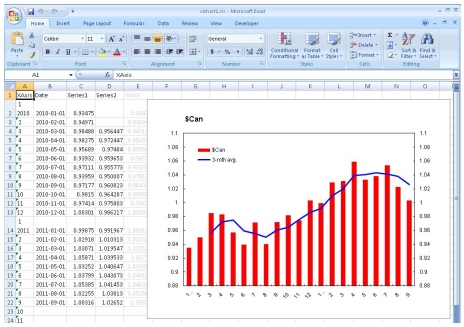
Charting the APL/Excel Waters

Using Excel and other Applications with Dyalog APL

Richard Procter, APL Borealis Inc., Canada
rjp@aplborealis.com



Contents

Terminology and TLAs

We will explore how APL can interact with and use some of these technologies:

COM - Component Object Model DCOM - Distributed COM	(*MS 2000 Automation Help:*) an industry-standard technology that applications use to expose their objects, methods, and properties to development tools, macro languages, and other applications. (DCOM = COM extended for network apps.) (*Wikipedia:*) ...often used in the software development world as an umbrella term that encompasses the OLE, OLE Automation, ActiveX, COM+ and DCOM technologies.
OLE - Object Linking and Embedding & OLE Automation	sharing the properties and methods of applications by combining and exposing objects within a standard framework (eg. document) (*Wikipedia:*) the formal interprocess communication mechanism based on COM. It provides an infrastructure whereby applications can access and manipulate (i.e. set properties of, or call methods on) shared **automation objects** that are exported by other applications. It supersedes DDE. The OLE Automation controller is the "client" and the application exporting the automation objects is the "server".
OLE Server	a usually invisible application which supports the main interface (client) (*MFC:*) an Automation server is an application (a type of COM server) that exposes its functionality through COM interfaces to other applications, called Automation clients. The exposure enables Automation clients to automate certain functions by directly accessing objects and using the services they provide. (*Dyalog Help:*) The OLEServer object allows you to export an APL namespace so that its functions and variables become directly accessible to an OLE Automation client application such as Microsoft Visual Basic or Microsoft Excel.
OLE Client	the application interface or application which controls or calls upon the OLE Server.
ActiveX Control (or OLE Control)	a usually visible object which user interacts with; may be embedded in another application. (*Wikipedia:*) a Microsoft term that is used to denote reusable software components that are based on Microsoft Component Object Model (COM). ActiveX controls provide encapsulated reusable functionality to programs and they are typically but not always visual in nature. (*Microsoft:*) an ActiveX control is implemented as an in-process server (typically a small object) that can be used in any OLE container.
Object Oriented	numerous meanings, but generally: a modular approach, using reusable units (called object, **class**, control, etc.) with common design, including **properties**, **methods**, and **events**; (nouns, verbs, things that happen?) (*APL+Win:*) an instance of any class is an object. consider an everyday example: book
ADO - ActiveX Data Objects	MS specification for interfacing to databases, using ActiveX/COM structure and methodology
DAO - Data Access Objects	older MS specification for database access, similar to ADO; see: C:\Program Files\Common Files\Microsoft Shared\DAO\DAO35.hlp
ODBC - Open Data Base Connectivity	older MS specification for interfacing to databases via SQL queries

Dyalog APL - Terminology

OLEClient	- object that enables your APL application to interact with non-APL objects (eg. Excel) - note how the Dyalog OLEClient object is an OLE Server application
OLEServer	- created by converting an APL Namespace to an object, which is then referred to by any non-APL application (eg. Excel) - requires dyalog.dll to run - a Dyalog OLEServer object may be called by an OLE Client application
OCXClass	used to access non-APL-derived ActiveX controls (OCX being one of several names associated with this class of objects)
ActiveXControl	stand-alone object created by APL, but accessed from non-APL applications; requires dyalog.dll to run - based on a Dyalog Namespace object
ActiveXContainer	"read-only"; used to represent the application that is hosting an ActiveXControl object, and provides access to its ambient properties (eg. Font, colour) (see: OLEQueryInterface Method, etc.)
GUI Memory	use `'#'` or `'.'` (root object), and `⎕wc, ⎕wn, ⎕ws, ⎕wg` to identify/explore
GUID or CLSID	Globally Unique IDentifier or CLaSs IDentifier - Windows handle

Let's take a glimpse of some of these things. **'.'** is the "root object" in Dyalog APL

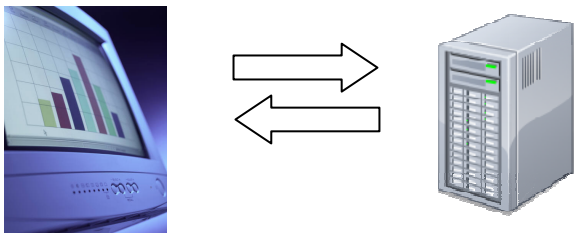
```
'.' ⎕wg 'PropList'                    ⍝ root level properties

⍴C←'.' ⎕wg 'OLEControls'              ⍝ Windows registry OLE Controls
↑C

⍴S←'.' ⎕wg 'OLEServers'               ⍝ Windows registry OLE Server objects
↑30↑S

(∨/'xcel'⍷¨↑S)⌿↑S                     ⍝ Excel objects?
(∨/'APL'⍷¨↑C)⌿↑C                      ⍝ other APL objects?
```

References

- Dyalog session Help > GUI Help
- Dyalog Interface Guide (download .pdf)
- Dyalog Release Notes / Help > Latest Enhancements (especially Version 11/12/13)
- Excel Help menu > see: "Table of Contents" > "Visual Basic Reference" > "Excel Object Model"
- Auto2000.chm - Microsoft Office 2000 automation Help file, available at:
  http://support.microsoft.com/default.aspx?scid=http://support.microsoft.com:80/support/kb/articles/q260/4/10.asp&NoWebContent=1
- MSDN Reference - Vast Microsoft Archive of developer info, eg.
  http://msdn.microsoft.com/en-us/library/microsoft.office.tools.excel(VS.80).aspx
- MFC - Microsoft Foundation Class Library - and other online resources,
  eg. http://msdn.microsoft.com/en-us/library/d06h2x6e(VS.80).aspx
- APL+Win workspaces and documentation

APL as Client, Server, etc.

