



**ROBERTSON (Publishing)**  
15 Little Basing, Old Basing,  
Basingstoke, RG24 8AX, UK.

Copyright © Graeme Donald Robertson 2007-2008

*This publication may be used, reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the permission of the publisher.*

*This document is distributed subject to the condition that it shall not, by way of trade or otherwise, be sold or hired out without the publisher's prior consent. It may however be used in APL classes and circulated in any form of binding or cover with a similar condition, including this condition, being imposed on the subsequent owner.*

*First published January 2007 as APL3\_4\*.PDF.  
Second edition published December 2007 as APL3\_4\*.PDF.  
Third edition published May 2008 as APL3&4\*.PDF.*

ISBN 0 9524167 2 7

### **Dyalog APL is worth learning because it:**

- can be obtained for **£50** from [http://www.dyalog.com/private\\_registration.html](http://www.dyalog.com/private_registration.html)
- strictly complies with APL ISO **Standard 8485**
- has a brilliant **tracer** for the interpreted APL environment
- runs on **Windows** (9x, NT, 2000, CE, XP, Vista), AIX, Solaris, Linux and more...
- affords **multilingual** development and runtime environments
- has 71 built-in **GUI classes** (based on Windows API), each with 10 to 80 properties...
- is **.NET enabled** and Microsoft supported
- has freely available downloadable **manuals** and detailed **help** files (created using APL)
- has many useful supporting sample **workspaces** and files
- provides a *Dyalog Support Service* via [www.dyalog.com](http://www.dyalog.com) and [support@dyalog.com](mailto:support@dyalog.com)

The level 1 course in **APL 1 & 2** introduces mainstream 1<sup>st</sup> and 2<sup>nd</sup> generation APLs such as IBM APL2, Dyalog APL, STSC APL\*PLUS and MicroAPL APL.68000. **APL1&2.PDF** is freely available from MicroAPL at <http://www.microapl.co.uk/apl/APL1&2.PDF> and may be given by instruction.

This level 2 course in **APL 3 & 4** introduces Dyalog APL versions 7-11 under Microsoft XP Pro. The course, when delivered by Graeme Robertson Ltd., comes bundled with supporting materials:

- **varChar**, a Dyalog APL addin for studying Dyalog APL arrays, (a free stand-alone version is available from the [www.dyalog.com](http://www.dyalog.com) download zone)
- a handy, short, fold-up Dyalog APL **Reference Card**
- a sample Pocket Dyalog application workspace called **AddrBk.DWS**

**Conduct of this course:** The suggested conduct of the course is similar to that stated for **APL1&2.PDF** as reproduced below. The course notes are provided in Portable Document Format, **one APL3&4Module<sub>n</sub>.PDF file at a time**, to enable cut & paste, to encourage experimentation and remote learning, and to increase the likelihood of obtaining feedback ☺.

After short introductions, the group is invited to divide up into pairs. Each pair works on one computer for the duration of the course. Each pair is given the first module and asked to work through it on their computer at their own pace. Pairs are encouraged to help each other with new concepts and difficulties as they arise and to experiment on the computer with any ideas that they think they can express in APL statements. Tuition is given when the pair cannot resolve problems. Questions may be answered directly on matters of fact, or indirectly by way of a suggestion as to how the problem might be tackled. Each day could cover up to 10 modules, depending upon the pace and background of each pair. There is no pressure to complete all modules (remaining modules are given out at the end of the course). At the discretion of the tutor, modules may be skipped or assigned for private study after the course. Introductions and short synopses are given with an overhead projector at suitable intervals throughout the course to the group as a whole.

# APL 3 & APL 4

## Course Contents

<b>Module0: Notation and Conventions</b> .....	<b>9</b>
§ 0.1 New Symbols.....	9
§ 0.2 Naming Conventions.....	10
§ 0.3 Variable “ <i>dataTypes</i> ” .....	11
<b>Module1: Objects and their Properties</b> .....	<b>13</b>
§ 1.1 Object Spaces .....	13
§§ 1.1.1 Creating vanilla Namespaces with $\square NS$ .....	13
§§ 1.1.2 Creating GUI Object Spaces with $\square WC$ .....	13
§§ 1.1.3 Changing Space with $\square CS$ .....	14
§ 1.2 Properties of Object Spaces.....	14
§§ 1.2.1 Examining Properties of an Object with $\square WG$ .....	14
§§ 1.2.2 Setting Properties of an Object with $\square WS$ .....	14
§§ 1.2.3 Building complex Objects .....	15
§ 1.3 Property Variables .....	15
§§ 1.3.1 Exposing Object Properties with $\square WX$ .....	16
§§ 1.3.2 Assigning Properties with $\leftarrow$ .....	16
§§ 1.3.3 Rebuilding complex Objects .....	16
<b>Module2: Methods and Events</b> .....	<b>17</b>
§ 2.1 Object Methods .....	17
§§ 2.1.1 Enqueuing Object Methods with $\square NQ$ .....	17
§§ 2.1.2 Invoking Event default Action using $\square NQ$ .....	17
§§ 2.1.3 Method Functions.....	17
§ 2.2 Object Events and callback Functions.....	18
§§ 2.2.1 Firing Events by User Actions .....	18
§§ 2.2.2 Attaching callback Functions to Events .....	18
§§ 2.2.3 Bringing Objects to Life.....	19
§ 2.3 The Event Queue .....	19
§§ 2.3.1 Dequeuing Events with and without $\square DQ$ .....	19
§§ 2.3.2 Tracing $\square DQ$ .....	20
§§ 2.3.3 Defining complex Behaviour .....	21
<b>Module3: Dot Syntax</b> .....	<b>23</b>
§ 3.1 Object References.....	23
§§ 3.1.1 Making References with $\$$ and $\leftarrow$ .....	23
§§ 3.1.2 Parent.Child Hierarchy .....	24
§§ 3.1.3 Object.Object. .. Object.Object Rationale .....	25
§ 3.2 Direct Property Access .....	26
§§ 3.2.1 Object.Variable Syntax.....	26
§§ 3.2.2 Object.Object. .. Object.Property Rationale .....	27
§§ 3.2.3 Using Object.Object. .. Object.Property Constructions.....	27
§ 3.3 Direct Method Invocation.....	27
§§ 3.3.1 Object.Function Syntax .....	27
§§ 3.3.2 Object.Object. .. Object.Function Rationale.....	28
§§ 3.3.3 Defined Operators in Object Space .....	28

<b>Module4: The Session Object .....</b>	<b>30</b>
§ 4.1 Using the Session Object .....	30
§§ 4.1.1 Immediate Execution Mode of <code>□SE</code> .....	30
§§ 4.1.2 Tracer and Editor of <code>□SE</code> .....	30
§§ 4.1.3 Choosing syntax Colours .....	31
§ 4.2 Inside the Session Object .....	33
§§ 4.2.1 Exploring the Workspace and <code>□SE</code> .....	33
§§ 4.2.2 Examining Session Menus and Buttons .....	33
§§ 4.2.3 Miscellaneous Properties and Methods .....	34
§ 4.3 Building the Session Object .....	34
§§ 4.3.1 Tracing <code>▽BUILD_SESSION▽</code> .....	34
§§ 4.3.2 Foreign Language Support .....	35
§§ 4.3.3 Adding useful Extensions .....	35
<b>Module5: Control Structures .....</b>	<b>36</b>
§ 5.1 Logical Decisions and Jumps .....	36
§§ 5.1.1 The <code>:If</code> Statement .....	36
§§ 5.1.2 Further truth Conditionals .....	36
§§ 5.1.3 The <code>:select</code> Statement .....	37
§ 5.2 Looping Constructs .....	37
§§ 5.2.1 The <code>:For</code> Statement .....	37
§§ 5.2.2 Generalised <code>:For</code> Statements .....	38
§§ 5.2.3 <code>:Repeat</code> and <code>:While</code> Loops .....	38
§ 5.3 Digging .....	38
§§ 5.3.1 The <code>:With</code> Statement .....	38
§§ 5.3.2 Digging into SubSpaces .....	39
§§ 5.3.3 <code>:Trap</code> versus <code>□TRAP</code> .....	39
<b>Module6: In-Process OLE Servers .....</b>	<b>40</b>
§ 6.1 Creating an OLE Server in a DLL .....	40
§§ 6.1.1 The <code>OLEServer</code> Object .....	40
§§ 6.1.2 Exporting Variables and Functions as Properties and Methods .....	40
§§ 6.1.3 Saving and registering your <code>OLEServer</code> .....	41
§ 6.2 Variable type Information .....	41
§§ 6.2.1 Setting Method Information .....	41
§§ 6.2.2 Setting Property Information .....	41
§§ 6.2.3 Exploring the Registry Entry .....	42
§ 6.3 Using your OLE Server .....	42
§§ 6.3.1 The <code>OLEClient</code> Object .....	42
§§ 6.3.2 Examining Type Libraries .....	43
§§ 6.3.3 Calling an OLE Server from VB .....	44
<b>Module7: OLE Clients .....</b>	<b>46</b>
§ 7.1 Inside Microsoft Word .....	46
§§ 7.1.1 Registry Entries, Object Models and Type Libraries .....	46
§§ 7.1.2 Digging into Word .....	47
§§ 7.1.3 Demonstrating the Power of OLE .....	48
§ 7.2 Manipulating Microsoft Excel from the Inside .....	49
§§ 7.2.1 Recognising the Object Model .....	49
§§ 7.2.2 Digging into Excel .....	49
§§ 7.2.3 Gaining full Control of Excel .....	50
§ 7.3 Linking to other Servers .....	50
§§ 7.3.1 Outlook .....	50
§§ 7.3.2 Microsoft Internet Explorer .....	51
§§ 7.3.3 Beyond .....	52

