            If oItems(iIdx(j)) < oItems(iIdx(j-1)) Then
                temp = iIdx(j)
                iIdx(j) = iIdx(j-1)
                iIdx(j-1) = temp
                bChanged = True
            End If
        Next j
    Next i
End Sub

```

```

Next
If Not bChanged Then Exit For
Next
End Sub

```

18.5.2. Creating a dialog at run time

The Object Inspector uses Private variables for the dialog and frequently referenced objects. The dialog contains a progress bar that is updated while information is retrieved to fill the text control. The inspected object is also stored as a global variable because it is used in event handlers that have no other way to obtain the object. See Listing 550.

Listing 550. *Global variables defined in the Inspector module.*

```

Private oDlg          'Displayed dialog
Private oProgress     'Progress control model
Private oTextEdit     'Text edit control model that displays information
Private oObjects(100) 'Objects to inspect
Private Titles(100)   'Titles
Private iDepth%       'Which object / title to use

```

Unfortunately, dialogs created in the IDE can only be created and run from within Basic. It is possible, however, to create and use a dialog at run time in any programming language that can use the OOo UNO API. The dialog functionality is scheduled to be improved in OOo 2.0 so that you can design dialogs in an IDE and display them using languages other than Basic. The Inspect() method in Listing 551 creates a dialog and all of its contained controls without using the IDE.

Follow these steps to create and display the dialog:

1. Use CreateUnoService("com.sun.star.awt.UnoControlDialogModel") to create a dialog model and then set the model's properties.
2. Use the createInstance(name) method in the dialog model to create a model for each control that is inserted into the dialog model. Set the properties on the control's model and then use the insertByName(name, oModel) method—defined by the dialog model—to insert the control's model into the dialog's model. This step is important! To insert a control into a dialog, you must use the dialog model to create a control model, which is then inserted into the dialog model. You don't create or manipulate controls, only models.
3. Use CreateUnoService("com.sun.star.awt.UnoControlDialog") to create a dialog.
4. Use the setModel() method on the dialog to set the dialog's model.
5. Use CreateUnoListener(name) to create any desired listeners. Add the listener to the correct object; the correct object is usually a control.
6. Use CreateUnoService("com.sun.star.awt.Toolkit") to create a window toolkit object.
7. Use the createPeer(oWindow, null) method on the dialog to tell the dialog to create an appropriate window to use.
8. Execute the dialog.

Listing 551. *Create the dialog, the controls, and the listeners.*

```

Sub Inspect(Optional oInsObj, Optional title$)
    Dim oDlgModel          'The dialog's model

```

```

Dim oModel                'Model for a control
Dim oListener             'A created listener object
Dim oControl              'References a control
Dim iTabIndex As Integer 'The current tab index while creating controls
Dim iDlgHeight As Long   'The dialog's height
Dim iDlgWidth As Long    'The dialog's width
Dim sTitle$

iDepth = LBound(oObjects)
sTitle = ""
If Not IsMissing(title) Then sTitle = title

REM If no object is passed in, then use ThisComponent.
If IsMissing(oInsObj) Then
    oObjects(iDepth) = ThisComponent
    If IsMissing(title) Then sTitle = "ThisComponent"
Else
    oObjects(iDepth) = oInsObj
    If IsMissing(title) Then sTitle = "Obj"
End If
Titles(iDepth) = sTitle

iDlgHeight = 300
iDlgWidth  = 350

REM Create the dialog's model
oDlgModel = CreateUnoService("com.sun.star.awt.UnoControlDialogModel")
setProperties(oDlgModel, Array("PositionX", 50, "PositionY", 50, _
    "Width", iDlgWidth, "Height", iDlgHeight, "Title", sTitle))

createInsertControl(oDlgModel, iTabIndex, "PropButton", _
    "com.sun.star.awt.UnoControlRadioButtonModel", _
    Array("PositionX", 10, "PositionY", 10, "Width", 50, "Height", 15, _
    "Label", "Properties"))

createInsertControl(oDlgModel, iTabIndex, "MethodButton", _
    "com.sun.star.awt.UnoControlRadioButtonModel", _
    Array("PositionX", 65, "PositionY", 10, "Width", 50, "Height", 15, _
    "Label", "Methods"))

createInsertControl(oDlgModel, iTabIndex, "ServiceButton", _
    "com.sun.star.awt.UnoControlRadioButtonModel", _
    Array("PositionX", 120, "PositionY", 10, "Width", 50, "Height", 15, _
    "Label", "Services"))

createInsertControl(oDlgModel, iTabIndex, "ObjectButton", _
    "com.sun.star.awt.UnoControlRadioButtonModel", _
    Array("PositionX", 175, "PositionY", 10, "Width", 50, "Height", 15, _
    "Label", "Object"))

createInsertControl(oDlgModel, iTabIndex, "EditControl", _
    "com.sun.star.awt.UnoControlEditModel", _
    Array("PositionX", 10, "PositionY", 25, "Width", iDlgWidth - 20, _

```

```

        "Height", (iDlgHeight - 75), "HScroll", True, "VScroll", True, _
        "MultiLine", True, "HardLineBreaks", True))