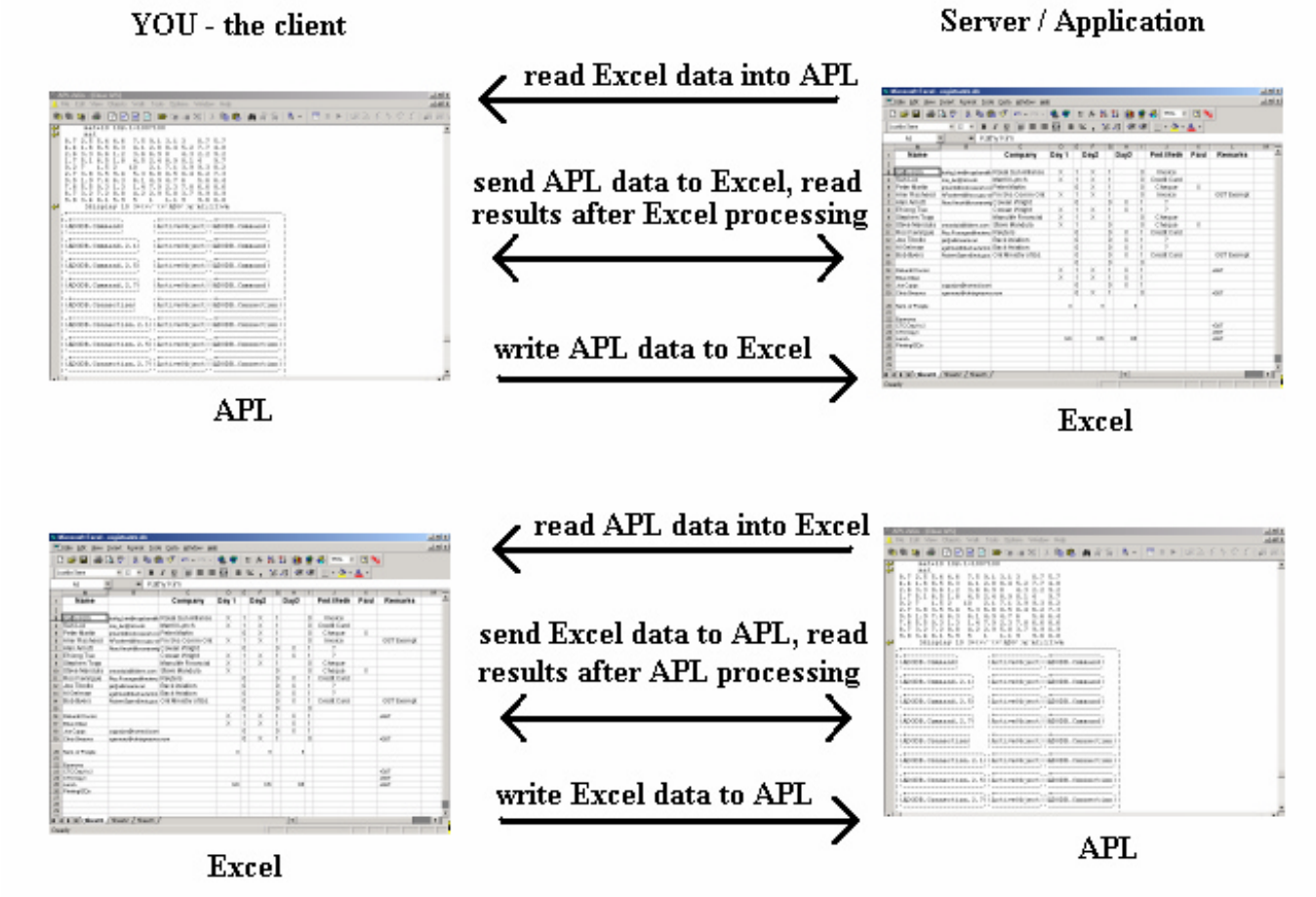
The "usual" client-server relationship involves multiple machines:



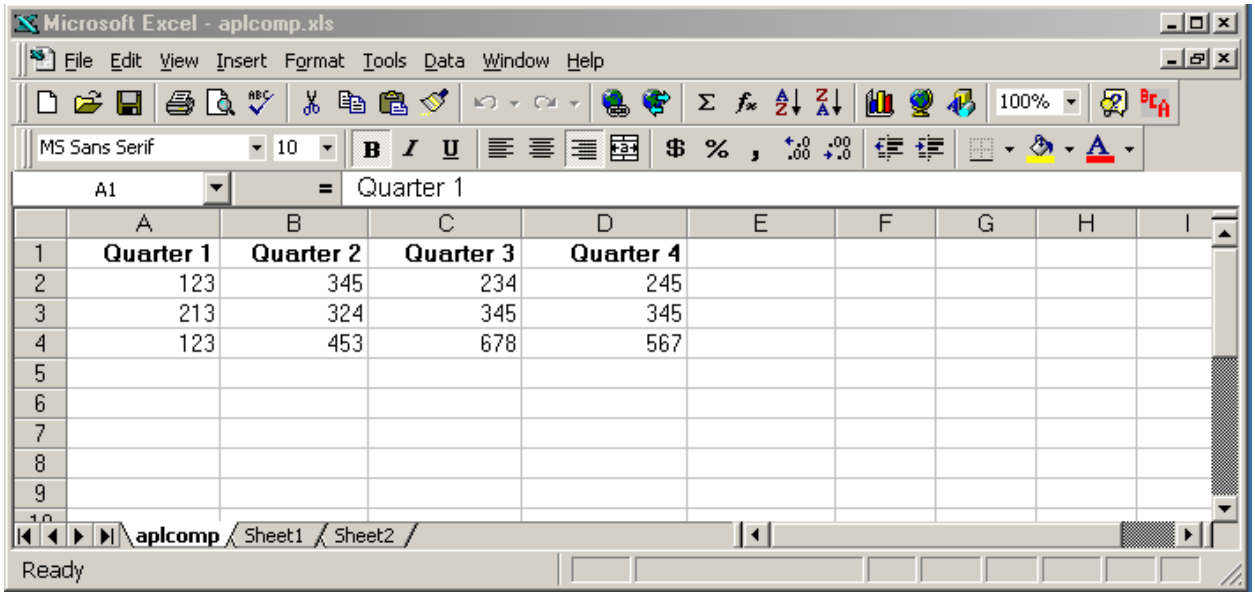
In our case, more likely the client and server are **concurrent tasks** on the same machine.

What do you really want to do?

APL as Client - in an APL development session, or an APL-GUI (runtime) application	- you **write** data (eg. to Excel), and perhaps go on to format and print or do further processing, eg. charts - you **read** data into your APL environment (eg. from Excel)
APL as Server - from within another Office application (eg. Excel)	- you call upon APL functions to perform calculations, read an APL-driven database etc. and typically return results to the Office application - or, you send data to an APL system and perform final processing there only
Other OLE Options	eg. create ActiveX controls using APL; use non-APL ActiveX objects from within APL; use ADO for database access; etc.



Excel - Components and Overview



Excel's Object-Oriented Structure

In very brief terms, the Excel spreadsheet is a hierarchical collection of **objects**; objects have **properties**; most have **methods**; and when we interact with these objects - **events** take place. This is the model we must work with to program such applications from any point of view, including APL.

Excel Help provides a good visual display of these objects, and an in-depth resource for understanding how to use these: for starters, try:

> "Contents and Index" > "Microsoft Excel Visual Basic Reference" > "Microsoft Excel Object Model"
> (it's Microsoft, so your version may differ!)

This diagram displays the hierarchical relationships between objects. Click on any of these to explore the details further down the branches of the tree to expand further levels. Notice the top-level object is called "Application".

Key Excel Components and Concepts

Obvious

Menu	Windows standard
Toolbars	one or more; contain tool buttons, combos, etc.; user may modify
Formula Bar	input area
Name Box	apply a name to a cell or range
Workbooks	**collection** object containing individual Workbooks, ie. opened *.xls files, each one being a Workbook object
Worksheets	(or Sheets) **collection** object; tabs at bottom-left; add, delete, move
Rows/Columns	A1:Z99 designation; some maximum values; note Rows numeric, Columns alpha; drag/drop heading areas to resize; Ctrl-down-arrow etc.
Cells	contain values or formulas; many properties to consider: datatype, format, font, behaviour, alignment (text left, num right), display precision, etc.

Not so obvious

Collection	special Excel object used to contain other objects, eg. Workbook > Worksheets; Item and Count properties are useful; special syntax needed in APL to use these
Item	Property; collection object unit; demo: select several cells, hit Enter key repeatedly
Count	Property; collection object unit count
Range	ambiguous term: a selection of cells (can you say "array" ?), either contiguous or non- (may even span Worksheets); note that a "Range Object" is created by using the Range property of a parent object (eg. Worksheet)
Select/Selection	- Method and Property; drag/drop to select one or more contiguous cells, or whole Rows/Columns/Worksheets; highlight to modify content or format in bulk - useful to programmatically change a block of cells - also see: Activate method; CurrentRegion; UsedRange, etc.
PrintOut	and other Methods; eg. print an object; Open, Close, Quit, Select, Activate, etc.
numerous sub-objects	various commonly-used objects and nomenclature such as UsedRange, ThisWorkbook, ActiveSheet, ActiveChart, ActiveCell, etc.
VBA/Macros; GUI elements	GUI objects and code stored as part of the spreadsheet; macro recording facility; built-in edit and debug environment

Now, let's go back to APL. First, we need to spend 5 minutes on:

APL GUI Programming - 101 (very lite)

In all APLs, object programming is accomplished with some handy ⎕-functions, now in conjunction with "dot syntax", ie. ParentObject.ChildObject.Property...