<b>Module8: ActiveX Controls</b> .....	<b>53</b>
§ 8.1 Creating an ActiveX Control in an OCX .....	53
§§ 8.1.1 The <i>ActiveXControl</i> Object.....	53
§§ 8.1.2 The <i>Create</i> Callback .....	53
§§ 8.1.3 Creating the OCX on <i>)SAVE</i> .....	54
§ 8.2 Using your ActiveX Control .....	55
§§ 8.2.1 Creating an Instance from an <i>OCXClass</i> .....	55
§§ 8.2.2 Using Controls in IE 6.0.....	55
§§ 8.2.3 Using Controls in Visual Basic and VBA .....	56
§ 8.3 Browsing registered OLE Controls .....	56
§§ 8.3.1 Having a quick Look .....	56
§§ 8.3.2 Having a deeper Look .....	56
§§ 8.3.3 Trying some Examples .....	57
<b>Module9: C Function Access</b> .....	<b>59</b>
§ 9.1 Declaring “ <i>dataTypes</i> ” of Arguments and Results.....	59
§§ 9.1.1 Quick View of DLLs and their Contents.....	59
§§ 9.1.2 The Meaning of the right Argument of <i>⊞NA</i> .....	59
§§ 9.1.3 Discovering C Function Syntax.....	60
§ 9.2 Examples of C Function Calls.....	60
§§ 9.2.1 Simple Examples .....	60
§§ 9.2.2 More complex Examples.....	62
§§ 9.2.3 Other API Calls .....	64
§ 9.3 Harnessing large C Libraries .....	65
§§ 9.3.1 Fastest Fourier Transform in the World .....	65
§§ 9.3.2 Open Graphics Library .....	66
§§ 9.3.3 Linear Algebra Package .....	66
<b>Module10: Stand-Alone Applications</b> .....	<b>70</b>
§ 10.1 Building GUI Applications.....	70
§§ 10.1.1 The bare Minimum .....	70
§§ 10.1.2 Completing the Address Book Application.....	71
§§ 10.1.3 Enhancing your Address Book .....	71
§ 10.2: Making runtime Executables.....	71
§§ 10.2.1 Files to Include .....	71
§§ 10.2.2 Inifiles and the Windows Registry .....	72
§§ 10.2.3 The [File][Export] MenuItem.....	74
§ 10.3 Aspects of Pocket APL.....	74
§§ 10.3.1 Pocket Platforms.....	74
§§ 10.3.2 Creating the executable Program.....	76
§§ 10.3.3 Building a distributable Application .....	76
<b>Module11: Advanced Dot Syntax</b> .....	<b>77</b>
§ 11.1 Object Variables .....	77
§§ 11.1.1 Stranding Object Properties.....	77
§§ 11.1.2 Stranding Objects .....	80
§§ 11.1.3 Arrays of .. Arrays of Objects .....	83
§ 11.2 Understanding <i>(...).( ...)</i> .....	87
§§ 11.2.1 Expanding Array.Strand .....	87
§§ 11.2.2 Expanding Array.Array .....	94
§§ 11.2.3 Expanding Array.Function .....	96
§ 11.3 Arrays of Programs.....	99
§§ 11.3.1 Interpreting <i>.. (...)... (...).f<sub>1</sub></i> .....	99
§§ 11.3.2 Arrays of .. Arrays of defined Functions.....	100
§§ 11.3.3 Arrays of .. Arrays of defined Operators.....	101

<b>Module12: Dynamic Programs</b> .....	<b>103</b>
§ 12.1 Direct Definition.....	103
§§ 12.1.1 Programming DFns.....	104
§§ 12.1.2 MultiLine DFns.....	106
§§ 12.1.3 Guards and Error Guards.....	108
§ 12.2 Extended direct Definition.....	109
§§ 12.2.1 Programming DOPs.....	109
§§ 12.2.2 Idioms and Utilities.....	110
§§ 12.2.3 Object.Object. .. Object.Operator Rationale.....	111
§ 12.3 Recursion.....	111
§§ 12.3.1 Recursive Functions.....	111
§§ 12.3.2 Recursive Operators.....	113
§§ 12.3.3 Biological Beauties.....	114
<b>Module13: APL Threads</b> .....	<b>117</b>
§ 13.1 Spawning a new Thread.....	117
§§ 13.1.1 The Spawn Operator, &.....	117
§§ 13.1.2 Thread Identity from <code>⎕TID</code> and <code>⎕TNAME</code> .....	119
§§ 13.1.3 Thread Numbers with <code>⎕TNUMS</code> and <code>⎕TCNUMS</code> .....	120
§ 13.2 MultiThread Interactions.....	122
§§ 13.2.1 Thread Synchronisation with <code>⎕TSYNC</code> .....	122
§§ 13.2.2 Holding Tokens with <code>:Hold</code> .....	122
§§ 13.2.3 Pooling Tokens with <code>⎕TPUT</code> and <code>⎕TGET</code> .....	123
§ 13.3 General Thread Programming.....	124
§§ 13.3.1 Thread Switching.....	124
§§ 13.3.2 External Threads with <code>⎕NA</code> .....	125
§§ 13.3.3 Threading callback Functions.....	126
<b>Module14: TCP/IP Sockets</b> .....	<b>129</b>
§ 14.1 The <code>TCPSocket</code> Object.....	129
§§ 14.1.1 IP Addresses and Ports.....	129
§§ 14.1.2 <code>SocketType</code> and <code>Style</code> Properties.....	129
§§ 14.1.3 Workspace to Workspace Communications.....	130
§ 14.2 A simple character Socket.....	132
§§ 14.2.1 Connecting to a server Socket with <code>TCPConnect</code> .....	132
§§ 14.2.2 Sending to the server Socket using <code>TCPsend</code> .....	132
§§ 14.2.3 Receiving from the server Socket with <code>TCPRecv</code> .....	132
§ 14.3 Some Complications.....	133
§§ 14.3.1 HTTP and HTML.....	133
§§ 14.3.2 Buffering received Data.....	134
§§ 14.3.3 Servicing multiple Connections.....	134
<b>Module15: APL Web Servers</b> .....	<b>136</b>
§ 15.1 Making a simple Server.....	136
§§ 15.1.1 Creating a listening Socket.....	136
§§ 15.1.2 Cloning a listening Socket on <code>TCPAccept</code> .....	137
§§ 15.1.3 Sending an HTML File on <code>TCPRecv</code> .....	138
§ 15.2 Making a realistic Server.....	139
§§ 15.2.1 Threading multiple Connections.....	140
§§ 15.2.2 Communicating through HTTP.....	141
§§ 15.2.3 Running APL Functions on a Server.....	142
§ 15.3 Internet Practicalities.....	143
§§ 15.3.1 Domain Name Servers.....	143
§§ 15.3.2 Firewalls and proxy Servers.....	144
§§ 15.3.3 An ISP running Dyalog.DLL.....	145

<b>Module16: APL Web Clients.....</b>	<b>146</b>
§ 16.1 Getting to the outside World .....	146
§§ 16.1.1 Direct Connection through your Internet Service Provider.....	146
§§ 16.1.2 Proxy Servers and Firewalls .....	146
§ 16.2 Asking the Web .....	146
§§ 16.2.1 Connecting and sending the Question .....	146
§§ 16.2.2 Receiving and interpreting the Answer .....	147
<b>Module17: Dyalog.Net.....</b>	<b>148</b>
§ 17.1 Revealing the .NET Framework.....	148
§§ 17.1.1 Getting Microsoft .NET.....	148
§§ 17.1.2 Assemblies (à), Namespaces (ñ) and Classes (¢).....	148
§§ 17.1.3 Using □ <i>USING</i> .....	152
§ 17.2 Exploring the .NET Interface .....	153
§§ 17.2.1 Examining Classes.....	153
§§ 17.2.2 Examining Methods.....	154
§§ 17.2.3 Examining Properties .....	156
§ 17.3 Digging into .NET .....	158
§§ 17.3.1 Windows Forms.....	158
§§ 17.3.2 Communications.....	161
§§ 17.3.3 Generalising APL Primitives.....	163
<b>Module18: Dyalog.Net Classes .....</b>	<b>165</b>
§ 18.1 Writing Dyalog.Net Classes .....	165
§§ 18.1.1 Dyalog Namespaces and .NET Namespaces.....	165
§§ 18.1.2 Creating a <i>NetType</i> Object .....	165
§§ 18.1.3 Writing Functions and defining Variables .....	165
§ 18.2 Exporting Methods and Properties .....	165
§§ 18.2.1 Arguments and Result “ <i>dataTypes</i> ” .....	165
§§ 18.2.2 Making an Assembly .....	166
§§ 18.2.3 Checking the MetaData .....	167
§ 18.3 Calling Dyalog.Net Classes.....	167
§§ 18.3.1 Calling your Dyalog.Net Class from Dyalog APL.....	167
§§ 18.3.2 Calling your Dyalog.Net Class from C# and VB.NET .....	168
§§ 18.3.3 Complications.....	168
<b>Module19: Dyalog.Asp.Net.....</b>	<b>169</b>
§ 19.1 Dynamic Web Pages.....	169
§§ 19.1.1 Active Server Pages in VBScript or JScript.....	169
§§ 19.1.2 The <i>System.Web.UI.Page</i> Class .....	171
§§ 19.1.3 The <i>System.Web.UI.WebControls</i> Namespace.....	173
§ 19.2 Dyalog Script Language.....	174
§§ 19.2.1 Callbacks in Dyalog APL.....	174
§§ 19.2.2 Workspace behind .....	175
§§ 19.2.3 The TextBox Control.....	176
§ 19.3 Remote Applications .....	178
§§ 19.3.1 The C:\Inetpub\wwwroot\ Directory .....	179
§§ 19.3.2 The <i>System.Drawing</i> Namespace .....	180
§§ 19.3.3 The <i>System.Web.Services</i> Namespace .....	181
<b>Module20: Dyalog APL Classes .....</b>	<b>183</b>
§ 20.1 User defined Classes.....	183
§§ 20.1.1 The <b>:Class</b> Structure.....	183
§§ 20.1.2 The <b>:Field</b> Statement .....	185
§§ 20.1.3 Name sub Classifications of □ <b>NC</b> .....	186
§ 20.2 Methods and Properties in Classes .....	187

§§ 20.2.1 The $\nabla$ (Method) Structure .....	187
§§ 20.2.2 The <b>:Implements</b> Statement.....	188
§§ 20.2.3 The <b>:Property</b> Structure .....	190
§ 20.3 Architecture with Class Factories .....	192
§§ 20.3.1 Designing an Object Model .....	192
§§ 20.3.2 Building with Objects .....	194
§§ 20.3.3 Encapsulating, Inheriting and Morphing .....	194

Some innocent facts ☺

- There is no exact finite Boolean representation of the decimal number 0.1 as a binary floating-point number.
- The density of rational numbers on the real line is infinitesimal compared to the density of transcendentals.
- One Jupiter day equals 0.416.. Earth days.

Question ☺ Is there a planet somewhere whose day is approximately equal to 10 Earth days?

Answer ☺ Very likely! (On that planet, this is a two day teach-yourself course ☺.)

## Module0: Notation and Conventions

### § 0.1 New Symbols

Let us introduce a few new suggestive symbols with well-defined meanings suggestive of their shape. Most of these new symbols are **not** to be found in the □AV of any APL font and are simply introduced to clarify explanation and writing in the spirit of the original Iverson Notation. Some of these symbols are to be found in some APL fonts, but they are here not coloured green and are, by implication, not available as part of an executable Dyalog APL statement. Executable APL keywords are written in green.

An executable input expression is written in green APL font and indented by 6 spaces. Any resulting output from an expression is coloured red and is placed on the subsequent unindented line or lines, or on the same line as the input expression, separated from it by the instructive (non-executable) new (black) symbol ↵ .

$Expr \mapsto Result$	Expression $Expr$ returns result $Result$
-----------------------	---

The symbol ↵ (returns) is black to identify it as non-executable, and the definition is in black to emphasise that is not genuine (current) APL. The  $Result$  may also be an executable expression which itself would, if executed, return a result identical to that returned by the original expression,  $Expr$ . So, a valid illustration of ↵ would be  $3 \times 4 \mapsto + / 3 \rho 4$

$Expr \mapsto Obj$	$Expr$ instantiates object $Obj$ (as a side effect)
--------------------	---

This is used to document functions that internally create global GUI objects.

$Expr \mapsto Function$	$Expr$ can fire callback function $Function$
-------------------------	--

This is used to document expressions or functions that assign callbacks to events in global objects.