REM Store the edit control's model in a global variable.
oTextEdit = oDlgModel.getByName("EditControl")

createInsertControl(oDlgModel, iTabIndex, "Progress", _
    "com.sun.star.awt.UnoControlProgressBarModel", _
    Array("PositionX", 10, "PositionY", (iDlgHeight - 45), _
        "Width", iDlgWidth - 20, "Height", 15, "ProgressValueMin", 0, _
        "ProgressValueMax", 100))

REM Store a reference to the progress bar
oProgress = oDlgModel.getByName("Progress")

REM Notice that I set the type to OK so I do not require an action
REM listener to close the dialog.
createInsertControl(oDlgModel, iTabIndex, "OKButton", _
    "com.sun.star.awt.UnoControlButtonModel", _
    Array("PositionX", Clng(25), "PositionY", iDlgHeight-20, _
        "Width", 50, "Height", 15, "Label", "Close", _
        "PushButtonType", com.sun.star.awt.PushButtonType.OK))

createInsertControl(oDlgModel, iTabIndex, "InspectButton", _
    "com.sun.star.awt.UnoControlButtonModel", _
    Array("PositionX", Clng(iDlgWidth / 2 - 50), "PositionY", iDlgHeight-20, _
        "Width", 50, "Height", 15, "Label", "Inspect Selected", _
        "PushButtonType", com.sun.star.awt.PushButtonType.STANDARD))

createInsertControl(oDlgModel, iTabIndex, "BackButton", _
    "com.sun.star.awt.UnoControlButtonModel", _
    Array("PositionX", Clng(iDlgWidth / 2 + 50), "PositionY", iDlgHeight-20, _
        "Width", 50, "Height", 15, "Label", "Inspect Previous", _
        "Enabled", False, _
        "PushButtonType", com.sun.star.awt.PushButtonType.STANDARD))

REM Create the dialog and set the model
oDlg = CreateUnoService("com.sun.star.awt.UnoControlDialog")
oDlg.setModel(oDlgModel)

REM the item listener for all of the radio buttons.
oListener = CreateUnoListener("radio_", "com.sun.star.awt.XItemListener")
oControl = oDlg.getControl("PropButton")
ocontrol.addItemListener(oListener)
oControl = oDlg.getControl("MethodButton")
ocontrol.addItemListener(oListener)
oControl = oDlg.getControl("ServiceButton")
ocontrol.addItemListener(oListener)
oControl = oDlg.getControl("ObjectButton")
ocontrol.addItemListener(oListener)
oControl.getModel().State = 1

```

```

oListener = CreateUnoListener("ins_", "com.sun.star.awt.XActionListener")
oControl = oDlg.getControl("InspectButton")
ocontrol.addActionListener(oListener)

oListener = CreateUnoListener("back_", "com.sun.star.awt.XActionListener")
oControl = oDlg.getControl("BackButton")
ocontrol.addActionListener(oListener)

REM Now, set the dialog to contain the standard dialog information
DisplayNewObject()

REM Create a window and then tell the dialog to use the created window.
Dim oWindow
oWindow = CreateUnoService("com.sun.star.awt.Toolkit")
oDlg.createPeer(oWindow, null)

REM Finally, execute the dialog.
oDlg.execute()
End Sub

```

To shorten the code in Listing 551, try using two utility methods to set properties, create a control model, and insert the control model into the dialog model. Notice that the control itself is never created—only the model is created. Listing 552 contains the methods to create the control models and to set the properties. Although properties are set using the `setProperty` method, in Basic you can set the properties directly if desired.

Listing 552. Create a control model and insert it into the dialog's model.

```

'*****
'** Create a control of type sType with name sName, and insert it into
'** oDlgModel.
'**
'** oDlgModel - Dialog model into which the control will be placed.
'** index      - Tab index, which is incremented each time.
'** sName      - Control name.
'** sType      - Full service name for the control model.
'** props      - Array of property name / property value pairs.
'*****
Sub createInsertControl(oDlgModel, index%, sName$, sType$, props())
    Dim oModel

    oModel = oDlgModel.createInstance(sType$)
    setProperties(oModel, props())
    setProperties(oModel, Array("Name", sName$, "TabIndex", index%))
    oDlgModel.insertByName(sName$, oModel)

    REM This changes the value because it is not passed by value.
    index% = index% + 1
End Sub

```

The following macro is used to set many properties based on a single call.

Listing 553. Set many properties at one time.

REM Generically set properties based on an array of name/value pairs.


```

Sub setProperties(oModel, props())
    Dim i As Integer
    For i=LBound(props()) To UBound(props()) Step 2
        oModel.setPropertyValue(props(i), props(i+1))
    Next
End sub

```

18.5.3. Listeners

Radio buttons

Each time a new radio button is selected, the method `radio_itemStateChanged()` is called (the listener is set up in Listing 551). The event handler calls the `DisplayNewObject()` subroutine, which handles the actual work when a new property type is requested. See Listing 554. I wrote a separate routine so that it can be used apart from the event handler. For example, it calls `DisplayNewObject()` before the dialog is displayed so that the dialog contains useful information when it first appears.

Listing 554. *The event handler is very simple; it calls `DisplayNewObject()`.*

```

REM This method is to support the com.sun.star.awt.XItemListener interface
REM for the radio button listener. This method is called when a radio button
REM is selected.

```