Dyalog	Examples

⎕WC = create
⎕WS = set
⎕WG = get
⎕WN = names
⎕DQ = wait
⎕NQ = invoke
	`'F' ⎕WC 'Form'`
	`'F' ⎕WG 'PropList'`
	`'F' ⎕WG 'Size' 'Posn'`
	`'F' ⎕WS 'Caption' 'Hello World'`
	`'F' ⎕WS 'BCol' 255 0 0`
	`'F.ED1' ⎕WC 'Edit'('Posn' 10 10)`
	`'F.B1' ⎕WC 'Button' 'OK'('Event' 'Select' '⍎⎕EX''F'''')`
	`⎕WN 'F'`
	`⎕DQ'F'`

and...	`F.Caption←'My Really Cool App'` `F.Size←20 40`

In similar fashion, we can take this methodology, and apply it to the Excel object, or other COM objects.

Exploring the Excel Object from APL

From APL, to use an OLE object as a server, we create an OLEClient object, which is implemented as a "namespace" in Dyalog APL, and we associate a particular server object with it, eg.:

```
⎕WX ← 3                    ⍝ or 1? , see: "External Object Behaviour" below

'XL' ⎕wc 'OLEClient' 'Excel.Application'
```

This creates an OLEClient object which will interact with an OLE Server, with the ClassName property as specified from our list of OLEServers, ie. from the Root Object property which examines the Windows Registry.

'XL' becomes a **namespace** object which we can explore or query and use. Dyalog provides system commands to enable us to explore a namespace's objects:

```
)CS XL                     ⍝ enter the "XL" namespace
)methods                   (and V12+: ⎕NL -3)
)events
)props                     ⍝ list properties
Version                    ⍝ because we have "exposed" the object properties...
LibraryPath
```

The above can also be accomplished via ⎕w_ syntax from the root level:

```
)CS                        ⍝ return to root level
'XL' ⎕wg 'MethodList'
'XL' ⎕wg 'Version'         ⍝ Excel version number (may be important)
'XL' ⎕wg 'Visible'         ⍝ is the server visible?
'XL' ⎕ws 'vIsIBLe' 1       ⍝ make it so; does case matter?
'XL' ⎕ws 'Visible' 0       ⍝ make it not
```

Dyalog also allows the use of direct object/property naming via "." syntax, eg.

```
XL.PropList
XL.Version
XL.Visible
XL.Visible←1
XL.visible←0               ⍝ suddenly, case matters! (and AutoComplete helps a lot)
XL.Workbooks.Count
XL.Rows.Count              ⍝ see Excel-Help re "Active Workbook"...
```

Interesting diversion...

```
XL.Speech.Speak ⊂'Are we having fun now? Or what?'
```
(vs. `XL.Speech.Speak 'Are we having fun now? Or what?'`)
```
XL.Speech.Speak ⊂'wut in hale yoo thank yore doone, bowah'
```

Also, Dyalog provides the "**Workspace Explorer**" tool to browse such objects.

Want to find out what any of these properties, methods or events **really means**?
For starters, try searching for the particular term in the Excel/VBA Help feature.

Key Points

- Dyalog implements GUI memory and OLE objects using the namespace concept, so:
    - use)OBS or ⎕NL to explore these
    -)CS or the ...object.sub-object... syntax allows us to explore, etc.
    -)ERASE or ⎕EX erases the object (but has it really gone away?)
    - assignment ← is used to set properties
    - ⎕NQ or object.sub-object syntax is used to invoke methods
    - even the each operator ¨ can be put to use for implicit looping, etc.

- upper/lower case-sensitivity varies when addressing properties, methods and events (according to the syntax used?)

- Dyalog AutoComplete - displays object Properties & Methods; useful to explore objects to some extent

- Incomplete or incorrect usage often returns a useful response, eg. Methods are indicated by APL del symbol ∇ when invoked but missing an argument, or in fact an "Exception Error" may appear in a separate "Status" form, eg. try XL.Workbooks.Open with an invalid path/filename. eg. "Range" objects are indicated by ...[Range] in the output.

- Visibility - is Excel visible in the task bar? Processes in Task Manager? etc.; set the Visible property

- Is Excel already running? can we connect to it? (try opening a spreadsheet, then create the OLEClient/Excel object in APL, then check XL.Workbooks.Count)
  also see: Dyalog GUI Help > InstanceMode property (of OLEClient)
  or see: alreadyrunningexceldiscussion.htm

- Which 'Excel.Application' server object name to use?
  see: "How to run multiple versions of Excel on the same computer"
  http://support.microsoft.com/kb/214388

- ⎕WX, main issues are: enclosed arguments and Item vs. [item] - with Version 11+
  see: Language Help > ⎕WX > External Object Behaviour

- 3 ⎕NQ invokes a method in an OLE Control. The (shy) result of ⎕NQ is the result produced by the method.

- Collection Objects - can be confusing, require special syntax, see below

- when things seem to be "hung"... maybe the Excel object is asking a question (via dialog box) - check it

Collection Objects

Consider these definitions from the Excel-Help:

Workbook Object - The Workbook object is a member of the Workbooks collection.

Workbooks Collection Object - A collection of all the Workbook objects that are currently open in the Microsoft Excel application.

Workbooks Property - Returns a Workbooks collection that represents all the open workbooks. Read-only.

Worksheet Object - Represents a worksheet. The Worksheet object is a member of the Worksheets collection. The Worksheets collection contains all the Worksheet objects in a workbook.

Worksheets Collection Object - A collection of all the Worksheet objects in the specified or active workbook. Each Worksheet object represents a worksheet.

See also: ActiveWorkbook; ThisWorkbook; Sheets; ActiveSheet; etc.

In other words, the many objects, levels and similar terms can be confusing. The main points are that collection objects have a special purpose and syntax, and we may refer to the objects they contain by using the **Item** property or equivalent reference via indexing.

Getting Started...
(First, look for this file or equivalent: 'c:\Program Files\Microsoft Office\Office\Library\common.xls', then:)

```
'XL' ⎕wc 'OLEClient' 'Excel.Application'
XL.Workbooks.Count
XL.Workbooks.Open ⊂'c:\...\common.xls'        (NB: v.11+, ⎕WX←3)
XL.Workbooks.Count
XL.Workbooks.PropList
(XL.Workbooks) ⎕WG 'PropList'
XL.Workbooks.⎕WG 'PropList'
WBS ← XL.Workbooks
WBS.PropList
XL.Workbooks.Item[1].PropList (or (XL.Workbooks.Item 1).PropList  in v.10)
XL.Workbooks.Item[1].Name
XL.Workbooks[1].Name                  ie. collection indexing
WB1 ← XL.Workbooks[1]
WB1.Name
WB1.Sheets.Count
```

Collection objects are typically a part of the hierarchy or path to get to the underlying information in the spreadsheet. The key Collection Objects include: Workbooks, Sheets (Worksheets), Rows, Columns, Range.