$APLCode \dots \downarrow$	Code continues on next line ...
$.. APLCode \mathcal{I}$	.. end of line of code

The pair ( $\downarrow, \mathcal{I}$ ) is used to wrap 2 or more input lines while maintaining their syntactic unity and integrity.

$\vdash Prop$	Proposition $Prop$ is contingently true
---------------	---

Symbol  $\vdash$  (contingently) means factually true, so here  $Prop \mapsto 1$ .  $\vdash a \vee b$  implies, *a posteriori*, that  $a \vee b \mapsto 1$

$\vDash Prop$	Proposition $Prop$ is necessarily true, so $Prop \mapsto 1$
---------------	---

Symbol  $\vDash$  (logically) establishes the logical truth of a proposition, so  $\vDash a \vee b$  implies, *a priori*, that  $a \vee b \mapsto 1$ , and further therefore, that  $(\vDash a) \text{ or } (\vDash b)$ , where *or* is understood to be inclusive rather than exclusive.

$Code .. Code$	Means fill in the gap with the obvious code
----------------	---

This single character .. (double-dot or *gap*) can also be used at the very start of a line (in an ambivalent fashion) to mean fill in the obvious start.

(Note in the modules to follow, boxed syntax definitions often just apply to one of a number of possible syntaxes. For example, possible ambivalence and shy results are not usually left unspecified.)

$Code \dots$	Means finish the statement with the obvious ending
--------------	--

This single character ... (triple-dot or *ending*) asks you, the reader, to complete the extreme end of any expression or suitably parenthesised expression. It asks more of you than standard dittos, which just imply repetition, and more than gaps (..), which are usually essentially repetition too, given the clues from the code on either side. An author who uses an ending should feel that the code to the left is sufficient for the reader to be able to fill in an appropriate ending without undue pain, *ie* suitable filler code should spring to mind.

You might consider the symbols .. and ... as notation for  $x$  or  $y$  in some 'literal algebra'. eg "Hello .., this is .., how are you ... ?" where .. and ... are any suitable, possibly empty, strings.

$$() \quad \vdash () \equiv \emptyset$$

This implies that  $() \equiv \emptyset \vdash 1$  and that therefore  $\vdash(, 0) \equiv \rho()$

Other symbols which may be introduced in the following modules include:

$f_1$  as a generic monadic function (distinguish *monadic* function from *monistic* operator),

$g_2$  as a generic dyadic function (distinguish *dyadic* function from *dualistic* operator),

$\hat{o}_a$  to represent a typical unipotent operator and

$\tilde{n}_a$  to represent a typical namespace.

## § 0.2 Naming Conventions

The following suggested naming conventions have been adopted where possible and where appropriate.

Variable names are regarded as proper nouns and therefore start with a capital letter; an exception being allowed for local *temporary variables* used near to their creation origin. These exceptions may be regarded as pronouns for which we may use small letters and shortened words.

Functions and operators play the roles of verbs, adverbs, adjectives, conjunctions or prepositions depending on the details of their syntax. Their names generally start with a small letter, as they are mid-sentence words. However, *niladic functions* may start with a capital because, if they return a result, they are syntactically like variables, and if they do not return a result they are like complete imperative intransitive verb sentences. Likewise, *monadic functions that do not return a result* may also begin with a capital letter because they are like imperative transitive verbs and therefore usually begin a sentence.

To give you the idea, an attempt has been made to compose the Course Contents roughly in this style. Module titles are noun phrases that represent the subject matter. Section and subsection headings are subjects or predicates or object noun phrases. An attempt is thus made to capitalize the words according to their contextual parsing character as elaborated in APL Linguistics in *Vector Vol.2 No.2 p118*.

Sentence Syntactic Elements	Grammatical Rôle	Example Names
<i>Variable</i>	Noun/Pronoun	<i>Data</i> $\diamond$ <i>DATA</i> $\diamond$ <i>d</i>
<i>Niladic</i>	Intransitive imperative	<i>Run</i> $\diamond$ <i>RunApplication</i>
<i>R<math>\leftarrow</math>Niladic</i>	Noun	<i>.. is Temperature</i>
<i>Monadic</i> $\omega$	Transitive imperative	<i>RunOn ...</i>
<i>R<math>\leftarrow</math>monadic</i> $\omega$	Adjective/Participle	<i>.. is purple ...</i>
$\alpha$ <i>dyadicFn</i> $\omega$	Verb	<i>.. takes ...</i>
<i>R<math>\leftarrow</math><math>\alpha</math> dyadic</i> $\omega$	Preposition/Conjunction	<i>.. is .. tiedTo ...</i>
<i>R<math>\leftarrow</math>(<math>\alpha\alpha_1</math> monistMonadOpr)</i> $\omega$	Adverb	<i>.. is .. accurately ...</i>
<i>R<math>\leftarrow</math><math>\alpha</math> (<math>\alpha\alpha_1</math> monistDyadOpr)</i> $\omega$	Adverb	<i>.. is .. .. separately ...</i>
<i>R<math>\leftarrow</math>(<math>\alpha\alpha_1</math> dualMonadOpr <math>\omega\omega_1</math>)</i> $\omega$	Conjunction/Preposition	<i>.. is .. togetherWith ...</i>
<i>R<math>\leftarrow</math><math>\alpha</math> (<math>\alpha\alpha_1</math> dualDyadOpr <math>\omega\omega_1</math>)</i> $\omega$	Conjunction/Preposition	<i>.. is .. .. and .. ...</i>
etc...		

This loose heuristic pragmatic association with ordinary language is for the purpose of producing meaningful readability. Note the convention of capitalizing the first letter of each word or syllable after the first rather than using the APL-specific underscore character to separate words, as was an older convention.

Object names follow APL *variable* naming conventions. Object method names should follow *function* conventions but are often predetermined by another author. From a glance at the method names in IE6 (or OpenGL) Microsoft and others may be moving in the above direction. Object property names are nouns and are generally, by our convention, ‘correctly’ capitalized.

An effort has been made to capitalize and colour keywords in the text when used as keywords rather than as general concepts: *eg* an object initiates an event, *cf* an object has an *Event* property. (Choosing long meaningful names is less painful than it used to be as AutoComplete springs into action in version 10.)

Pre-empting a Unicode (see <http://www.unicode.org>)  $\square AV$ , letters in names here assume all the familiar characteristics of modern fonts, *ie* you can have  $A A A A \underline{A} \underline{A} \underline{A} A$  etc... all for the price of  $\square AV [ 6 6 ]$  and all equivalent, just as an ordinary word is assumed to have the same meaning in any colour. The three alphabets in  $\square AV$  are taken to be  $A \dots Z$   $a \dots z$   $a \dots z$  *ie* capital letters, small letters (possibly subscripted) and superscripted letters of various styles. This is done to enable full tensor notation such as  $(X_1, X_2, X_3)$  or  $T_{ijkl} \leftarrow G_{im} + . \times T^m_{jkl}$  (as used in *eg* <http://arxiv.org/ftp/hep-th/papers/0304/0304244.pdf> ;-)) and also to encourage traditional British shorthand forms such as  $W^m$  Robertson &  $C^o L^{td}$  or  $Sci^{ntfc}$  &  $Med^{cl} Net^{wk}$ . We also suppose, in these notes, that *any* linear Rich Text may follow a comment symbol in the APL Session and we assume that any number of new symbols (here written in black) will one day be available in green for APL programmers.

Menus and other one-click options relating to an application under discussion are placed in square brackets. Thus [File][Load] and [Tools][Options] might refer in an obvious way to specific APL Session menu items.

### § 0.3 Variable “dataTypes”

Different languages use different words to indicate the number of bytes employed and the interpretation of the individual bits. For example, the C programming language, which we meet via  $\square NA$  in Module 9, describes numbers as integer, double, long double, float, .. whereas  $\square NA$  itself uses notation similar to that used in the Larg (left argument) of  $\square FMT$  which itself originates from FORTRAN nomenclature. More recent definitions of numeric “dataTypes” are written VT\_I2, VT\_I4, VT\_DECIMAL, VT\_R4, VT\_R8, ...

Happily, APL does not generally demand that a programmer be aware in advance of exactly what type of binary representation is being used for any particular piece of data. Mathematica, J and some other modern computer languages are even more forgiving - numbers may even become characterised as infinite!

In pure mathematics, occasionally the domain of numbers in which one is thinking has to be generalised to the next set. Whole numbers are a subset of natural numbers which are a subset of the group of integers which is a subgroup of the group of rational numbers which is a subgroup of the field of real numbers which is a subfield of the complex field which is a subfield of the non-commutative quaternion ring which is a subring of the non-commutative, non-associative octonion ring.

$$N \subset Z \subset Q \subset R \subset C \subset H \subset O$$

There is a natural hierarchy that is embraced at each level, as and when required, when learning advanced arithmetic. Thus when learning, at an early age, that division of integers can land you in the rationals, you are not generally obliged thenceforth to predetermine whether the result of any particular division will result in a number which is integer or one which is non-integer (although necessarily rational).

In APL the actual representation employed for storing a number can change from line to line and from function call to function call without the programmer having to know. The interpreter itself attempts to discover the most efficient storage representation. However, when communicating through APL with the computer world at large you have to start thinking about machine architecture a bit more. You often have to tell functions, in advance, exactly what sort of arguments to expect and exactly what sort of results to generate. This is the *dataType* signature of a function. It helps hardware to know how to store and retrieve.

### Loosely Correlated *data*Type Definitions (of internal memory formats)

APL if you had to say...	⊞NA	C	Interface Definition Language (OLE/COM)	Visual Basic	C#	.NET <i>data</i> Types
CharSc	C1, T	uchar		Byte	byte, char	System.Byte, System.Char
CharVec	C, T[]	char[], char*	VT_BSTR, VT_PTR TO VT_BSTR	String	string	System.String
VecCharVec	{T[4]}[5]	(char* ) []	VT_ARRAY OF VT_BSTR, VT_STRING[]	String(0 to 12)	string[]	
NumSc, BoolSc, IntSc	I2, I4, F8, U1, U4	short, uint, int, long, float, dword, lptstr double	VT_BOOL, VT_DECIMAL, VT_I1, VT_I2, VT_I4, VT_R4, VT_F8, VT_R8, VT_PTR TO VT_UI4	Integer, Long, Single, Double, Decimal, Boolean, Date	sbyte, short, int, uint, ulong, float, double, bool, decimal	System.SByte, System.Int16, System.Int32, System.Int64, System.UInt16, System.UInt32, System.UInt64, System.Decimal
NumVec, IntVec	{F8 I2}, I[], F8[]	Int []	VT_CY, VT_DATE, VT_I4[], VT_COLOR,	Integer(1 to 3)	int[], double[]	
Arr	A	void[], int[]	VT_VARIANT, VT_ARRAY	Variant()	object	System.Object, System.Array
ArrEncArr	{I4 T[9]}, {I4 U4 {U1[4]} I4 I4}	struct	VT_ARRAY OF VT_VARIANT, VT_SAFEARRAY	Variant( 1 to 3, 1 to 4, 1 to 2)	object	System.Object
MatEncArr			VT_VARIANT[;]		int[[,]	System.Object
RefSc			VT_DISPATCH, VT_PTR TO VT_COCLASS	Object		System.Object, etc...
Empty, ⊞NULL		void	VT_VOID,	Empty, Nothing, Null	void	System.Void
⊞OR of space			VT_DISPATCH			

0.3.1 Review **APL1&2.PDF** which introduces mainstream 1<sup>st</sup> and 2<sup>nd</sup> generation APLs such as *IBM APL2*, *Dyalog APL*, *STSC APL\*PLUS* and *MicroAPL APL.68000*.