```

Sub radio_itemStateChanged(oItemEvent)
    DisplayNewObject()
End Sub

```

```

Sub DisplayNewObject()
    REM Reset the progress bar!
    oProgress.ProgressValue = 0
    oTextEdit.Text = ""

    On Local Error GoTo IgnoreError:
    If IsNull(oObjects(iDepth)) OR IsEmpty(oObjects(iDepth)) Then
        oTextEdit.Text = ObjInfoString(oObjects(iDepth))
    ElseIf oDlg.getModel().getByName("PropButton").State = 1 Then
        processStateChange(oObjects(iDepth), "p")
    ElseIf oDlg.getModel().getByName("MethodButton").State = 1 Then
        processStateChange(oObjects(iDepth), "m")
    ElseIf oDlg.getModel().getByName("ServiceButton").State = 1 Then
        processStateChange(oObjects(iDepth), "s")
    Else
        oTextEdit.Text = ObjInfoString(oObjects(iDepth))
    End If
    oProgress.ProgressValue = 100

    IgnoreError:
End Sub

```

Inspect selected

The handler for the Inspect Selected button looks at the text that is selected and then based on the selected text, attempts to inspect a new object. I considered making the processing more user

friendly, but it meets my needs. If you select text, a property with the selected name is inspected. If you select a number, the current object is assumed to be an array, and the object at the specified index is inspected.

Listing 555. *Inspect a new item.*

```
Sub Ins_actionPerformed(oActionEvent)
    Dim oControl
    Dim oSel
    Dim sText$
    Dim v
    Dim oOldObj
    Dim sOldTitle$
    Dim sNewTitle$

    If iDepth = UBound(oObjects) Then
        Print "Sorry, already nested too deeply"
        Exit Sub
    End If

    oControl = oDlg.getControl("EditControl")
    sText = oControl.getSelectedText()
    sOldTitle = oDlg.Title
    sNewTitle = sOldTitle & "/" & sText

    If IsNumeric(sText) Then
        ' oDlg.getModel().getByName("ObjButton").State = 1
        If NumArrayDimensions(oObjects(iDepth)) <> 1 Then
            Print "You can only select a number with a one dimensional array."
        Else
            oOldObj = oObjects(iDepth)
            iDepth = iDepth + 1
            oDlg.getControl("BackButton").GetModel().Enabled = True
            oObjects(iDepth) = oOldObj(sText)
            oDlg.Title = sNewTitle
            Titles(iDepth) = sNewTitle
            DisplayNewObject()
        End If
    Else
        v = RunFromLib(GlobalScope.BasicLibraries, _
            "xyzzylib", "TestMod", oObjects(iDepth), sText, False, False)
        If IsNull(v) OR IsEmpty(v) Then
            '
        Else
            iDepth = iDepth + 1
            oDlg.getControl("BackButton").GetModel().Enabled = True
            oObjects(iDepth) = v
            oDlg.Title = sNewTitle
            Titles(iDepth) = sNewTitle
            DisplayNewObject()
        End If
    End If
    'oSel = oControl.getSelection ()
```

```

'Print "Min: " & oSel.Min & " Max: " & oSel.Max
'sText = oControl.getText()
'Inspect oControl
End Sub

```

Inspect previous

A stack of inspected objects is maintained. The “previous item” means the last item on the stack.

Listing 556. Inspect a previous item.

```

Sub back_actionPerformed(oActionEvent)
    If iDepth <= LBound(oObjects) Then
        Exit Sub
    End If
    iDepth = iDepth - 1
    If iDepth = LBound(oObjects) Then
        oDlg.getControl("BackButton").GetModel().Enabled = False
    End If

    oDlg.Title = Titles(iDepth)
    DisplayNewObject()
End Sub

```

18.5.4. Obtaining the debug information

In Listing 554, notice that if the properties, methods, or services are requested, then the processStateChange() subroutine (see Listing 557) is called. If properties, methods, or services are not requested, simple object information is added to the text control.

Again, the processStateChange() subroutine contains very little logic. The primary purpose of Listing 557 is to add the supported services to the end of the text control after the interfaces are obtained from the dbg_SupportedInterfaces property.

Listing 557. Build the text string that is displayed about the object.

```

'*****
'*** Set the text. For a service, show the interfaces and services in
'*** Separate sections so that they are easier to find.
'*****
Sub processStateChange(oInsObj, sPropType$)
    Dim oItems()
    BuildItemArray(oInsObj, sPropType$, oItems())
    If sPropType$ = "s" Then
        Dim s As String
        On Local Error Resume Next
        s = "*** INTERFACES ***" & CHR$(10) & Join(oItems, CHR$(10))
        s = s & CHR$(10) & CHR$(10) & "*** SERVICES ***" & CHR$(10) & _
            Join(oInsObj.getSupportedServiceNames(), CHR$(10))
        oTextEdit.Text = s
    Else
        oTextEdit.Text = Join(oItems, CHR$(10))
    End If
End Sub

```

The BuildItemArray() subroutine contains the majority of the important code for this dialog. First, the code inspects the dbg_properties to obtain the list of properties, methods, or interfaces as a single string. It then separates the individual items from the single string and sorts them. If properties are displayed, the property set information object is used to obtain the value of each supported property. This allows you to see both the property name and its value. Although the property set information object does not support all of the properties returned by the property dbg_properties, it supports most of them. The individual data is returned as an array of strings in the oItems() array. See Error: Reference source not found for an example.

Listing 558. Build an array of properties, methods, or services.