NB: Use of indexing via [*N*] , (or ...Item *N*) , to select a Collection Object Item depends on Dyalog version and ⎕WX setting. (see "External Object (COM and .Net) Behaviour" - Version 11 Release Notes)

APL in Control

First, look for this file or equivalent: 'c:\Program Files\Microsoft Office\Office\Library\common.xls', then find a second .xls file on your system to open, then work through these examples:

```
'XL' ⎕WC 'OLEClient' 'Excel.Application'
XL.Workbooks.Open ⊂ 'c:\... \common.xls'
XL.Workbooks.Open ⊂ 'c:\... \any_other_spreadsheet.xls'
XL.Workbooks.Count
XL.Workbooks[1].Name
WB1←XL.Workbooks[1]  ⋄  WB2←XL.Workbooks[2]
WB1.Name  ⋄  WB2.Name
WB1 WB2 ⎕WG¨⊂'Name'  ⋄  (WB1 WB2).Name
)COPY "C:\PROGRAM FILES\DYALOG\DYALOG APL 12.1 UNICODE\WS\DISPLAY"
DISPLAY WB1.Sheets[⍳WB1.Sheets.Count].Name
WB1.Sheets[1].PropList
SHT1←WB1.Sheets[1]
SHT1.Rows
SHT1.Rows.Count  ⋄  SHT1.Columns.Count
SHT1.UsedRange                    (and set  XL.Visible←1)
SHT1.UsedRange.Rows.Count
SHT1.UsedRange.Cells.Count
SHT1.Range
SHT1.Range[⊂'A1:E5']
SHT1.Range[⊂'A1:E5'].PropList
SHT1.Range[⊂'A1:E5'].Columns.Count
```

...and finally, we READ the worksheet values:

```
SHT1.Range[⊂'A1:E5'].Value2
RNG1← SHT1.Range[⊂'A1:E5']
RNG1.Value2
SHT1.UsedRange.Value2
SHT1.UsedRange.Rows[4].Value2          (or... .Columns[4]. )
M←SHT1.UsedRange.Value2
M ≡¨ ⎕NULL
```

...now let's WRITE to Excel:

```
SHT1.Range[⊂'A1:E5'].Value2←5 5⍴⍳25
SHT1.Range[⊂'A1:E5'].Value2←⍳25          (vs. reshape... above?)
SHT1.Range[⊂'A1:E5'].Value2←99           ⍝ scalar extension!
```

Using Excel Methods

eg. to Create a new Workbook (spreadsheet) and Write to it, Save it, etc. (the basics):

```
XL.Workbooks.MethodList
WB3←XL.Workbooks.Add 1                          (note: returns an object as result)
WB3.Sheets.Count
WB3.Sheets[1].Range[⊂'A1:E5'].Value2←5 5⍴⍳25
WB3.Path
WB3.SaveAs ⊂'APL2XLdemo1.xls'
WB3.Path                                        (vs. XL.Path ?)
WB3.Close 0          (vs. XWB3.Close 1  ? ie. save changes or not?)
```

see: MSDN > Workbook.Close Method:
http://msdn.microsoft.com/en-us/library/microsoft.office.tools.excel.workbook.close(VS.80).aspx

so don't forget to : `WB1.Close 0 ...`

also useful : `...Sheet.UsedRange.Columns.AutoFit`

`...Sheet.UsedRange.ClearContents`

back to writing APL>Excel for a moment:

try writing an enclosed array, eg.

```
MAT←2 3⍴'Date' 'Account' 'Amount' 40819 'payments' 23.34
SHT1.Range[⊂'A1:E5'].Value2←MAT
```

Datatypes, Formatting, etc.

COM objects such as Excel typically have data represented by more datatypes than those available in APL. Excel datatypes for example include: Boolean, Date/Time, Double, Error, Integer, Long, String, Currency, Variant.

When transferring data back and forth between APL and Excel therefore, we may need to pay attention to the datatype of our data to make sure it is both stored and represented (displayed) correctly (especially Dates).

- APL provides `⎕null` (displayed as `[Null]`) which is used to represent null values which COM methods often return. Other data conversions for data transferred between systems are automatic.

- Value vs. Value2? (*from Excel-Help*)

"The only difference between the Value2 property and the Value property is that the Value2 property doesn't use the Currency and Date data types. Depending on how a cell is formatted (for example, with date, currency, or other formats), the two properties may return different values for the same cell."

More info - see: *http://support.microsoft.com/kb/213719 and http://support.microsoft.com/kb/182812 and http://blogs.msdn.com/b/eric_carter/archive/2004/09/06/225989.aspx*

In general, for Dyalog v11+ - the Value property doesn't seem to work (?).

- IS Functions (*from Excel-Help*) - use these to determine particular characteristics of cells, eg.

    XL.ISNUMBER XL.Workbooks[1].Sheets[1].UsedRange

This returns a 2-cell result with a boolean array in cell-1, indicating cells with numeric values (presumably of any of the numeric datatypes?). (see also: ISNA, ISBLANK, ISLOGICAL...) Note that not all of the IS_ functions are available - see XL.MethodList - ie. some Methods are "exposed", others are not (?); and the list seems to grow as these are used (??).

- Formula vs. Value2? Enter some formulae on a spreadsheet, then compare ...Range.Value2 vs. ...Range.Formula (and check the shape of each cell...)

- APL+Win offers root-object-level query & set functionality for data type and value (see the VT_VV workspace).

- Under-filling cells results in "#N/A", eg.

    ```
    XL.Workbooks[1].Sheets[1].Range[⊂'D15:F17'].Value2←2 2⍴⍳4
    ```

    (and seems to cause DOMAIN ERROR when reading back??)

Dates

Dates are stored in Excel as a day-count number (days since 1900-01-01) but typically represented (displayed) in other ways, such as '13-Sep-09'. Make sure your date information in Excel is numeric and not a character string that looks like a date. There are several display options, see Format > Cells > Number > Date in the Menu, or In in *Excel-Help*, see: "Available Number Formats" > "Display numbers as dates or times" > "Custom date and time codes".

When calculating day-count values, note that Excel incorrectly counts 1900 as a leap year, hence dates are offset by 1 between APL and Excel (see DateToIDN Method in Dyalog GUI Help > IDN definition).

Some examples: (open a spreadsheet... enter some dates, numerics and text if not already present...)

```
WB1←XL.Workbooks[1]
RNG1←WB1.Sheets[1].Range[⊂'A1:A7']      ⍝ choose any appropriate range with dates
RNG1.NumberFormat
RNG1.NumberFormat←'yyyy/mm/dd'          ⍝ or other date formats, view the result in Excel
RNG1.NumberFormat←'###.####'
RNG1.NumberFormat←'$###.00'

)COPY C:\Dyalog_Folder...\WS\UTIL SM_TS TS_SM DISP
TODAY←SM_TS 3↑⎕TS
WB2←XL.Workbooks.Add 1          ⍝ create a new Workbook, write some numbers and dates
RNG2←WB2.Sheets[1].Range[⊂'B3:F7']
RNG2.Value2 ← (TODAY+0,⍳4),5 4⍴ 0.001×100?10000
WB2.Sheets[1].Range[⊂'A1:A5'].NumberFormat' ← 'dd-mmm-yy'
```

(...and a bit of formatting, etc.)

```
RNG2.Font.Size ← 14  ⋄  RNG.Font.Italic ← 1
RNG2.Interior.Color ← 256 ⊥ 0 0 255
RNG2.Interior.ColorIndex ← 44           ⍝ some pre-set colours
```

(and by the way, try: `RNG2.Interior.Colorindex ← 44` - no error)

```
RNG2.ClearFormats
RNG2.ClearContents
```

Utilities

Rather than invent most of the wheels...

First, find or create an Excel spreadsheet that has more than one worksheet, and some data (numeric or char or a mixture) in a few cells. Then:

```
)LOAD ...\samples\ole\oleauto          (in the Dyalog install folder)
DESCRIBE
```

Reading a spreadsheet

```
ρmat←XLCONTENTS 'C:\...common.xls'
ρ¨mat                                  (etc., explore the structure of the result)
```

Examine XLCONTENTS, (or XLCONTENTS1) and note how control structures are used to navigate through Excel's object hierarchy and collection objects.