Note: You can download **APL1&2.PDF** free from <http://www.microapl.co.uk/apl/APL1&2.PDF> .

## Module1: Objects and their Properties

Amongst **Ken Iverson**'s many aphorisms is the cautionary dictum not to labour too hard on any particular explanation of any particular fine point of APL notation (such as `0 ρ 0`) because it is as likely to be an indication of an unsuccessfully hurdled hurdle behind the tutor as an especially difficult conceptual leap for that audience. We try to adhere to his advice, albeit that APL becomes not more, but less rational with each new generation, and grey areas should be exposed and not left hidden. Nevertheless Dyalog APL does an admirable job of incorporating object-oriented concepts into the core of the APL language.

In 1967 Simula 67 introduced most of the key concepts of **object-oriented (OO) programming** - objects, classes, subclasses (involving inheritance) and virtual procedures. Most (but not all) of the objects discussed below have a graphical (visual) manifestation. The first **graphical user interface (GUI)** was designed by Xerox Corporation's Palo Alto Research Center in the 1970s. It was not until the 1980s and the emergence of the Apple Macintosh that GUIs started to prevail, and not until the appearance of Microsoft Windows in the early 1990s that GUIs became almost ubiquitous.

Modern computer applications, including operating systems themselves, are designed using object-oriented architecture. Microsoft's graphical user interface has evolved into an archetypically object-oriented collection of buttons, forms, and other progressively more complex active constructs. Most of Dyalog's 71 or so Microsoft-based GUI objects are virtually tangible, arguably real and thoroughly recognisable.

<sup>1.1</sup>For a comprehensive list of primitive GUI objects available in Dyalog APL version 10, see [Help][GUI Help][Objects Overview]. Or type *Type* into the APL session and hit **F1**. Or see the invaluable [Dyalog APL Object Reference](#) manual. Load workspace *WINTRO* and follow the 56 lessons. Load *WTUTOR* and follow 37 tutorials in there. Load *WTUTOR95* and follow the 18 extra tutorials in there. Alternatively, begin your investigations at <http://www.dyalog.com> [Products]. But first, please follow Modules 1-3 below for a simple introduction to this vast encyclopaedic network of information about APL objects 😊.

### § 1.1 Object Spaces

#### §§ 1.1.1 Creating vanilla Namespaces with `⊞NS`

<code>CVec ⊞NS ''</code>	Ⓐ Creates an empty Namespace, named through <i>CVec</i>
--------------------------	---

<sup>1.1.1.1</sup>Create an empty namespace called MySpace.

<code>)CS Name</code>	Changes current space into Namespace named <i>Name</i>
-----------------------	--

<sup>1.1.1.2</sup>Change space into your new namespace and verify that `⊢(⊞NL ι4)=0 0 ρ ''`  
Read this as, ".. it happens to be true that `⊞NL ι4` matches an empty character matrix." Compare with `⊢1∨0` which reads, ".. it is necessarily true that `1∨0`."

<code>)CS</code>	Changes Space to the workspace Root space - named #
------------------	---

<sup>1.1.1.3</sup>Change back to the Root space using system command `)CS` and examine the result of `⊞NL 9`

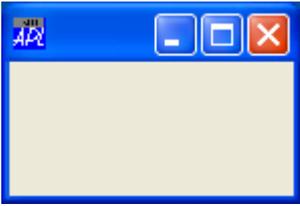
#### §§ 1.1.2 Creating GUI Object Spaces with `⊞WC`

<code>CVec2 ⊞WC CVec1</code>	Ⓐ Creates GUI Object named in <i>CVec2</i> of type in <i>CVec1</i>
------------------------------	--

<sup>1.1.2.1</sup>Create an object of *Type Form* with name MyForm. The structure of the right argument of this system function can be much more complicated (essentially *Name-Value* pairs) as we shall see later.

<code>)OBS</code>	Displays a list of global objects
-------------------	-----------------------------------

<sup>1.1.2.2</sup>Use system command `)OBS` to verify the existence of MyForm, although its existence is manifest:



1.1.2.3 Try moving and resizing your *Form* with your mouse.

With a less obvious existence are other objects, such as the *SysTrayItem* object created by typing

```
'STI'⊞WC'SysTrayItem'
```

But notice the new APL icon that consequently appears in the system tray at bottom right of your task bar.

### §§ 1.1.3 Changing Space with `⊞CS`

```
⊞CS CVec
```

⌘ Changes current space to object named in *CVec*

An object of a particular *Type* can only exist as the child of particular parent *Types*. For example, a *Button* can be the child of a *Form* but not of a *SysTrayItem* object.

1.1.3.1 Change space to MyForm and create an object of *Type Button* with name MyButton.



The *Button* has no *Caption* because this property has not yet been specified.

1.1.3.2 Change into MyButton space and verify `⊞NL 19↵0 0ρ''` then change back to the Root space (#).

## § 1.2 Properties of Object Spaces

Objects have properties. These properties determine the specific appearance and behaviour of individual objects. For example, the *Size* property, which is common to many objects, determines the height and width of the object.

### §§ 1.2.1 Examining Properties of an Object with `⊞WG`

Some information about MyForm object can be discovered by right-clicking on the word MyForm in the APL Session and then selecting the menu item [Properties]. However, the value of a specific individual property is found by way of the property name.

```
Arr ← CVec2 ⊞WG CVec1
```

⌘ Gets the value of property *CVec1* of object *CVec2*

1.2.1.1 Get the *Size* of MyForm. Go into MyForm space and get the *Size* of MyButton.

```
)PROPS
```

Reports the properties in the space of the current object

1.2.1.2 Display the list of properties in the MyForm space and the MyButton space. These are the intrinsic properties of *Forms* and *Buttons* respectively. Examine the values of various properties. Return to #.

### §§ 1.2.2 Setting Properties of an Object with `⊞WS`

```
CVec2 ⊞WS CVec_Arr
```

⌘ Sets prop-value pair *CVec\_Arr* of object *CVec2*

For example,

```
'MyForm'⊞WS'Size' (6.5 11)
```

sets the *Size* of MyForm to 6.5% by 11% of the full screen size.

```
'MyForm'⌈WS'Caption' 'Form' ⌋ ⌈CS 'MyForm'
'MyButton' ⌈WS'Caption' 'Button'
```

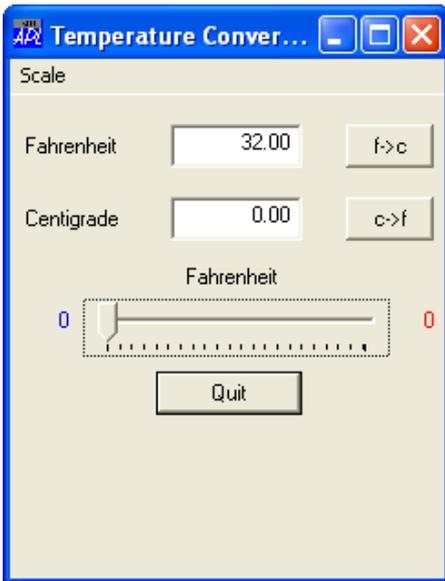


1.2.2.1 Try getting and setting the values of various properties of *Forms* and *Buttons*. Return to # and erase your *Form* with *)ERASE* or *)EX*.

As we shall soon discover, it is possible to get (*)WG*) and set (*)WS*) many properties in a single call, and to set many properties at create time (*)WC*). Indeed a few (irregular) properties, such as the *Points* property, *must* be set at create time, as must *Type*. Most properties, however, have sensible default values and do not need to be explicitly set at all on most occasions.

### §§ 1.2.3 Building complex Objects

*)WC* returns a shy result of the name of the object just created (*Larg*). Here are some lines that illustrate the flexibility of the right argument (*Rarg*) of *)WC* and *)WS*. Note that *Name-Value* pairs do not need the *Name* if specified in the default property order (see for example the *Menu* and *MenuItem* objects below).



```
w←,←'Type' 'Form' ⌋ w,←'Caption' 'Temperature Conv↵
erter'⌋ w,←'Size'(266 238)⌋ w,←'Coord' 'Pixel'⌋
⌈CS'Temp'⌈WC w
⌈CS'MB'⌈WC'MenuBar'⌋ ⌈CS'M'⌈WC'Menu' 'Scale'
'C'⌈WC'MenuItem' 'Centigrade'⌋'F'⌈WC'MenuItem' 'Fah↵
renheit'⌋⌈CS'#'⌋⌈CS'Temp'⌋num←'FieldType' 'Numeric'⌋
'LF'⌈WC'Label' 'Fahrenheit'(16 8)(24 72)
'F'⌈WC'Edit' ''(16 88)('Decimals' 2)num
'F'⌈WS('ValidIfEmpty' 1)('Value' 32)('Size' 24 72)
'F2C'⌈WC'Button' 'f->c'(16 184)(24 48)
'LC'⌈WC'Label' 'Centigrade'(56 8)(24 72)
'C'⌈WC'Edit' ''('Decimals'(2))num
'C'⌈WS('ValidIfEmpty' 1)('Size' 24 72)('Posn' 56 88)
'C2F'⌈WC'Button' 'c->f'(56 184)(24 48)
'LTB'⌈WC'Label' 'Fahrenheit'(88 96)(24 72)
'LLO'⌈WC'Label' ''(112 16)('Decimals' 0)num
'LLO'⌈WS('FCol'(0 0 255))('Value' 0)('Size' 24 16)
'TB'⌈WC'TackBar'('Limits'(0 212))('Posn'(112 40))
'TB'⌈WS('Size'(32 168))('TickSpacing' 10)
'LHI'⌈WC'Label' ''(24 24)('Decimals' 0)num
'LHI'⌈WS('FCol'(255 0 0))('Value' 0)('Posn' 112 208)
'Q'⌈WC'Button' 'Quit'(152 80)(24 80)('Default' 1)
```

1.2.3.1 Verify that a *Menu* is only visible if it has set a non-empty *Caption*. A *MenuBar* is never visible.

<i>)NS</i>	Reports the name of the current space
------------	---------------------------------------

### § 1.3 Property Variables

It is possible to create ordinary APL variables in vanilla as well as in GUI namespaces.

1.3.1 Create a namespace called *MySpace* and inside it create a variable called *MyVar*. Verify that the variable is only visible inside *MySpace* and is invisible from the *Root* space.