```

*****
'** This routine parses the strings returned from the dbg_Methods,
'** dbg_Properties, and Dbg_SupportedInterfaces calls. The interesting
'** data starts after the first colon.
'**
'** Because of this, all data before the first colon is discarded
'** and then the string is separated into pieces based on the
'** separator string that is passed in.
'**
'** All instances of the string Sbx are removed. If this string
'** is valid and exists in a method name, it will still be removed so perhaps
'** it is not the safest thing to do, but I have never seen this case and it
'** makes the output easier to read.
'**
'** the oItems() contains all of the parsed sections on output.
*****
Sub BuildItemArray(oInsObj, sType$, oItems())
    On Error Goto BadErrorHere
    Dim s As String          'Primary list to parse
    Dim sSep As String       'Separates the items in the string
    Dim iCount%              '
    Dim iPos%                '
    Dim sNew$                '
    Dim i%                    '
    Dim j%                    '
    Dim sFront() As String   'When each piece is parsed, this is the front
    Dim sMid() As String     'When each piece is parsed, this is the middle
    Dim iIdx() As Integer    'Used to sort parallel arrays.
    Dim nFrontMax As Integer 'Maximum length of front section
    Dim sArraySep$           : sArraySep = ", "
    Dim iArrayMaxDepth%      : iArrayMaxDepth = 10

    nFrontMax = 0

    REM First, see what should be inspected.
    s = ""
    On Local Error Resume Next
    If sType$ = "s" Then
        REM Dbg_SupportedInterfaces returns interfaces and
        REM getSupportedServiceNames() returns services.
        s = oInsObj.Dbg_SupportedInterfaces

```

```

's = s & Join(oInsObj.getSupportedServiceNames(), CHR$(10))
sSep = CHR$(10)
ElseIf sType$ = "m" Then
    s = oInsObj.DBG_Methods
    sSep = ";"
ElseIf sType$ = "p" Then
    s = oInsObj.DBG_Properties
    sSep = ";"
Else
    s = ""
    sSep = ""
End If

REM The dbg_ variables have some introductory information that
REM we do not want so remove it.
REM We only care about what is after the colon.
iPos = InStr(1, s, ":") + 1
If iPos > 0 Then s = TrimWhite(Right(s, Len(s) - iPos))

REM All data types are prefixed with the text Sbx.
REM Remove all of the "SbX" characters
s = Join(Split(s, "Sbx"), "")

REM If the separator is NOT CHR$(10), then remove
REM all instances of CHR$(10)
If ASC(sSep) <> 10 Then s = Join(Split(s, CHR$(10)), "")

REM split on the separator character and update the progress bar.
oItems() = Split(s, sSep)
oProgress.ProgressValue = 20

Rem Create arrays to hold the different portions of the text.
Rem the string usually contains text similar to "SbxString getName()"
Rem sFront() holds the data type if it exists and "" if it does not.
Rem sMid() holds the rest
ReDim sFront(UBound(oItems)) As String
ReDim sMid(UBound(oItems)) As String
ReDim iIdx(UBound(oItems)) As Integer

REM Initialize the index array and remove leading and trailing
REM spaces from each string.
For i=LBound(oItems()) To UBound(oItems())
    oItems(i) = Trim(oItems(i))
    iIdx(i) = i
    j = InStr(1, oItems(i), " ")
    If (j > 0) Then
        REM If the string contains more than one word, the first word is stored
        REM in sFront() and the rest of the string is stored in sMid().
        sFront(i) = Mid$(oItems(i), 1, j)
        sMid(i) = Mid$(oItems(i), j+1)
        If j > nFrontMax Then nFrontMax = j
    Else
        REM If the string contains only one word, sFront() is empty

```

```

    REM and the string is stored in sMid().
    sFront(i) = ""
    sMid(i) = oItems(i)
End If
Next

oProgress.ProgressValue = 40
Rem Sort the primary names. The array is left unchanged, but the
Rem iIdx() array contains index values that allow a sorted traversal
SortMyArray(sMid(), iIdx())
oProgress.ProgressValue = 50

REM Dealing with properties so attempt to find the value
REM of each property.
If sType$ = "p" Then
    Dim oPropInfo 'PropertySetInfo object
    Dim oProps    'Array of properties
    Dim oProp     'com.sun.star.beans.Property
    Dim v         'Value of a single property
    Dim bHasPI As Boolean
    Dim bUsePI As Boolean
    Dim bConvertReturns As Boolean

    bConvertReturns = True
    bHasPI = False
    bUsePI = False
    On Error Goto NoPropertySetInfo
    oPropInfo = oInsObj.getPropertySetInfo()
    bHasPI = True
NoPropertySetInfo:

    On Error Goto BadErrorHere:

    For i=LBound(sMid()) To UBound(sMid())
        If bHasPI Then
            bUsePI = oPropInfo.hasPropertyByName(sMid(i))
        End If