Writing To a Spreadsheet

```
)ED XLPRINT      (modify it to not PrintOut, Close or Quit, and remove EX from the header...
                 change name to XLDISPLAY)
mat2←10 10ρ0.1×100?1000
1 XLDISPLAY mat2                       (note EX object, EX.Visible, etc.)
```

From here, you can modify & print the Excel sheet, close it, close Excel,)erase EX

or from APL, you could do these actions separately under program control, as in:

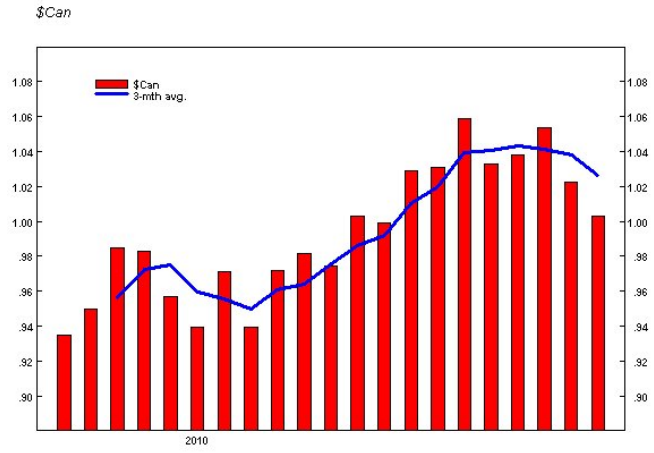
```
EX.Workbook[1].Close 1
EX.Quit ⋄ ⎕EX 'EX'
```

Excel Charts?

```
)ED XLCHART
```

AND... *new in 2011*: The David Crossley Excel Toolkit

...used recently to facilitate creation and delivery of detailed timeseries charts



Excel Toolkit - APL drives Excel

```
∇ XLDEMO1
 DATED 12 2010 1 2011 12
 GET 'CA.INT.XR'
 TITLE '$Can'
 LABEL '1¨$Can◇2¨3-mth avg.'
 2 LINE'SOLID,THICK'
 2 LINECOLOUR'BLUE'
 BARCOLOUR'RED'
 BARLINE 1
 BARWIDTH 0.15
⍝ PLOT'XR,3 MOVAVG XR'
 'bar,line' XLCHART 'XR,3 MOVAVG XR'('colours' 'red,blue')
```

How does it work?

- usual steps up to PLOT command
- format timeframe dates for X-axis text
- generate "dummy" 2nd Y-axis values; Y-axis min/max, etc. (override Excel behaviour)
- gather X-axis text, dates, timeseries data in one array
- open Excel, place data in initial columns
- create chart object with spreadsheet data
- insert existing workspace items (title, legend, etc.)
- apply "Company Standard" settings (size, margins, fonts, line styles, colours, object positions, etc.)

Observations

- coding is complicated by different chart types (pie, bar, line...)
- Excel VBA web examples - can be highly useful for direct use in Excel, or translated to APL; plenty of "how to..." code (VBA) can be found on the web and implemented (after much trial & error)
- APL & Excel can be tightly integrated for specific applications, even legacy APL
- support for different operating systems and different versions of Excel can lead to complexity
- Excel chart object - complex; difficult documentation; non-trivial to customize
- APL drives Excel, or APL serves Excel - moving code from one to the other is relatively painless
- APL drives Excel = better data handling in APL; Excel as display mechanism
- APL serves Excel = more scope for adopting VBA web examples

Excel Toolkit - APL drives Excel

```
   ∇ {type}XLCHART DATA;V;ARG;...
⍝ DATA = 1 or more timeseries, same arg as PLOT...
⍝ type = optional chart type(s) ie. one or more of: line bar...
...
:If ∨/~type∈ΔXLCHARTTYPES[;1]
     'Invalid chart type. Choose one of: ',DEB⍕,' ',⍕ΔXLCHARTTYPES[;1] ...
:If 0=1 XLFNS.startexcel''    ⍝  already up and running?
...
     XLFNS.open''
...
     0 0⍴XLFNS.xl.ActiveSheet.UsedRange.Clear ⍬
...
res←type XLFNS.tsplot'data'DATA'title'TL'plotposn'((55×1↓⍴DATA),15)...
```

```
      ΔXLCHARTTYPES                    ↑↑↑ (property value) pairs...
line              4
bar              51
3dbar         ¯4100
hbar             57
pie               5
scatter       ¯4169
linemarkers      65
```

```
c←{X}tsplot Y;parms;all_parms;defined;dataposn...;⎕WX
⍝ timeseries version - Y = 2 or more numeric cols; col-1=dates (in EXCELBASE form)
⍝Y: name-value pairs OR numeric data array
⍝X: chart type: number or character string (default = line)
 ⎕WX←1
 all_parms←'data' 'dataposn' 'title' 'xtitle' 'ytitle' 'legend' 'labels' 'labelposn'
'datarowscols' 'legposn' 'plotposn' 'null' 'yrange' 'scaling' 'colours' 'export'
...
⍝ ACME Company Standard look
 c.(Axes 2).HasMajorGridlines←0
 n←¯1+1↓⍴data
 (c.SeriesCollection n).Fill.Visible←0
 (c.SeriesCollection n).Border.LineStyle←-4142
 :If type≠-4100 ⋄ (c.SeriesCollection n).AxisGroup←2 ⋄ :EndIf ⍝ not 3dbar
 (c.Legend.LegendEntries n).Delete
 c.PlotArea.Border.ColorIndex←1
 (c.Axes 2 1).Border.ColorIndex←1
 :If type≠-4100 ⋄ (c.Axes 2 2).Border.ColorIndex←1 ⋄ :EndIf
 (c.SeriesCollection n).MarkerStyle←-4142
 c.(Axes 1 1).MajorTickMark←2  ⍝ tik marks inside
 :If type≠-4100 ⋄ c.(Axes 2 2).MajorTickMark←2 ⋄ :EndIf
 (c.Axes 1).TickLabels.Font.Size←F2←'Fonts.SF2'ObjData'PointSize'
 :If type≠-4100 ⋄ (c.Axes 2 2).TickLabels.Font.Size←F2 ⋄ :EndIf
 (c.Axes¨(1 1,type≠-4100)/(1 1)(2 1)(2 2)).TickLabels.Font.Name←c#.Fonts.SF2.PName
```

A Few More Tips & Tricks

ActiveSheet & UsedRange

```
XL.Workbooks[1].Sheets[1].Activate      ⍝ "activate" a worksheet, then use "ActiveSheet"
XL.ActiveSheet.UsedRange.Count          ⍝ count of what?
XL.ActiveSheet.UsedRange.Address ''     ⍝ ?
XL.ActiveSheet.UsedRange.Value2         ⍝ get it all - what about multiple Worksheets?
XL.ActiveSheet.Range???                 ⍝ select a more specific range?
```

Note: generally - UsedRange is tricky, ie. affected by what the user does with the spreadsheet, so exact ranges are more reliable.