### §§ 1.3.1 Exposing Object Properties with `⌈WX`

`⌈WX`

⌈ Whether GUI names are exposed

Every object has a property called `PropList` that lists all the object's properties in default order. `⌈WX` is a localisable system variable that determines whether or not the names of properties, methods and events provided by Dyalog APL GUI objects are *exposed*. The value of `⌈WX` in a clear workspace is defined by the `default_wx` parameter (see the definitive [Dyalog APL User Guide](#)). Ensure that `⌈WX=0`.

1.3.1.1 Create a Printer object. Check that `⌈WX` has inherited the value 0 from the Root. Type `PropList` and get a `VALUE ERROR`. Now assign `⌈WX` to 1 and again type `PropList`. The property variable has now been *exposed*. The listed keywords are case dependent and cannot be erased. Show that they are exposed.

### §§ 1.3.2 Assigning Properties with `←`

Notice that most properties, having been exposed, act just like variables in the object space. Go into a `Form`'s object space and type

`Size`

50 50

1.3.2.1 By manipulating object properties, the state of an object may be changed and influenced. Try assigning `Size` and notice the `Form` dynamically change size accordingly. Show that there is a minimum `Size` of a `Form`. Investigate the effect on `Size` and `Posn` when setting the `Coord` property to `'Pixel'`. Every object has a `Data` property to which may be assigned any APL array. Indeed, any other legitimate variable may be assigned within the object space and treated like a new static property. Verify these statements.

1.3.2.2 Create a `Bitmap` object in a `Form` with the `Bits` property set to a 150 by 150 matrix of random numbers between 0 and 15. Assign the `Picture` property of the `Form` to the name of the `Bitmap` that you have just created. This displays the `Bitmap` in the centre of your `Form` in colours chosen from the first 16 rows of the default colour map. Now assign the `CMap` property of the `Bitmap` to a 16 by 3 matrix of random numbers between 0 and 255. The picture changes each time this property is so set.

1.3.2.3 You can get more control over the `Bitmap`. Create an `Image` object as a child of a `Form` and then assign the `Picture` property of the `Image` to the name of the `Bitmap`. Assign the `Dragable` property of the `Image` to 2 and use the mouse to drag the `Image` around the `Form`. (Remember that the `Points` property is irregular in that it must be set explicitly at object create time.)

1.3.2.4 Using a `Poly` object, draw a solid triangle in the middle of a `Form`. Rotate it.

### §§ 1.3.3 Rebuilding complex Objects

Essentially, properties are (shared) variables and can be treated as such. All use of `⌈WG` and `⌈WS` may be obviated in favour of this more natural approach as of Dyalog version 9.0.

1.3.3.1 Rewrite the lines of code in §§1.2.3 to eliminate the use of `⌈WS` and to minimise the Rarg of `⌈WC`.

Now you have to learn some details about some of the primitive objects available to you in Dyalog APL. Typing any GUI keyword into the APL Session and hitting **F1** brings up the easily navigable `GUI Help` file at the page of the selected keyword topic. Learning is greatly facilitated by the extensive, thorough and surprisingly comprehensive, [Dyalog APL Object Reference](#) manual.

1.3.3.2 Ask for the next module on **methods** and **events**. How did you get on with Module 1? 😊

## Module2: Methods and Events

Objects can be brought to life by built-in functionality that is intrinsic to the object (methods and default event processing) or by arbitrary user-defined functions (callbacks) which are assigned to events in such a way as to run whenever the particular event occurs as a result of some user interaction with the object.

### § 2.1 Object Methods

A method is essentially a type of event that can only be generated under program control. A method may perform an operation and may return a result.

#### §§ 2.1.1 Enqueuing Object Methods with `□NQ`

```
2 □NQ CVec_Arr
```

```
▫ Enqueues method given by Arr for object CVec
```

`□NQ` adds the method specified in `Arr` to the end of the Windows *event queue*. The method will subsequently be processed as soon as it reaches the front of the queue. If the method returns a result then it is returned by `□NQ` as a shy result.

2.1.1.1 Investigate the methods associated with a *Form* by way of the *MethodList* property or the `)METHODS` system command, and try

```
2 □NQ(= 'MyForm' ), = 'ChooseFont'
```

#### §§ 2.1.2 Invoking Event default Action using `□NQ`

The default action of an event can be invoked by calling the event as a method.

2.1.2.1 Investigate the events associated with a *Form* by way of the *EventList* property or the `)EVENTS` system command and try

```
2 □NQ'MyForm' 'Configure' 10 10 20 20
```

This has the same effect as moving the *Form* to *Posn* 10 10 and resizing it to *Size* 20 20 but without invoking (because of `Larg 2`) any associated callback function on the *Configure Event* of the *Form*.

Unnecessary (through lenience) catenations and enclosures in the above right arguments for `□NQ` may be included to emphasise the separation of the object name and method calling information. The forgiving nature of `□NQ` syntax (like that of `□WC`) can obscure the essential components of the arguments. At this point `DISPLAY.DWS` or `varChar.exe` might help to clarify some details of strand notation.

(Note that it can be useful to put system commands onto programmable function keys with `□PFKEY`.)

#### §§ 2.1.3 Method Functions

When `□WX` is set to 1(or 3) then not only are property names exposed and become immediately accessible as pseudo-variables, but also method and event names are exposed and become pseudo-functions. Thus

```
□CS'MyForm'□WC'Form'  
Configure 10 10 20 20
```

will configure the *Form* as above.

2.1.3.1 Experiment with mouse events on a *Button*, adding a *Caption* to the *Button* for clarity

```
□CS'MyButton'□WC'Button'  
MouseDown 50 50 1 0  
MouseUp 50 50 1 0
```

Notice that when you are inside MyForm the *Close* event does make the *Form* disappear, but the remaining vanilla namespace called MyForm does not disappear until you exit the space.

## § 2.2 Object Events and callback Functions

An event may occur as a consequence of a user action. The user action triggers default behaviour and/or some programmer-defined process of arbitrary complexity.

### §§ 2.2.1 Firing Events by User Actions

2.2.1.1 Create a *Form* with a *Calendar* on it.

```
'F'⊂WC'Form' ⋄ 'F.C'⊂WC'Calendar'
```

Hit the right and left cursor keys to move the highlight day back and forth. Given that

```
⊂KL'RC'↳'Right'
```

experiment in the F namespace with

```
2 ⊂NQ'C' 'KeyPress' 'RC'
```

```
⊂NQ'C' 22 'RC'
```

and in the C namespace with

```
KeyPress 'RC'
```

On the one hand the event may be triggered by user action, on the other hand it may be generated under program control.

2.2.1.2 Use the *GetEnvironment* method of the Root object (or alternatively use REGEDIT.EXE) to discover the name of the APL input (.DIN) file. Open that file in Notepad and view the key labels section. Which of these could apply to the *Calendar*? Look at the *EventList* for the *Calendar* and guess which events are triggered by which keystrokes. Check whether some do what you expect.

### §§ 2.2.2 Attaching callback Functions to Events

Functions may be associated with events by way of the *Event* property. The *Event* property is very flexible and tolerant (lenient) in its assignment. We shall focus on the simplest forms of assignment and leave the more complicated forms aside for now.

In its simplest form, the *Event* property expects a 2-element vector of character vectors, the first element being the name of the particular event and the second being the name of the callback function. (View the default content of this property in varChar.) For example, in a clear workspace,

```
⊂CS'F'⊂WC'Form'
```

```
Event←'MouseMove' 'MoveForm'
```

where we here define the callback function simply as

```
▽MoveForm Msg
```

```
[1] Posn←Posn+Msg[3 4]÷100▽
```

The Rarg of a callback function is supplied automatically if required by APL syntax, and its content is determined by the event in question. In any event, the first element supplied contains the object in question and the second element contains the event in question. In the case of a *MouseMove* event, for example, the definition of the remaining 4 elements may be found in [Help][GUI Help]. In particular, the third and fourth elements of the event message refer to the coordinates of the position of the mouse.

2.2.2.1 Move the mouse over the *Form* and explain the effect. Put a stop on the first line of the callback and investigate the elements of the incoming right argument.

The *Event* property may be assigned to more than one event. Each assignment only affects the events named in the assignment. All other events retain their current settings.

Event names may be prefixed with "on" to give a new set of properties that can obviate the need for the *Event* property, and make it redundant. For example,

```
onMouseMove←'MoveForm'
```

results in behaviour very much like that produced by setting the *Event* property, but with one subtle difference in the message right argument automatically supplied to the callback.

2.2.2.2 Can you discover the difference? Use varChar if necessary.

Callback functions may return a result. If there is no result, or the result is 1, then APL proceeds to process the default action for the event. If the result is 0 the default processing does not take place. If the callback function modifies the incoming argument and returns that modified message as the result of the callback function then APL (leniently) processes default action in accordance with that new message. (APL is even surprisingly lenient with a would-be *VALUE ERROR* result and a niladic callback.)

### §§ 2.2.3 Bringing Objects to Life

Now we have all the ingredients necessary to breathe life into primitive objects - and into 'superobjects' built by straight-forward 'superposition' of primitive Dyalog GUI objects.

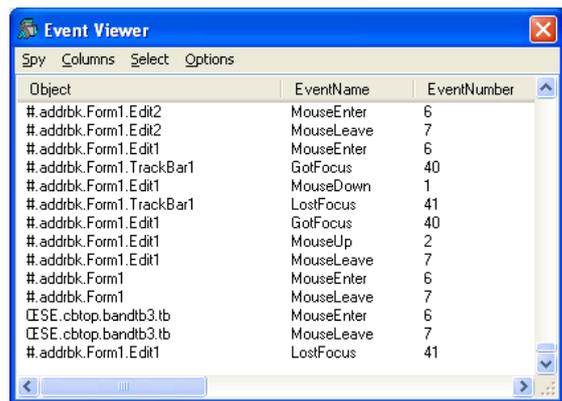
There is a poorly documented but very useful event called *All* that is recognised by every primitive GUI object. This event can be used to associate a callback function with *all* the events relating to an object. Write a trivial callback that just prints its message argument in the Session. Assign it to the *All* event.

```
▽ Show Msg
[1]   Msg
▽
onAll←'Show'
```

Now use the mouse and keyboard to interact with the *Form*. Doing this you will discover many ways in which you can programmatically intervene in the default behaviour of a *Form* using callbacks. This technique is a good way to explore the event structure of any GUI object.

In version 10.1, under the [Tools][Spy] menu, or in version 11, under the [Tools][Event Viewer] menu, there is a new Event Viewer that allows you to spy on events in a similar manner to the above, but with much more control through many extra features.

Compare this with the EventViewer in [Control Panel][Administrative Tools].



## § 2.3 The Event Queue

### §§ 2.3.1 Dequeuing Events with and without *DDQ*

Given the elements above it would be possible to write an APL cover-function to control the events and activities associated with collections of GUI objects. However, as the complexities of superobjects grow, and the numbers of combinations of events multiply, and 'heavy' callbacks begin to take noticeable amounts of time to run, the administration of events waiting to be processed becomes a significant issue.