        If bUsePI Then
            v = oInsObj.getPropertyValue(sMid(i))
        Else
            v = RunFromLib(GlobalScope.BasicLibraries, _
                "xyzzylib", "TestMod", oInsObj, sMid(i), False, False)
        End If
        sMid(i) = sMid(i) & " = " & _
            ObjToString(v, sArraySep, iArrayMaxDepth, bConvertReturns)
    Next
End If
oProgress.ProgressValue = 60

nFrontMax = nFrontMax + 1
iCount = LBound(oItems())
REM Now build the array of the items in sorted order

```

```

REM Sometimes, a service is listed more than once.
REM this routine removes multiple instances of the same service.
For i = LBound(oItems()) To UBound(oItems())
    sNew = sFront(iIdx(i)) & " " & sMid(iIdx(i))
    'Uncomment these lines if you want to add uniform space
    'between front and mid. This is only useful if the font is fixed in width.
    'sNew = sFront(iIdx(i))
    'sNew = sNew & Space(nFrontMax - Len(sNew)) & sMid(iIdx(i))
    If i = LBound(oItems()) Then
        oItems(iCount) = sNew
    ElseIf oItems(iCount) <> sNew Then
        iCount = iCount + 1
        oItems(iCount) = sNew
    End If
Next
oProgress.ProgressValue = 75
ReDim Preserve oItems(iCount)
Exit Sub
BadErrorHere:
MsgBox "Error " & err & ": " & error$ + chr(13) + "In line : " + Erl
End Sub

```

The object inspector is shown below. ThisComponent was inspected, I viewed properties, selected “ChapterNumberingRules”, and then I clicked **Inspect Selected**. The path is shown in the dialog title.

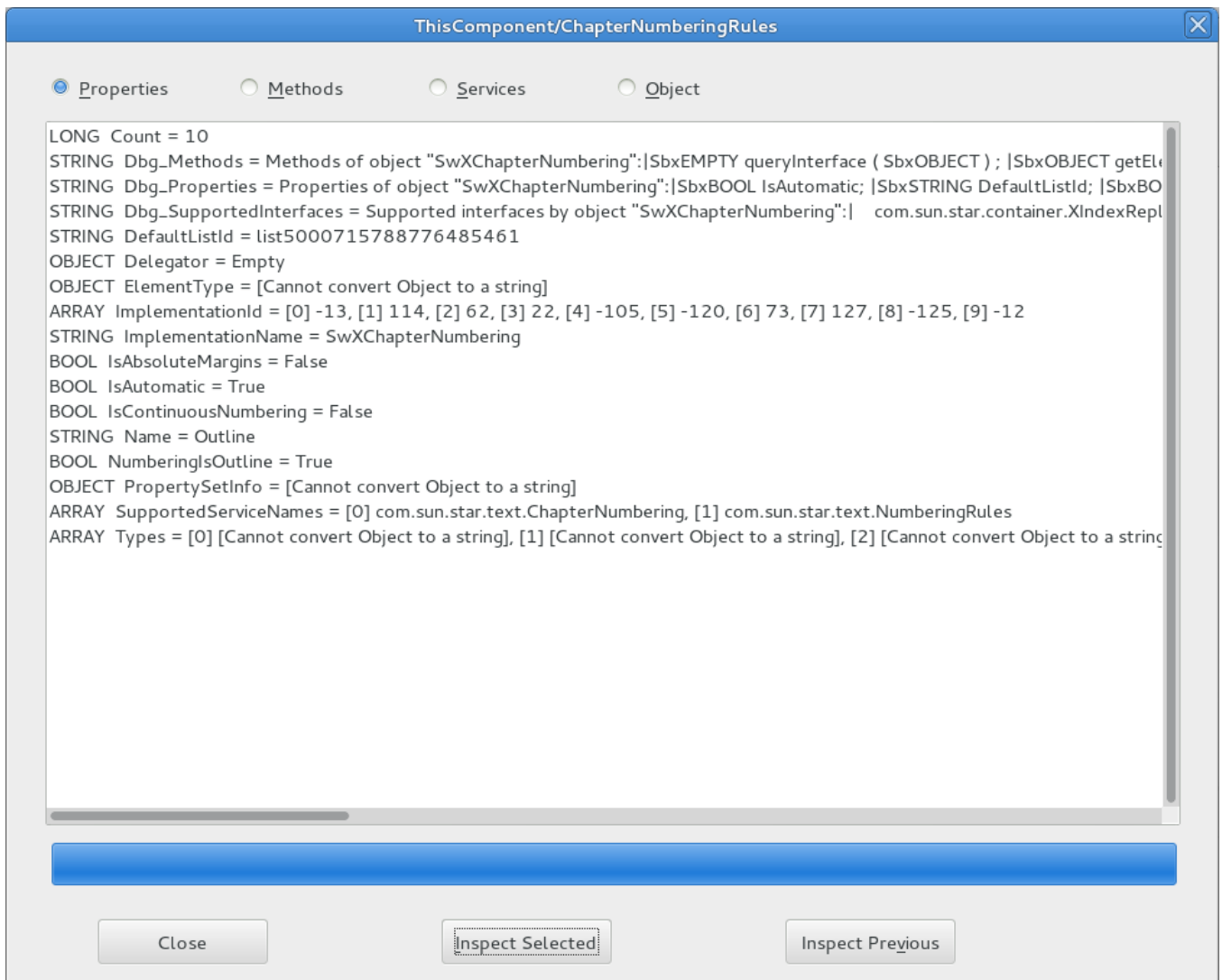


Figure 139. The Object Inspector dialog.

You can use the object inspector to investigate objects when you aren't certain exactly what you can do with them. You might want to copy this to an application-level library so that you can access the macro from all of your other macros.

18.5.5. Getting property values

Most UNO objects have a PropertySetInfo object, which allows you to see what properties exist, get the properties, and set the properties. Oddly, not all of the properties are available through this interface, but they are available from the Dbg_Properties property. When the PropertySetInfo object is unable to get a property value, the RunFromLib method is used.

The following macro creates a library, module, and function and then calls it. The library and module name are provided. The module is deleted if it exists, and replaced with a single function. The primary reason is to access a named property and return the value of the named property.

Listing 559. *Create a library, module, and function, then call it.*