Result of Single vs. Multiple Cell Values

```
⍴XL.Workbooks[1].Sheets[1].Range[⊂'C99'].Value2 ⍝ depends on cell content (empty?)
⍴XL.Workbooks[1].Sheets[1].Range[⊂'A1:C3'].Value2
DISPLAY XL.Workbooks[1].Sheets[1].Range['A1:C3' 'D1:F4'].Value2
```

Named Ranges (for template-driven output?)

```
XL.ActiveSheet.Range[⊂'E5:H9'].Name ← 'MyRng1'
XL.ActiveSheet.Range[⊂'MyRng1'].Value2 ← mat3      ⍝ populate known range(s)
```

...create some named ranges in Excel directly, then:

```
XL.Workbooks[1].Names.Count
(XL.Workbooks[1].Names.Item 1).Name          (or... Names[1]... or... .Value ?)
(XL.Workbooks[1].Names.Item¨⍳XL.Workbooks[1].Names.Count).Name
rngs←(XL.Workbooks[1].Names.Item¨⍳XL.Workbooks[1].Names.Count).Name
DISP(XL.Workbooks[1].Sheets[1].Range[rngs]).Value2
```

Multi-Area Ranges

```
XL.ActiveSheet.Range[⊂'C7:D9,F7:G9'].Name←'myrng2'
XL.ActiveSheet.Range[⊂'myrng2'].Value2←(3 2⍴⍳6)(3 2⍴100+⍳6)   ⍝ any result?
XL.ActiveSheet.Range[⊂'myrng2'].Value2←(3 2⍴⍳6),(3 2⍴100+⍳6)
XL.ActiveSheet.Range[⊂'myrng2'].Areas.Count
XL.ActiveSheet.Range[⊂'myrng2'].Areas[1].Value2
XL.ActiveSheet.Range[⊂'myrng2'].Areas[1 2].Value2←(3 2⍴⍳6)(3 2⍴100+⍳6)
XL.ActiveSheet.Range[⊂'myrng2'].Value2←99          ⍝ scalar extension
```

Conclusion: passing data to multi-area ranges is tricky, note the simple vs. enclosed argument usage

A Few Other Nifty Things... *OLE!*

We can launch other OLE applications and/or create documents from APL.

Word

In similar fashion to Excel, we link APL to the OLE Server object for MS-Word, then set or query its various sub-objects, etc.

- study the VB code in some of the following: (ie. Google "Word OLE Automation")
- http://vbcity.com/forums/topic.asp?tid=34572
- http://support.microsoft.com/kb/237337
- http://support.microsoft.com/?kbid=250501
- emulate this code in APL and create an APL driver function to send arbitrary text to a Word document

PowerPoint

- find the PowerPoint OLEServer object
- create an OLEClient object which uses the PowerPoint server
- explore the server object properties & methods
- create an APL driver function which will open and run a PowerPoint .pps file

OCX Demo

OCX Controls behave similarly to the OLE Server objects above, except they may need to be associated with a Form or other container to be useful. These objects are designed to take on some of the properties of their environment (container).

See: "Loading an ActiveX Control", in the Dyalog Windows Interface Guide (Chapter 11).

- use the 'Microsoft Forms 2.0 TextBox' OLE Control object by building an APL driver function to contain and present this object

PDF

Similar to the OCX control above, Adobe Acrobat provides an ActiveX control which can be used to display a PDF document.

- find the Adobe Acrobat ActiveX component needed to display PDF documents
- create an APL driver function which will display a given PDF document

Outlook

MS-Outlook is an OLE-compliant application. APL can be used to send a message, for example.

```
     ∇ OUTLOOKSENDMSG ARG;OL;OLNS;OLM;⎕ML;⎕IO
[1]   ⍝ send an email message via Outlook
[2]   ⍝ ARG = 3-strings: 'To' 'Subject' 'Body'
[3]   ⍝  eg. OUTLOOKSENDMSG 'everybody' 'Drinks' 'Drinks are on me tonight...'
[4]   ⍝  Body may be a CR-delimited string
[5]
[6]    ⎕IO←1 ⋄ ⎕ML←0
[7]    'OL'⎕WC'OLEClient' 'Outlook.Application'
[8]    OLNS←OL.GetNamespace'MAPI'
[9]    OLNS.Logon''
[10]   OLM←OL.CreateItem'olMailItem'
[11]   OLM.To←1⊃ARG
[12]   OLM.Subject←2⊃ARG
[13]   OLM.Body←3⊃ARG
[14]   OLM.Send
[15]   OLNS.Logoff
[16]   'Message Sent, to: ',,⍕1⊃ARG
[17]   OL.Quit
     ∇
```

What Else Is New?

1. **⎕WX** changes - see Session Help "Latest Enhancements": "Expose Windows Object Properties" and "External Object Behaviour"

2. **⎕NL** and COM Objects - see: Session Help "Latest Enhancements", **⎕NL**, **⎕NC** and "name-class" definitions

Extensions for more detailed object information.

3. APL Classes based on OLEClient - see:

240 Dyalog APL/W Interface Guide

Writing Classes based on OLEClient

You may define APL Classes (See Language Reference) based upon the OLEClient object. For example:

```
:Class Excel: 'OLEClient'
    ∇ ctor wkbk
      :Access Public
      :Implements Constructor :Base ,⊂('ClassName' 'Excel.A
pplication')
      Workbooks.Open ⊂wkbk
    ∇
:EndClass ⍝ Excel


      XL←⎕NEW Excel 'f:\help11.0\days.xls'
      XL.Workbooks[1].Sheets[1].UsedRange.Value2
   From       To     Days  Hours
  38790    38791        0   3.25
  38792    38792   [Null]   2.25
  38793    38793   [Null]   2.5
  38799    38799   [Null]   5
  38800    38800   [Null]   3
 [Null]   [Null]   [Null]  16
```

Excel in Control

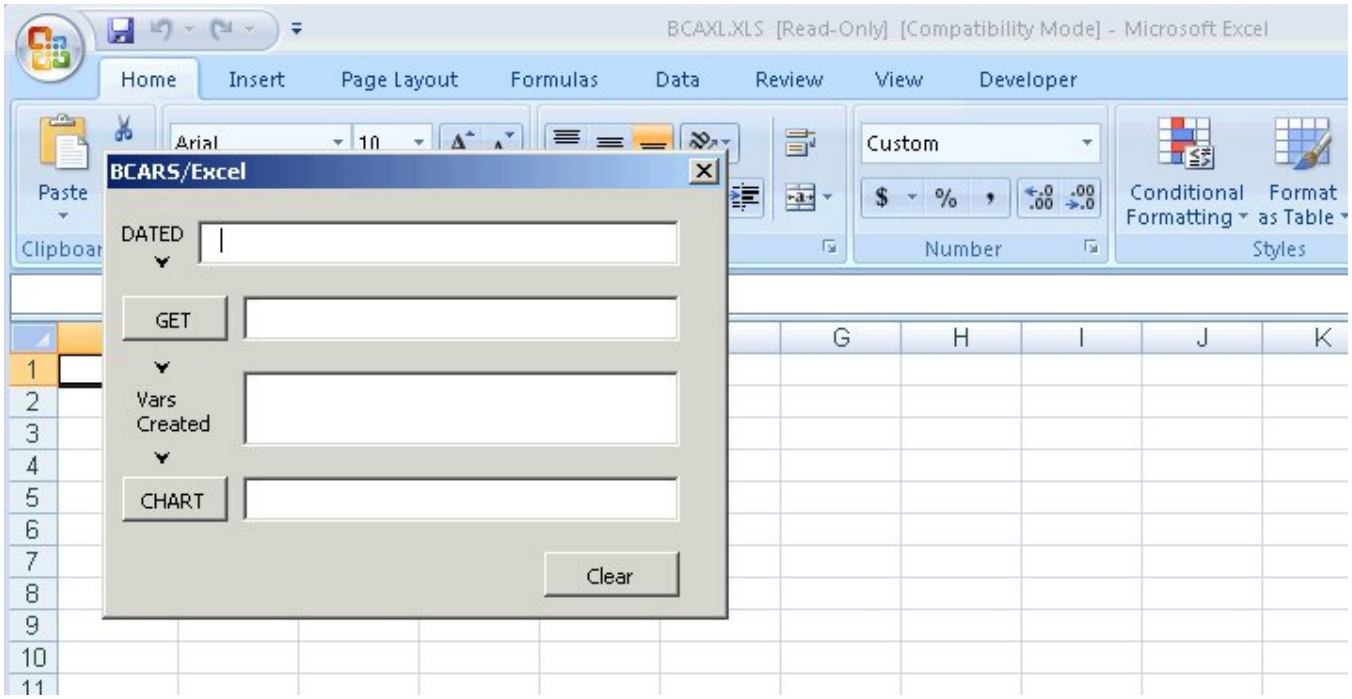
Here, we focus on designing the user interface in Excel/VBA. APL may be "plugged into" this application as a black box, hidden from view normally. APL functions are passed arguments, and return results as with any other Excel/VBA function. Excel is the client, APL the server (often called "calculation server or engine").