In Immediate Execution Mode (IEM) events are processed as they arrive in the queue and the need for further control might seem unnecessary. However, IEM is *not* available in runtime systems and an alternative then becomes essential.

```
□DQ CVec
```

▫ Dequeues events associated with object CVec

□DQ takes the name of a top-level object as its argument and administers its events, plus any events from subobjects (child objects). (□DQ may also take a vector of top-level names, or most simply '#'.)

```
'F'□WC'Form'
□DQ'F'
```

2.3.1.1 Write a callback on the *Configure* event of *Form* F which resizes the *Calendar* to fit the *Form* exactly whenever the *Form* is resized. What is the difference between running in IEM and under □DQ?

The default value for an *Event* is zero. So, if the *Close Event* for a *Form* has not been set to a callback function or any other action code then *onClose* ← 0. Zero means handle the event normally, *ie* close the form. Conversely, if the *Event* action code is set to -1 then the event is entirely ignored.

2.3.1.2 Investigate the consequence of

```
onClose←-1
```

or, alternatively,

```
Event←'Close' -1
```

both in IEM and under □DQ.

Note that the way to terminate □DQ as a programmer developing a system is either by the keyboard **Ctrl+Break**, or by the Session menu [Action][Interrupt], or via the SysTray APL icon - select [Weak Interrupt] or [Strong Interrupt].

Within the program itself, one way to terminate □DQ is via action code 1.

```
onMouseMove←1
```

This causes □DQ to terminate when the mouse is moved over the *Form*. We shall make use of this shortly.

### §§ 2.3.2 Tracing □DQ

□DQ'F' is just like the "immediate" default processing of the Session with one very significant difference - **YOU CAN TRACE INTO IT!** It is possible on a callback to put a □STOP which causes it to suspend when run, but tracing into □DQ is a much more powerful means of debugging applications. (Use [Options][Configure][Trace/Edit] with Classic Dyalog mode and Independent trace stack checked!)

2.3.2.1 In the Root space, write a function (as in §§ 2.2.3) called *vShowv* which simply displays its right argument. Then create a *Form* with some *Event* or *Events* set to call *vShowv*. For example,

```
'F'□WC'Form' ('Event' 'All' 'Show')
```

Then trace □DQ by typing

```
□DQ'F'
```

and hitting **Ctrl+Enter**. Tracing □DQ'F' allows you to see all the callback code that is running - this is a very important ingredient in debugging GUI applications.

```
Msg←□DQ CVec
```

▫ Dequeues events of CVec and returns a message

It is important to know how to terminate □DQ under program control. This is especially important for time-consuming processes that the user might want to terminate early. Controlled interruption can be achieved in a two-step arrangement. First set some arbitrary event on the object to action code 1.

```
'F' WS'Event' 501 1
```

Enqueuing an event with action code 1 just before a `DQ` can allow you to pass through a `DQ` in a loop while waiting for some other `Event`. Then analysing the result of `DQ` allows one to continue, or not.

2.3.2.2 By placing line

```
ENQ'F' 501
```

in a callback function, show that this will terminate `DQ` as soon as the enqueued event reaches the top of the stack of events to be dequeued. (Tracing is not usually fast enough to both enqueue then dequeue.)

2.3.2.3 Show that it is possible to monitor some arbitrarily complex looping process in a function, such as that below, while still displaying a GUI via `DQ`

```
▽ Process;I;Sink
[1] 'F' WC'Form'('Event' 'Close' 1)('Event' 501 1)
[2] I←0
[3] Loop:→End×110=I←I+1
[4] ENQ'F' 501      A Enqueue event 501
[5] R←R←DQ'F'      A Dequeue event at top of queue
[6] →(501≠2>R)/0    A Was that event a 501 or a Close?
[7] Sink←□□□□□□□□□□?200 200p10000 A Good on a 3GHz machine :)
[8]   ◇ ◇ ◇        A eg ProgressBar object in here?
[9]   →Loop
[10] End:
▽
```

Why does tracing through line [4] not have the desired effect? (Instead put a stop on line [6] and run.)

2.3.2.4 Trace `DQ'SE'` and interact with the session.

Immediate Execution Mode makes most objects come alive and be usable - but a few special objects actually need `DQ` to give them life. For example, a `MsgBox` requires a `DQ` to make it visible.

```
'MBX' WC'MsgBox'('Style' 'Info')('Caption' 'Info')('Text' 'Msg')
```



```
DQ'MBX'
```

2.3.2.5 In [Help][GUI Help], investigate the `Wait` method and the objects to which it applies.

### §§ 2.3.3 Defining complex Behaviour

You now have all the ingredients necessary to write complex GUI applications that call arbitrary APL functions as a result of user actions. These functions can modify the GUI itself or create new GUI objects and pass user control from one object to another. Essentially, all the visible GUIs and the all functionality to be found in Microsoft Office applications you can now reproduce in Dyalog APL applications!

It is quite a conceptual leap to go from traditional linear programming to object-oriented programming, but you might not have that baggage... A GUI object on a user's screen may now be teeming with programmed hotspots that are ready to spring into action at the whim of the user and radically change his virtual world (or indeed her real world).

2.3.3.1 Below is a function that creates a *Form* with a number of controls on it, and associates a number of (undefined) callback functions with these controls. Run the function (via [Edit][Paste Non-Unicode]). Consider various alternative styles in which this function might have been written. Discuss with your partner good practices and what the callback functions might do in the completed application.

```

▽ MakeForm
[1]   □CS'Form1'□WC'Form' 'Address Book'(60 268)(266 238)↵
      ('Coord' 'Pixel')('Event'('Close' 1))⍉
[2]   'Label2'□WC'Label' 'Name:'(0 8)(24 32)↵
      ('Attach'('Top' 'Left' 'Top' 'Left'))⍉
[3]   'Edit1'□WC'Edit' ''(0 48)(24 184)↵
      ('Attach'('Top' 'Left' 'Top' 'Right'))('FCol'(0 0 192))↵
      ('Event'('KeyPress' 'eEditKeyPress'))⍉
[4]   'Label3'□WC'Label' 'E-mail:'(24 8)(24 32)↵
      ('Attach'('Top' 'Left' 'Top' 'Left'))⍉
[5]   'Edit2'□WC'Edit' ''(24 48)(24 184) ↵
      ('Attach'('Top' 'Left' 'Top' 'Right'))↵
      ('Event'('KeyPress' 'eEditKeyPress')) ⍉
[6]   'EditBox1'□WC'Edit' ''(48 0)(184 232)↵
      ('Attach'('Top' 'Left' 'Bottom' 'Right'))('HScroll' ^1)↵
      ('VScroll' ^1)('Style' 'Multi')('Event'↵
      ('KeyPress' 'eEditBoxKeyPress'))⍉
[7]   'TrackBar1'□WC'TrackBar'↵
      ('Attach'('Bottom' 'Left' 'Bottom' 'Right'))('Limits'(1 1))↵
      ('Posn'(232 0))('Size'(32 184))('TickAlign' 'Top')↵
      ('Event'('KeyPress' 'eTrackBarKeyPress')↵
      ('Scroll' 'eTrackBarEvent')('ThumbDrag' 'eTrackBarEvent'))⍉
[8]   'Push1'□WC'Button' 'Close'(240 184)(24 48) ↵
      ('Attach'('Bottom' 'Left' 'Bottom' 'Right'))('Data'(1))('Defa↵
      ult' 1)('Event'('Select' 1))▽⍉
    
```

Notice that extra spaces introduced at the front of the last line above *would* matter, whereas the spaces introduced at the front of the second last line displayed above *do not* matter - through expected APL syntax lenience.

One conceptual leap involved in object-oriented programming (OOP) relates to a new mental model of the place of a program. Evolving from a linear sequence of instructions with a few (screen) choice entry points and corresponding occasional embedded jumps (ignoring loops), OOP now conceives a model of a hierarchy of objects bristling with their own individual software programs that may each arbitrarily modify any existing objects and create with impunity other new bristling object hierarchies.

2.3.3.2 Ask for the next module on **dot syntax**. How did you get on with Module 2? Was it clear? 😊

## Module3: Dot Syntax

### § 3.1 Object References

#### §§ 3.1.1 Making References with $\Leftarrow$ and $\Leftarrow$

The APL primitive function *execute* ( $\Leftarrow$ ) has been generalised to take the name of an object as its argument and return a *reference* to that object (of notional APL *dataType* RefSc).

$RefSc \Leftarrow \Leftarrow CharVec$	$\Leftarrow$ Returns scalar reference to object, named in <i>CharVec</i>
---------------------------------------	--

This extended execute enables one to assign arbitrary names to a single GUI object or namespace.

$F \Leftarrow \Leftarrow 'F'$

This has no noticeable effect as *F* already refers to the *Form*.

$G \Leftarrow \Leftarrow 'F'$

This creates another ‘ref’ to the *Form*, previously identified by *F* but now, more or less equally, by *G*.

$Arr \Leftarrow RefSc \Leftarrow CharVec$	$\Leftarrow$ Returns result of executing <i>CharVec</i> in <i>RefSc</i>
---	---

The dyadic extended definition of *execute*, with a space to its left, means ‘execute Rarg in space Larg’.

Changes to the object of one reference will make changes to the object of all so defined equivalent references as there is in (virtual) reality only one underlying object. Thus for example,

$F \Leftarrow 'A \leftarrow 1'$   
 $\vdash G \Leftarrow 'A' \rightarrow 1$

$VAR \Leftarrow RefSc$	$\Leftarrow$ Creates new name, <i>VAR</i> , for the object referred to
------------------------	--

Assignment has been generalised to take a ref on the right and assign that ref to the name on the left. So refs can be assigned to names in the same way as variables, and their values can be accessed just like (shared) variables. Thus objects are like ‘deep variables’ with ‘shallow references’.

$J \Leftarrow I \Leftarrow H \Leftarrow G$   
 $\vdash J \Leftarrow 'A' \rightarrow 1$   
 $H \Leftarrow 'A \leftarrow 5'$   
 $\vdash J \Leftarrow 'A' \rightarrow 5$

Notice that the *Form* itself will not disappear on  $\square EX 'F'$  - not until the last ref to it has been erased.

In version 9.0 the system functions  $\square CS$ ,  $\square NQ$  and  $\square DQ$  accept namespace refs as arguments as well as quoted names (character vectors).

Objects are essentially like variables and therefore, quite naturally, user defined functions may take refs as arguments and return refs as results.

A number of other primitive APL functions have been extended to accept arguments that are references to objects, or to return references to objects as their results.

$BoolSc \Leftarrow RefSc_1 = RefSc_2$	$\Leftarrow$ Determines the absolute equality of refs
---------------------------------------	---

If  $\vdash RefSc_1 = RefSc_2 \rightarrow 1$  then *RefSc<sub>1</sub>* and *RefSc<sub>2</sub>* are two references to the same space. *Not equal* ( $\neq$ ) returns the opposite. *Match* ( $\equiv$ ) and *not match* ( $\neq$ ) have been similarly extended. (Note that  $\vdash \theta \equiv G = \theta$ .)

$BSc \Leftarrow RSc_1 \equiv RSc_2$	$\Leftarrow$ Determines the absolute identity of refs
-------------------------------------	---

3.1.1.1 Create two refs to the same object. Try varying some more or less subtle aspects of each ref (eg the  $\square AT$  of some internal function) to see if their equality and identity can be made to diverge.