```
*****
'** Create a module that contains a function that returns the value from a
'** property or makes a method call on an object.
'**
'** Not all properties are available from the PropertySetInfo object, and
'** I am not aware of any other way to make a method call.
'**
'** oLibs   - Library container to use
'** sLName  - Library name
'** sMName  - Module name
'** oObj    - Object of interest
'** sCall   - Method or property name to call
'** bClean  - If true, remove the module. Library is only removed if created.
'** bIsMthd - If true, create as a method rather than a property.
'** x       - If exists, then used as parameter for the method call.
'** y       - If exists, then used as parameter for the method call.
'**
*****
Function RunFromLib(oLibs, sLName$, sMName$, oObj, sCall$, _
                   bClean As Boolean, bIsMthd As Boolean, _
                   Optional x, optional y)
    Dim oLib      'The library to use to run the new function.
    Dim s$        'Generic string variable.
    Dim bAddedLib As Boolean

    REM If the library does not exist, then create it.
    bAddedLib = False
    If NOT oLibs.hasByName(sLName) Then
        bAddedLib = True
        oLibs.createLibrary(sLName)
    End If
    oLibs.loadLibrary(sLName)
    oLib = oLibs.getByName(sLName)

    If oLib.hasByName(sMName) Then
        oLib.removeByName(sMName)
    End If

    s = "Option Explicit" & CHR$(10) & _
        "Function MyInsertedFunc(ByRef oObj"

    If NOT IsMissing(x) Then s = s & ", x"
    If NOT IsMissing(y) Then s = s & ", y"
    s = s & ")" & CHR$(10) & "On Local Error Resume Next" & CHR$(10)
    s = s & "MyInsertedFunc = oObj." & sCall

    If bIsMthd Then
        s = s & "("
        If NOT IsMissing(x) Then s = s & "x"
        If NOT IsMissing(y) Then s = s & ", y"
        s = s & ")"
    End If
```

```

s = s & CHR$(10) & "End Function"

oLib.insertByName(sMName, s)

If IsMissing(x) Then
    RunFromLib = MyInsertedFunc(oObj)
ElseIf IsMissing(y) Then
    RunFromLib = MyInsertedFunc(oObj, x)
Else
    RunFromLib = MyInsertedFunc(oObj, x, y)
End If