In General

Calling APL functions from within your Excel application involves the following steps:

- create appropriate APL code in a workspace, with some special considerations; test and debug
- create/register the required .dll using this code
- create the Excel spreadsheet, including VBA code which links into APL code
- test and debug

In Dyalog APL, we use a Namespace object to contain the code which Excel will call upon, and Dyalog provides a special facility to turn that code into a .dll file.



Dyalog APL provides a built-in facility in the session manager for creating the required .dll object which will deliver APL capability to your Excel application. This is documented in the Dyalog APL "Interface Guide" (Chapter 12: "OLE Automation Server").

Worth reading in that chapter:

- Rules for Exported Functions
- Out-of-Process and In-Process OLE Servers

Key Steps

- Create an APL workspace containing a namespace, into which all the relevant code and other objects are placed
- make the namespace an OLEServer object, eg. `Loan.⎕WC 'OLEServer'`
- `)SAVE` the workspace (use a new name if starting with one of the demos)
- Choose File-Export from the menu, and select "In Process Server (*.dll)" or "Out of Process COM Server" (typically "In Process" - see Dyalog's Interface Guide)

Dyalog APL will automatically package up the namespace contents, "export" those items, and register the supplied Namespace as a .dll object (and produce a small log/report). This .dll is the object we then refer to in our Excel/VBA code, using the CreateObject function.

On the VBA/Excel Side

- Create your spreadsheet - typically this will be designed with input areas for data parameters to be passed to APL, and one or more "controls" (buttons, input boxes, etc.)
- Enter "Design Mode", using the toolbar brought into view by: View > Toolbars > Control Toolbox
- Right-click on controls to choose "View Code" or "Properties"; or enter the VBA Macro editor to modify these

Let's look at the supplied example: ...\samples\ole\loan.xls

```
Private Sub CBGet_Click()                (or use the "Assign Macro" feature)
  Dim APLWS As Object
  Set APLWS = CreateObject("dyalog.NAMESPACENAME")
            ...

Sub Calc()
   Dim APLLoan As Object
   Dim Payments As Variant
   Set APLLoan = CreateObject("dyalog.Loan")
   LoanAmt = Cells(1, 3).Value
   LenMax = Cells(2, 3).Value
   LenMin = Cells(3, 3).Value
   IntrMax = Cells(4, 3).Value
   IntrMin = Cells(5, 3).Value
   APLLoan.PeriodType = 1
   Payments = APLLoan.CalcPayments(LoanAmt, LenMax, LenMin, IntrMax, IntrMin)
   For r = 0 To UBound(Payments, 1)
      For c = 0 To UBound(Payments, 2)
         Cells(r + 1, c + 5).Value = Payments(r, c)
      Next c
   Next r
End Sub

Sub CalcPayments()                              (alternate coding to above)
...
```

The Tricks

For Dyalog APL, the following rules and caveats apply ***at creation time***, when creating the .dll through the above process:

- all ("called"?) functions must be niladic or monadic; no dyadic functions, nor dynamic functions, derived functions or operators; this could lead to some code changes for some existing systems
- any global variables cannot be enclosed arrays
- results (returned to Excel) may be simple or enclosed
- reporting exceptions, errors, etc. back to Excel can be a chore, depending on desired level of detail

from the *Interface Guide:*

Rule 1: Exported APL functions must be niladic or monadic *defined* functions; dyadic functions, dynamic functions, derived functions and operators are not allowed. However, ambivalent functions may be called (monadically) by OLE.
Rule 2: Character arrays whose rank is greater than 1 are passed as 1-byte integer arrays. This means that 1-byte integer matrices and higher-order arrays will always be converted to character arrays.
Rule 3: An exported APL function may not be called with an empty numeric vector (zilde) as its single argument. Zilde is used by an APL client to call a non-niladic OLE method with no arguments.
Rule 4: If an exported APL function is called with more than one parameter, its argument will be a nested vector. If it is called with a single parameter that is a character vector or an array whose rank is greater than 1, the argument supplied to the APL function will be a simple array. Effectively, a 1-element nested array received from an OLE Client is disclosed.

Valence

Make dyadic functions monadic, by combining the arguments into one:

```
 R← CAT1 arg                                  vs.   R← arg1 CAT1 arg2
[1] ⍝ catenate 2 arrays on the 1st axis
[2]  ...
```

Then, change the syntactic use of each function which has changed, eg.

```
 C← CAT1 (A B)
```

Globals

Create "make" functions instead of globals, and invoke these in a "QUADLX" function which is called by VBA before calling any other code, or invoke them as needed at the beginning of other processes or functions, eg.

```
MAKE∆Frequencies
 Frequencies←6 2⍴'CUT/D/DAILY/W/WEEKLY/M/MONTHLY/Q/QUARTERLY/S/SEMI/Y/YEARLY'
 Frequencies←261 52 12 4 2 1,Frequencies
```

Error, Exceptions

Error handling can be as (un)sophisticated as you wish, but consider these suggestions:

- do as much input-checking "up front", ie. in Excel, before sending bad info to APL
- modify your APL code to always return a result, being an error message or valid result; ie. let Excel deal with the result according to content
- return an error code with all results from APL (0 = OK; 1 = not OK, etc.)

A Few Other Issues

1. A hierarchy of Namespaces in your server application?

Implementing an Object Hierarchy

Despite the close correspondence between the object model and Dyalog APL namespace technology, there is one significant difference. OLE does not support object hierarchies in the sense that one object *contains* or *owns* another.

Instead you must implement object hierarchies using *properties* that refer to other objects and/or *methods* that return objects as results.

It is not possible to pass Dyalog APL namespace hierarchies through OLE because OLE does not support them. If you want to write an OLE Automation Server in APL that implements an object hierarchy, you must follow the OLE conventions for doing so.

2. Running your server application on a network?

Configuring an *out-of-process* OLEServer for DCOM

Introduction

When you register an *out-of-process* OLEServer using *File/Export* or OLERegister, Dyalog APL automatically updates the Windows registry so that your OLEServer is immediately accessible to an OLE client application running on *the same* computer.

If you wish to make the same object accessible to client applications running on *different* computers (using distributed COM, or DCOM) you have to install additional registry entries on the server and on each of the clients.

3. Calling your server asynchronously, see the OLEASYNC workspace, and:

Calling an OLE Function Asynchronously

Introduction

Functions exported by an OLEServer are executed (by the underlying OLE technology) in a *synchronous* manner. This means that the OLE client must wait for the function to complete before it can continue processing.

In certain cases the client may not be interested in a result from a function and it may be desirable for client not to have to wait. For example, if a function updates files or performs a printing task, it would be nice for the client application to continue while the server performs this task in background, or indeed (using DCOM) on another computer.

4. Writing ActiveX Controls in Dyalog APL - see Interface Guide, Chapter 13

What is an ActiveX Control ?

An ActiveX Control is a dynamic link library that represents a particular type of COM object. When an ActiveX Control is loaded by a host application, it runs *in-process*, i.e. it is part of the host application's address space. Furthermore, an ActiveX Control typically has a *window* associated with it, which normally appears on the screen and has a user interface.