In APL 1, arrays may be indexed or otherwise manipulated without giving them a name. From what we have seen so far it would appear that objects have to be given a name, but this is not the case.

$RefSc \leftarrow \square NS()$

$\alpha$  Return a ref,  $RefSc$ , to an **unnamed namespace**

Note  $\vdash () \equiv \emptyset$

A (pseudo-niladic) call to  $\square NS$  returns a reference to an ‘unnamed’ namespace that can be manipulated just like a ‘named’ object. This is such a natural candidate for a niladic primitive function that John Scholes has suggested that it be given a special symbol, @, which one might call *anon*.

$BSc \leftarrow \tilde{n}_1 \cong \tilde{n}_2$

Whether namespaces are isomorphic

Isomorphism of spaces ( $\cong$ ) implies ‘topological’ or operational similarity but not absolute identity.

@

New anonymous namespace such that  $\vdash @ \equiv \square NS''$

In Dyalog version 11, @ in fact has become the new executable niladic system functions,  $\square THIS$ .

$BSc_1 \Rightarrow BSc_2$

Material implication of  $BSc_2$  from truth value of  $BSc_1$

$(\vdash \tilde{n}_1 \equiv \tilde{n}_2) \Rightarrow \vdash \tilde{n}_1 \cong \tilde{n}_2$

The truth table of *implies* ( $\Rightarrow$ ) may be defined as

$BSc_1 \Rightarrow BSc_2 \hookrightarrow \sim BSc_1 \wedge \sim BSc_2$

The most basic implication is that ‘if the *truth-value* of  $BSc_1$  is true then  $BSc_2$  is also true.’

Logical Aside:  $P \Rightarrow Q, P \therefore Q$  is a *valid argument* (Modus Ponens)  
 $P \Rightarrow Q, \sim Q \therefore \sim P$  is a *valid argument* (Modus Tollens)  
 $P \Rightarrow Q, Q \Rightarrow R \therefore P \Rightarrow R$  is a *valid argument* (Hypothetical Syllogism)

3.1.1.2 Compare the deep similarity (isomorphism) of spaces  $A$  and  $B$  with the deeper identity of spaces  $C$  and  $D$ .

$A \leftarrow \square NS'' \diamond B \leftarrow \square NS''$   
 $C \leftarrow D \leftarrow \square NS''$

Although these objects have much in common with ordinary (APL 1 & 2) variables, the system function  $)VARS$  does not report the names of global objects, and the name class of (scalar) objects is not 2, but 9. Instead,  $)OBS$  reports the names of global objects and  $\square NL$  9 returns a matrix of all ‘visible’ global and local objects.

### §§ 3.1.2 Parent.Child Hierarchy

#

$\alpha$  Returns a ref to the Root space

The display name of the Root space is the one element vector  $(, '#')$ , thus  $\vdash \# \hookrightarrow , '#'$ . Similarly,  $\vdash \square SE \hookrightarrow ' \square SE'$  and furthermore  $\vdash \# \equiv \#'$  and  $\vdash \square SE \equiv \#'$   $\square SE'$

)NS

Displays the name of the current namespace

If you change space to the Root space and hit  $)NS$  then you will be told that you are in the Root space #. # is a direct object reference to the Root space, and  $\square SE$  is a direct object reference to the Session space. Every Dyalog primitive GUI object that you can create can trace its roots to # (or  $\square SE$ ).

**If we create a *Form* object in # called *FRM* then this object can also be referred to as #.*FRM*.** This is the beginning of *Dot Syntax* in Dyalog APL. Objects that are children of the Root can have #. prepended to their name without significant repercussions. Thus objects may be referenced hierarchically.

$CVec \leftarrow \# RefSc$

$\alpha$  Returns the display form of  $RefSc$

In the above example we have  $\vdash \# FRM \hookrightarrow ' \# . FRM'$ . In version 11,  $\square DF$  can modify the display form.

The *Parent.Child* relationship is valid at all levels. If our *Form* had a child *Button* called *BTN* then this *Button* may be identified while in *#* by the syntax *FRM.BTN*, relating parent and child, surname first.

$\tilde{n}_3 \leftarrow \tilde{n}_1 . \tilde{n}_2$	⌘ Returns a direct reference to subspace $\tilde{n}_2$
--	--

By means of dot syntax, objects may be referred to in a hierarchical fashion. Dot syntax describes object ancestry. If  $\tilde{n}_2$  is a direct descendent (child) of  $\tilde{n}_1$  then  $\tilde{n}_1 . \tilde{n}_2$  returns a reference to  $\tilde{n}_2$  from a space containing space  $\tilde{n}_1$ . The notation  $\tilde{n}_1$  is used to represent the name of an argument of *dataType RefSc*.

3.1.2.1 Create a ref to an unnamed child of an unnamed namespace.

$VecCVec \leftarrow \square WN CVec$	⌘ Returns the name of each child object of <i>CVec</i>
--------------------------------------	--

This system function returns the names of all objects whose parent's name is given in *CVec*.

In a clear workspace, given

```
'FRM' ⌘ WC 'Form'
'FRM.BTN' ⌘ WC 'Button'
```

then

```
⌈(⌘ WN '#') ≡ , c 'FRM' ↪ 1 and ⌈(⌘ WN 'FRM') ≡ , c 'FRM.BTN' ↪ 1
```

Also

```
⌘ wn 'se'
⌘ SE.cbtop ⌘ SE.cbbot ⌘ SE.mb ⌘ SE.popup ⌘ SE.tip
```

Consider John, also known as John Scholes, whose name is now to be written as Scholes.John to avoid any confusion with Daintree.John. In other words, prepending (rather than appending) the ancestral name identifies more specifically that John in question.

### §§ 3.1.3 Object.Object. .. Object.Object Rationale

Dot syntax can be used repeatedly to reference objects deep inside an object hierarchy. If, for example, a *Form F* has a child *Group G* which itself has a child *Edit E* and if *F.G.E* is called while **execution is inside the parent** of *F* then the result will be a direct relative reference to the *Edit* object. If *F* is a child of *#* then *#.F.G.E* will return an absolute reference to *E* when called in any space.

An address label written as *Country.City.Area.Road.Number.Surname.Forename* might serve as a useful model of a dotted hierarchy. A more precise analogy might be DOS (or UNIX) directories wherein C: is like the Root *#* and symbol \ (or /) is analogous to a dot.

Unfortunately the *dot* in dot syntax formally does not play the role of any regular APL syntactic element. In the current context, where dot has a namespace on either side and returns a namespace, the *dot* looks like a function. But parsing function expressions from right to left implies that *#.F.G.E* is equivalent to *#.F.(G.E)* and *G* is not necessarily visible from the current space and may give a **VALUE ERROR**.

However, interpreting *dot* as a dualistic **operator** with namespace operands and derived result is more consistent with APL uses of dot and with the required order of execution. Parsing operator expressions proceeds from left to right implying that *#.F.G.E* is equivalent to *((#.F).G).E* as required.

3.1.3.1 Examine the display forms of the derived functions

```
+ × o + × o ÷ ↪ + × o + × o ÷
- . + . × . ÷ ↪ - . + . × . ÷
```

Experiment with the effect of parentheses in these expressions and in other similar expressions in order to exhibit various alternative roles of the operators.

3.1.3.2 Write an operator such as

```

∇ r←a(f ∘ g)b
[1]   r←a f g b ∇
    
```

and a set of functions such as

```

∇ r←{a}f1 b
[1]   :If 0=⊞NC'a'
[2]       r←÷b
[3]   :Else
[4]       r←a÷b
[5]   :End ∇
    
```

then trace the order of execution of various expressions such as

```

3 (f1 ∘ f2) ∘ f3 ∘ f4 ∘ f5 4 ↵ 0.75
    
```

Try to force syntax errors. Note any interesting conclusions.

## § 3.2 Direct Property Access

### §§ 3.2.1 Object.Variable Syntax

The value of a variable may be accessed or assigned by name from outside a namespace.

<code>Arr←ñ.VAR</code>	↵ Read variable named <i>VAR</i> inside visible space <i>ñ</i>
------------------------	--

<code>ñ.VAR←Arr</code>	↵ Write variable named <i>VAR</i> inside visible space <i>ñ</i>
------------------------	---

Namespaces can contain variables. Dot syntax extends to variable names to the right of a dot. This facilitates direct access to variables in other spaces.

GUI objects are essentially namespaces containing predefined properties etc .. , and properties are essentially variables. Therefore the above syntax should and does apply to objects and their properties.

3.2.1.1 Access the *Caption* of the *Form F* directly from the Root, where

```

'#.F'⊞WC'Form' 'This is It'
    
```

Note that `⊞WX` must be set to 1 in space *F*. The default value of `⊞WX` in a clear WS is determined by the value of the registry parameter **default\_WX**. This can be changed in the registry using **REGEDIT.EXE** at location **HKEY\_CURRENT\_USER\Software\Dyadic\Dyalog APL/W 9.0** or directly through the APL Session in [Options][Object Syntax][Expose GUI Properties].

3.2.1.2 Create a *Calendar CAL* on a *Form F* and experiment with properties such as

```

F.CAL.CircleToday←0
F.CAL.CalendarCols←?6ρ<3ρ255
F.C.MinDate←38717 ↵ 2006 1 1
    
```

3.2.1.3 Create a *RichEdit RE* on a *Form F* and experiment with properties such as

```

F.RE.SelText←(1 1)(1 20)
F.RE.CharFormat[1]←<'Italic'
F.RE.CharFormat[5]←50 ↵ Superscript
F.RE.PageWidth←5×1440
F.RE.ParaFormat[1]←<'Centre'
F.RE.ParaFormat[3 5]←288 1
    
```

### §§ 3.2.2 Object.Object. .. Object.Property Rationale

3.2.2.1 Create a namespace  $\# . A . B . C . D$  containing a variable  $V$  with the value  $1$ . Access this variable from each of the spaces  $\#, A, B, C$  and  $D$ .

The analogy with DOS directories can be extended to files in directories. **File** names at the end of a directory string are like **variable** names at the end of a namespace string.

3.2.2.2 Create a *Menu* on a *MenuBar* on a *Form* and set the *Caption* and *Active* properties of the *Menu* directly from the *Form* space.

Since objects are essentially APL variables, the rationale behind multiply dotted expressions ending with a variable name is exactly the same as that for a similar expression ending with a namespace.

### §§ 3.2.3 Using Object.Object. .. Object.Property Constructions

Object construction, as used in Visual Basic, can now be adopted, almost in totality, by APL. This makes translation from VB often very straightforward. In particular, macros recorded in Microsoft Office products can be viewed in VBA (via **Alt+F11**) and generally can be translated easily into APL. This, as we shall see later, is a powerful way to transcribe Office VBA into OLE programs in APL.

3.2.3.1 Rewrite the function  $\nabla \text{MAKE\_Form} \nabla$  in §§ 2.3.3 setting as many properties as possible using direct assignment from the Root (or by any preferred compromise with or without  $\square W \dots$ ).