If bClean Then
    oLib.removeByName(sMName)
    If bAddedLib Then
        oLibs.removeLibrary(sLName)
    End If
End If
End Function

```

Warn

While debugging the code used to inspect objects, the system was left in a strange state, and removing the existing module failed. I believe that this was related to calling a handler containing breakpoints and then killing the dialog while the breakpoints were active. The important thing to note, however, is that if the module cannot be deleted, stop the object inspector and then remove the library named “xyzzylib” using the macro organizer.

18.6. Conclusion

The methods outlined in this chapter should get you started in using dialogs and controls. The methods to access and use controls in dialogs are very similar to accessing controls in forms, so you’re well on your way to using forms as well. When a control is stored in a document rather than a dialog, the control is stored in a form.

19. Sources of Information

The internals of OpenOffice.org are very extensive, not completely documented, and in a state of change due to ongoing feature development and bug fixes. This chapter introduces sources of information that can assist you in finding your own solutions.

While solving any problem, it is most important to have a basic understanding of the problem, and to know how to find the relevant information that you don't know about the problem and possible solutions. You might produce faster and more reliable code if you knew everything about OpenOffice.org, but this simply isn't possible because of the broad scope of the product. Even if you could know it all, and keep up with the changes of successive releases of OOo, it is probably more productive to use some of that mental energy to remember the phone numbers of your closest friends, your wedding anniversary, or funny stories about your parents—or to remember how to order extra copies of this book as gifts for your friends, spouse, and parents!

There are many excellent sources of information on OOo, each addressing different needs. Knowing which information is available at which location can save a lot of time and effort in solving problems.

19.1. Help pages included with OpenOffice.org

Do not neglect the excellent help that is included with OOo (see Figure 140). Although it may seem like I'm frequently complaining about inaccurate or missing items in the help pages, in reality those are few and far between. I mention this simply because it's important to note when there are deficiencies, so you can be prepared to deal with some of the inevitable rough spots. (Hey, if I didn't offer something of original value, what would be the point in writing this book? After all, I have a day job that keeps me busy—I don't want this book to be a waste of time for either of us!)

The help pages cover each of the components in OOo: Writer, Calc, Basic, Draw, Math, and Impress. The upper-left corner of the help page contains a drop-down list that determines which help set is displayed. To view the help for Basic, the drop-down must display "Help about OpenOffice.org Basic." If you open the help window from the Basic IDE, this is the default. The included help contains a lot of current information on Basic syntax and routines.



Figure 140. The included help is very good.

19.2. Macros included with OpenOffice.org

Many macros are included with OOo in libraries. Spend some time investigating these macros. Many wonderful examples are included. For example, the Gimmicks library contains a ReadDir module that reads an entire directory and creates a draw page with the information displayed in a tree structure. The Tools library also contains numerous excellent examples. The Debug module in the Tools library contains some useful macros, such as WriteDbgInfo and PrintDbgInfo—check them out.

If you don't know how to solve a problem, you should always try to find the answer first, rather than writing a completely new solution. The macros included with OOo are a good place to start. I once posted a request for help to determine the size of a control in a dialog. The response, which included the solution, indicated that I should look in the ModuleControls module in the Tools library that is included with OOo. In other words, the macros included with OOo contained the solution to my problem.

TIP	You can see the libraries and modules included with OOo by selecting Tools Macros Organize Macros to open the Macro dialog. The macros available at the application level are in the OpenOffice library container.

19.3. Web sites

On more than one occasion, I knew what I needed to use to solve a problem, but I just couldn't figure out the details. This is where a working example is invaluable. There are numerous sources for help on the Internet; more become available all the time. Here is a list of sites that may help you:

TIP	Although I have attempted to provide accurate links and descriptions, the OpenOffice Web sites are constantly changing and improving.

19.3.1. Reference material

<http://www.openoffice.org/api/> is the site where I spend most of my time. This site lists most interfaces and services and contains the primary Developer's Guide. The guide has a reputation for being difficult to understand, but the authors are working to make the content more accessible to less technical readers.

<http://www.odfauthors.org/> Documentation and books.

<http://wiki.openoffice.org/wiki/Documentation>: Apache documentation project.

19.3.2. Macro examples

<http://www.pitonyak.org/oo.php> is my personal Web site, created before I decided to write a book. This site contains numerous documents and links. "AndrewMacro" contains numerous examples. The direct link to my macro document is <http://www.pitonyak.org/AndrewMacro.odt>.

<http://kienlein.com/pages/oo.html> contains some excellent macros, with short descriptions of the macros in German.

http://www.darwinwars.com/lunatic/bugs/oo_macros.html is a good first stop when looking for Writer

macros. The author of this site is a professional writer who uses the macros—so they work!

19.3.3. Miscellaneous

<http://www.openoffice.org/> is the main OOo link. This site contains a lot of information, such as links to mailing lists that you can search, and the OOo bug-tracking system IssueZilla. You can join the site (it's free) to gain more capabilities, such as entering bug reports and subscribing to mailing lists.

<http://www.openoffice.org/development/index.html> is the developer project link from the main OOo Web page. Numerous links and resources are available from this page.

<http://forum.openoffice.org/> The official OpenOffice forum.

<http://www.ooforum.org/> contains numerous help forums. This site includes help for the different components of OOo—including macro programming.

<http://oodocs.sourceforge.net/> contains many help files separated into categories, including a section on macro programming.

<http://www.openoffice.org/documentation/> is the home page for the documentation project. This site contains documentation on numerous topics. The How-To link references a document that discusses how to embed controls that call macros into documents. This is an excellent interactive document.

http://documentation.openoffice.org/HOW_TO/various_topics/How_to_use_basic_macros.sxw

http://sourceforge.net/project/showfiles.php?group_id=43716 contains numerous templates and examples.

19.1. <http://www.openoffice.org/api/> or <http://api.libreoffice.org>

For the serious macro writer, the site <http://www.openoffice.org/api/> is very important. I spend most of my time here while investigating how to solve problems. The first accessible programming document for StarOffice version 5.x is available from this site. The document, www.openoffice.org/api/basic/man/tutorial/tutorial.pdf, may be old, but it contains some nice examples—and it is very accessible.