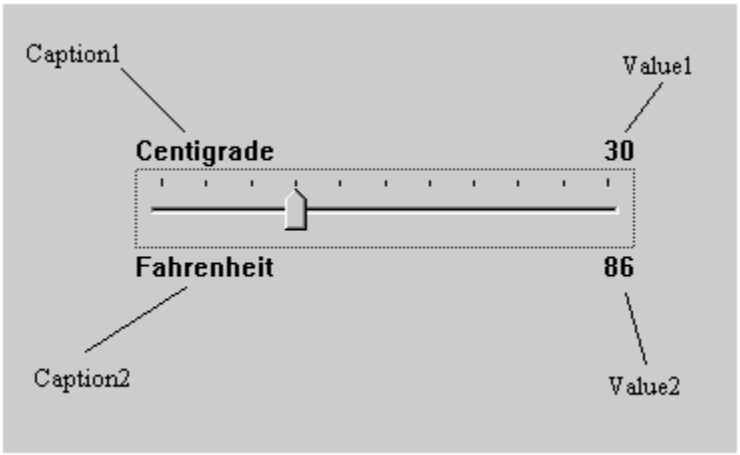
An ActiveX Control is usually stored in file with the extension .OCX. The functionality provided by the control can be supplied entirely by functions in that file alone, or can be provided by other dynamic link libraries that it loads and calls, i.e. an ActiveX Control can be stand-alone or can rely on one or more other dynamic link libraries.

What is a Dyalog APL ActiveX Control ?

An ActiveX Control written using Dyalog APL is also a file with a .OCX extension. The file combines a small dynamic link library *stub* and a workspace. The functionality of the control is provided by the functions and variables in the workspace combined with a dynamic link library version of Dyalog APL named `DYALOG101.DLL` or `DYALOG101RT.DLL` that is normally installed in the Windows System directory.

Note that an ActiveX Control written in Dyalog APL is a GUI object that has a visible appearance and a user interface.

To write an ActiveX Control in Dyalog APL, you use `⎕WC` to create an ActiveXControl object, as a child of a Form. An ActiveXControl is a container object, akin to a Group or a SubForm, that may contain a whole range of other controls such as Edit, Combo, Button and Grid objects. You may populate your ActiveXControl with other objects at this stage and save them in the workspace. However, you may prefer to create these sub-objects when an instance of the ActiveXControl is created. This happens when your control is loaded by a host application.



External Object (COM and .Net) Behaviour

(from: Dyalog Version 11 Release Notes)

Version 11.0 improves the behaviour of COM and .Net objects, but for backwards compatibility it is possible to select old or new behaviour using ⎕WX.

Old behaviour:

a) Character vectors supplied as arguments to external functions, which are defined as String parameters, are automatically enclosed for you. Similarly, string results are automatically disclosed.

b) Properties that take parameters, such as the Item Property in a Collection, are treated as methods.

c) APL provides lists of the Properties, Methods and Events provided by a GUI object by exposing additional properties named PropList, MethodList and EventList.

New behaviour:

a) Character vectors supplied as arguments to external functions, which are defined as String parameters, must be enclosed. Strings are returned as enclosed character vectors.

b) Properties that take indices, such as the Item Property in a Collection, are honoured as Numbered or Keyed Properties and may be accessed by indexing.

c) PropList, MethodList and EventList are not exposed. Instead, the information is provided by ⎕NL ¯2, ¯3 and ¯8 (but alphabetically sorted).

The actual behaviour of a COM or .Net object is now determined by its value of ⎕WX. If ⎕WX is 0 or 1, the old behaviour will apply. If ⎕WX is 3, the new behaviour will apply.

The behaviour of COM and .Net objects in existing applications will remain the same (because ⎕WX will be 0 or 1) but you may obtain the benefits of the new behaviour by setting ⎕WX to 3 at the appropriate level in your application. Then, everything below that (in the namespace hierarchy) will adopt the new behaviour.

Note that regardless of the value of ⎕WX, Version 11 will honour the Default Property of an external object thereby permitting the direct use of indexing on the object itself.

For example, if **xl** is an instance of the Excel.Application COM class, the following expression to obtain the contents of the first Sheet in the first Workbook will succeed, whatever the value of ⎕WX.

```
xl.Workbooks[1].Sheets[1].UsedRange.Value2
```

Note that it is the value of ⎕WX which the object acquired when it was created, rather than the current value of ⎕WX, which decide the behaviour.

Like other system variables, ⎕WX is inherited from the environment when a new namespace, class or instance is created. Classes inherit the value of ⎕WX when a class is edited or fixed, unless the class script explicitly sets a value for ⎕WX. In the case of .NET classes, ⎕WX is inherited when the class or namespace is loaded from a .NET assembly. For built-in (GUI) classes, each new instance inherits ⎕WX when it is created.

Examples

```
      ⎕WX←1
      'XLW' ⎕WC 'OleClient' 'Excel.Application'
      XLW.Workbooks.Add ⍬
      XLW.ActiveWorkbook.Sheets.(Item 'Sheet2').Index
2
      ⎕WX←3
      XL←⎕WC 'OleClient' (⊂'ClassName' 'Excel.Application')
      XL.Workbooks.Add ⍬
      XL.ActiveWorkbook.Sheets[⊂'Sheet2'].Index
2
```

Note that it is the value of ⎕WX in the object, and not in the calling environment, that decides the behaviour:

```
      ⎕wx←3
      ⎕using←''
      System.DateTime.Parse⊂'2006-09-12'
12/09/2006 00:00:00
      ⎕wx←1
      System.DateTime.Parse'2006-09-12'
LENGTH ERROR
      System.DateTime.Parse'2006-09-12'
      ^
      System.DateTime.⎕WX←1
      System.DateTime.Parse'2006-09-12'
12/09/2006 00:00:00
```

Note that, if we expunged the System.DateTime class instead of setting ⎕WX to 1, and repeated the expression, a new DateTime class would be created but it would inherit ⎕WX from its parent (System), where ⎕WX still has the value 3. Using .NET classes in an application where ⎕WX varies within a single APL namespace can therefore lead to unexpected results. It is recommended that applications only use more than one value for ⎕WX as a temporary measure during a conversion project.

⎕WX - APL Session Help > Language Help

⎕WX is a system variable that determines:
a) whether or not the names of properties, methods and events provided by a Dyalog APL GUI object are exposed.
b) the behaviour of .Net objects that have been created by any means other than ⎕NEW or .New.

The permitted values of ⎕WX are 0, 1, or 3. Considered as a sum of bit flags, the first bit in ⎕WX specifies (a), and the second bit specifies (b).

If ⎕WX is 1 (1st bit is set), the names of properties, methods and events are exposed as reserved names in GUI namespaces and can be accessed directly by name. This means that the same names may not be used for global variables in GUI namespaces.

If ⎕WX is 0, these names are hidden and may only be accessed indirectly using ⎕WG and ⎕WS.

If ⎕WX is 3 (2nd bit is also set) COM and .Net objects adopt the Version 11 behaviour, as opposed to the behaviour in previous versions of Dyalog APL.

ADO and Dyalog APL

Using DAO (precursor to ADO) Dyalog allows us an easy-to-use facility for exploring SQL-based data sources:

```
)LOAD c:\...dyalog...\samples\ole\oleauto

TESTDB← 'c:\...\fpnwind.mdb'     ⍝ the MS-Access "NORTHWIND" sample DB

SEL←'Select * from Customers where Country = ''Canada'''

TESTDB SQL SEL

TESTDB LIST_FIELDS SEL
```

With time, one could dive into the SQL function code and create new options for more elaborate selections, update mechanisms, etc.

For databases other than MS-Access and the like, Dyalog offers the ODBC interface tools (Chapter 15 - Interface Guide) which provide a complete and detailed set of utilities for interacting with any of these data sources. This is non-trivial in scope.