<http://wiki.openoffice.org/wiki/Documentation/DevGuide/>

OpenOffice.org Developers Guide contains the Developer's Guide, which is kept up-to-date. The target audience for this document is the professional developer. A lot of information is here, and it requires some talent to find what you need. This is only because so much information is available. Basic programming is only a small portion of this document's coverage. When you require the complete story, or at least the most complete story available, look in the Developer's Guide. (But, be prepared for the *complete* story...)

I spend most of my time in the portion investigating the individual modules, which list the interfaces and services available. A good starting point is

<http://www.openoffice.org/api/docs/common/ref/module-ix.html>

because it contains all of the main modules. Notice that the name of the Web page is “module-ix.html”. If you are looking at a Web page for a service or interface, you can usually remove the name of the Web page and substitute the name “module-ix.html”. For example, assume that you

are looking at the TextCursor service at <http://www.openoffice.org/api/docs/common/ref/com/sun/star/text/TextCursor.html> and you want to see the services. Change “TextCursor.html” to “module-ix.html” and you can see all of the definitions, services, contained modules, constants, and interfaces in the com.sun.star.text module.

You should start investigating the different modules at the main Sun module, <http://www.openoffice.org/api/docs/common/ref/com/sun/star/module-ix.html>, which will provide a feel for the different modules and their purposes. Because the modules group related functionality, I sometimes look here when I need to find a service that solves a specific problem but I don’t know what the service is.

19.2. Mailing lists and newsgroups

Numerous mailing lists are available for OpenOffice users. Information on each mailing list is available from the lists-related Web site.

- http://www.openoffice.org/mail_list.html
- <http://www.libreoffice.org/get-help/mailing-lists/>

When you ask a question on a mailing list, the answers go to the mailing list rather than directly to you. You must subscribe to the mailing list so that you can receive the answers. Instructions are provided to subscribe to each of the mailing lists. Although developers read the mailing lists and answer questions, this is not a support forum in the sense of guaranteed help. If other users or developers have a solution or an idea for a solution, they will post an answer. Answers requiring a lot of work or time are not likely to be answered. The LO development mailing list is expected to be used by people developing the LO software so you may not have much luck asking macro related questions on that mailing list.

A good first step before asking a question is to subscribe to the mailing list. Remember, the people answering your questions are not paid to do so. It is appreciated if you answer other people’s questions if you know the answer. Here are some hints to ensure that you receive answers on the mailing lists.

Before asking a question, try to find the answer yourself. Search the Web, search the mailing-list archives, search the Developer’s Guide, search Google, read a FAQ. Inspect returned objects. The user community members prefer to help people who can help themselves and give back to the community.

Choose a meaningful message header. I receive hundreds of e-mails every day so I am likely to skip a message that is titled “Please Help,” “Urgent,” or “Macro Problem.” A better title is “How do I run a macro from the command line?” or “Searching macro fails to find attributes.” I can then look at questions that I am likely to already understand or that interest me. We all get more e-mail than we need. Vague or misleading subject lines often inspire liberal use of the Delete feature.

Carefully word your question so that you ask the right question. “My macro has a run-time error” is rather vague. Including a little code certainly helps. Including the run-time error in your message also helps. The fact that your sort macro does not sort as it should might be specific, but I need more information and perhaps some code before I can comment. State the problem precisely, and include how you tried to solve

the problem. Specific error messages—and anything you know about the circumstances or likely causes of the problem—are welcome.

Do not expect someone else to do your work for you. “My client is paying me \$2000 to convert this macro; will you do it for free?” is not likely to receive a response. I have translated macros from other systems because they were interesting and of broad scope, and I was allowed to then release them to the user community. If you are really in a bind, someone on the list might be willing to do your work for a fee.

Do not directly ask a single individual, even if they are knowledgeable. Many people can answer your question using pooled knowledge, and you will often receive a better solution. In other words, ask the question on the list rather than directly to an individual, unless you have a good reason for doing so. The last question that I helped answer on the list generated roughly 10 messages from four different people.

It’s considered bad form to request an answer to be e-mailed to you at a different address. If you don’t care enough to check the mailing list for the answer, why should someone else care enough to spend time answering your question in the first place? Again, remember that these folks are volunteers! Also, the answer to your question—an online discussion leading to the understanding of your circumstances and eventual solution—may be of value to others in the community. In this way, the shared pool of knowledge grows larger and better over time.

When you do have a solution, post it to help others. A polite “thank you” is also appropriate. Again, courtesy and sharing are important to the development of a functioning community that benefits you and all the other members as well. (I feel like a public service announcement when I say these things, just in case you were wondering.)

19.3. How to find answers

When I want to know about a specific issue, I usually search the Developer’s Guide and the old StarOffice tutorial, and finally I perform a Google search such as “cursor OpenOffice”. When I want to narrow the search to the <http://www.openoffice.org/api/docs> site, I use “site:<http://www.openoffice.org/api/docs> cursor”; the results usually include an interface or a service that I can inspect, such as:

```
MsgBox vObj.dbg_methods
MsgBox vObj.dbg_supportedInterfaces
MsgBox vObj.dbg_properties
```

Libreoffice has a similar site at <http://api.libreoffice.org>, so you can search that as well.

If you aren’t certain what to do with an object, inspect it. For example, print the properties `dbg_methods`, `dbg_supportedInterfaces`, and `dbg_properties`. If you are still at a loss, you can search the Internet or other documents for the services, methods, or properties that are implemented.

TIP

I don’t really print the methods and properties, I use the object inspector or X-Ray (see section 18.5 The object inspector example).

Quickly search for examples that might solve the same problem that you face. This is an ideal situation because it requires less work on your part. As a last resort, post a message on a mailing list and ask for help.

19.4. Conclusion

Many excellent sources of information discuss how to write code that directs OpenOffice.org. Becoming familiar with these sources can save you a lot of time, and regularly browsing the mailing lists also keeps you in touch with how things are changing. Participation in the community of a mailing list or newsgroup offers opportunity to gain solutions and to allow others to benefit from your experience as well.