## § 3.3 Direct Method Invocation

Like much of the Dyalog APL implementation of GUI concepts (such as the use of name-value pairs), *Dot Syntax* is imported from mainstream GUI-oriented computer languages such as C#, Visual Basic and Java. Dot Syntax is a shorthand notation that can be used to specify the properties of any object or to call any method on any object without explicitly having to be (executing code) inside the object space. One difference worth noting, that we emphasise later, is that the object hierarchy in APL is more literal than that found in VB, which is more notional.

### §§ 3.3.1 Object.Function Syntax

$$f \leftarrow \tilde{n} . g$$

$\rho$   $f$  refers to function  $g$  in space  $\tilde{n}$

Namespaces can contain functions. Dot syntax extends to function names on the right of the dot, allowing immediate access to functions in different spaces. The functions may be niladic or ambivalent and their arguments are found in the correct places for *dot* to be interpreted (informally) as an operator.

In this case the dot has a niladic, monadic or dyadic function  $g$  on its right and a space  $\tilde{n}$  on its left and returns a function  $f$  – essentially a call to function  $g$  from the current space from which  $\tilde{n}$  must be visible. Here the dot looks more like the familiar primitive *inner product* dot operator that takes a dyadic function to left and right and returns a derived dyadic function  $(f_2 \leftarrow \tilde{h}_2 . g_2)$ .

Again the space on the left,  $\tilde{n}$ , can be replaced with a dotted string referring to any arbitrary subspace. Some justification for this extension has been given above.

3.3.1.1 Run a function defined in one namespace from another namespace, paying particular attention to the values of local and global variables and to the spaces in which sub-functions are actually executed.

### §§ 3.3.2 Object.Object. .. Object.Function Rationale

Primitive as well as user-defined variables and functions succumb to dot syntax.

```
F.⊖IO←0
F.ι 9↪0 1 2 3 4 5 6 7 8
```

In keeping with rich choice of algorithms that APL frequently affords, this introduces more choice. *eg*

```
F.⊘'Caption'↪F.Caption
'F'⊘'Caption'↪F.Caption
F⊘'Caption'↪F.Caption
```

All these expressions give same result, with natural extension of dot syntax to primitive functions  $\oplus$  and  $\iota$ , and a natural generalisation of *dyadic execute* already alluded to. In the case of a scalar ref, *dyadic execute* gives the same result as  $Arr\leftarrow RSc.\oplus CVec$ , *ie*  $RSc.\oplus CVec\leftarrow RSc\oplus CVec$

The following three statements all have the same effect:  $\square EX'F.MB.M'$  or  $F.\square EX'MB.M'$  or  $F.MB.\square EX'M'$ .  $\square WN$  may also take a dotted character vector argument and return a dotted result:

```
(⊖WN'F.MB')↪,c'F.MB.M'
```

In Visual Basic, you can use the dot syntax to access properties and invoke methods. For example: **Application.Workbooks.Add()** calls method **Add** (with no arguments) from the collection object returned by the **Workbooks** property of ... Note that the value of a property may be an object.

Dyalog APL dot syntax for functions extends to (niladic and monadic) methods, as in VB. *eg*

```
F.Close
F.CAL.KeyPress''⊣ρc'RC'
F.CAL.Size←3 4×F.CAL.GetMinSize
F.CAL.MouseDown 77 13 1 0 ⊖ F.CAL.MouseUp 77 13 1 0
F.RE.GotFocus ⊖ ⊘ F.RE.GotFocus()
F.RE.RTFPrint ⊖ ⊘ F.RE.RTFPrint()
F.RE.RTFPrintSetup ⊖
F.RE.RTFPrint F.RE.RTFPrintSetup'Selection'
#.GetEnvironment'MaxWS'
F.CAL.SelDate←#.DateToIDN(2003 12 1)
```

The last two examples invoke methods on the Root. Root methods and properties are exposed according to the setting of [Options][Object Syntax][Expose Root Properties] which, by default, depends on the value of the registry entry **PropertyExposeRoot** and not on the value of  $\square WX$ .

### §§ 3.3.3 Defined Operators in Object Space

3.3.3.1 What is the interpretation of  $\#.+.#\times$ ? Could you space-qualify the inner product operator?

Dot (.) is not a token with a strict interpretation as a rational APL syntactic element - a variable, function or normal operator. To see this clearly, consider dot syntax as applied to user-defined operators.

Given an operator  $\hat{O}_1$  in space  $\#.A.B$ , this operator can be referenced from any point in the code by the notation  $\#.A.B.\hat{O}_1$ . If dot is to be interpreted as an operator, then this dotted list of tokens involves a dualistic operator adjacent to another operator, which is a situation that would set a new precedent in APL grammar.

The complexities of interpretation are not helped by the fact that dot is already used in APL in at least two other different places. It is a neutral symbol used to represent the decimal point. (Perhaps in a later version

of APL this symbol will be supplied by the decimal symbol in [Control Panel][Windows Regional and Language Options].) Dot is also used for the primitive *inner* (.) and (irrational) *outer* (◦.) *product* operators. Normally, once a symbol or token has been used to represent an operator then it must always represent an operator (see APL Linguistics in *Vector Vol. 2 No. 2 p118* for some discussion of this). Therefore the dot in dot-syntax should be assumed to be an operator, given no other evidence to the contrary. (Remember, however, that the *reduce* operator (/) and the *replicate* function (/) unfortunately exemplify such a contradiction, albeit tolerated. Can you think of the other principal case?)

As a rule of thumb, the meaning of a dot (the big dot • here) may be (partially) interpreted by the class of the token to its left. In the special 'extra-APL' case of a decimal point, the symbol immediately to the left clearly must be a numeric digit or space. The interpretation of the class 2 case is outlined in Module 11. As we have seen above, the class 9 case includes various classifications of right 'operand'.

Class of Larg of •	Syntax (Grammar)	Semantics (Meaning)
(0)	<i>D</i> •...	<b>Decimal number &lt; 10</b>
(0)	<i>D</i> •• <i>D</i> •...	<b>Decimal number</b>
(0)	• <i>D</i> •...	<b>Decimal &lt; 1</b>
1		Label
<b>2</b>	•• <i>RefArr</i> •...	<b>See Module11</b>
2.1		Variable
2.2		Field
2.3		Property
<b>3</b>	•• <i>f</i> •...	<b>Inner Product</b>
(3)	••◦•...	<b>Outer Product</b>
3.1		Canonical Function
3.2		Dynamic Function
3.3		Derived Function
3.6		External Functions & Methods
4.1		Canonical Operator
4.2		Dynamic Operator
(9)	•• <i>Ref</i> •...	<b>Dot syntax</b>
9.1		Namespaces
9.2		GUI Object
9.3		Instances of Classes
9.4		Classes
9.5		Interfaces
9.6		.NET Classes?

3.3.3.2 What conclusions about the tokens involved can you draw from the syntax of the statement `#.A.B[K]` or `#.A5.B` or `#.A .5` or `B.0` or `#.5`?

3.3.3.3 Ask for the next module on the **Session Object**. How did you get on with Module 3? Was it easy to follow? 😊

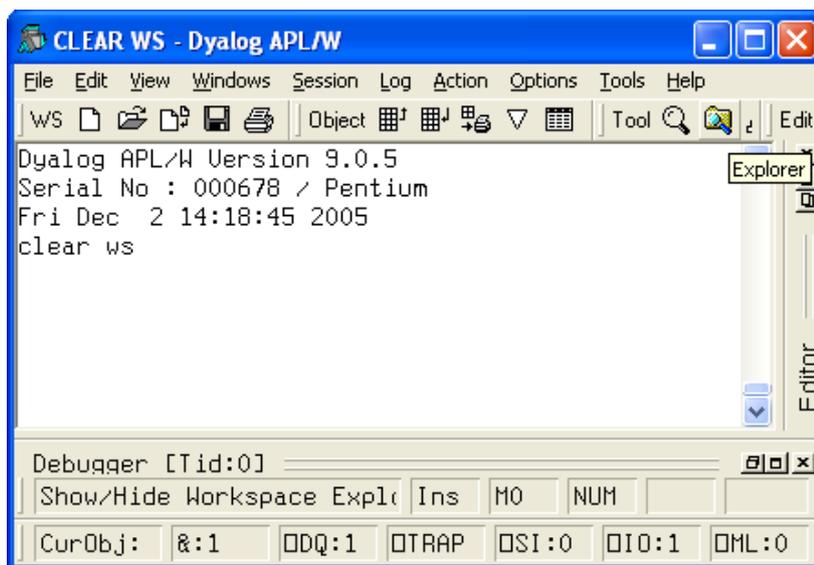
## Module4: The Session Object

There is only one object of *Type Root*, called #, and there is only one object of *Type Session*, called `SE`. `SE` itself has properties, events and methods, and can support certain types of children, just like other objects. In the development version of Dyalog the child content of `SE` is loaded from a DSE file when APL starts. In the runtime version `SE` is empty of children but still retains its own innate characteristics. First we shall approach the Session object from the point of view of a **user** of the development environment. Then we shall explore its contents from the point of view of an object-oriented programmer.

### § 4.1 Using the Session Object

#### §§ 4.1.1 Immediate Execution Mode of `SE`

An *APL session* is, traditionally, the environment wherein one writes and executes APL code. In APL 1 and APL 2 eras of **Timesharing**, the session was available for the period of time during which ones terminal was connected and signed on to a mainframe computer running APL. Now, in APL 3 and APL 4 eras of **Windows**, the session is the window in which one executes APL expressions and defines APL functions. (See *Vector Vol.3 No.3 p97* for mention of an early graphic vision of APL 3.)



The window is live. Expressions are, as ever, executed as soon as the **Enter** key is pressed. (**Enter** initiates execution of the code on the line inhabited by the cursor.) Immediate execution now extends to GUI objects. Objects are displayed as soon as they are created. Objects respond to mouse and keyboard when created in the development environment in the same way as they would in the runtime environment under `DQ`. This takes the Dyalog APL Session to a new level of interactivity as regards the modern GUI nature of immediate execution.

#### §§ 4.1.2 Tracer and Editor of `SE`

In the days of timesharing, one of the most important keystrokes to learn in APL was how to stop execution. Every second of execution time cost real money, so you had to learn **Ctrl+C** early in your career. Now, on a PC, runaway code is less frightening so **Ctrl+Break** is, perhaps, less important than:

- **Ctrl+Enter** initiates step by step execution of the code on the current line
- **Shift+Enter** opens for editing the program located under the cursor (or the suspended function when the cursor is in column 1)
- **Ctrl+Shift+Enter** jumps over the current line in the tracer (*cf* **Ctrl+Shift+Backspace**)

Other significant keys, such as **Esc**, are all listed in the current .DIN input file whose name and location may be discovered from the result of

```
#.GetEnvironment''aplkeys' 'aplk'
```

4.1.2.1 Investigate, using Notepad.exe, the content of your current .DIN file. Use the function *KEYPRESS* in supplied workspace ..\WS\UTIL.DWS to investigate your keyboard further.

The GUI implementation of the APL Session ought to follow, as far as is reasonable, the general Windows standards. Compare Microsoft's *Application Development Guide* with details in *Dyalog APL User Guide*.

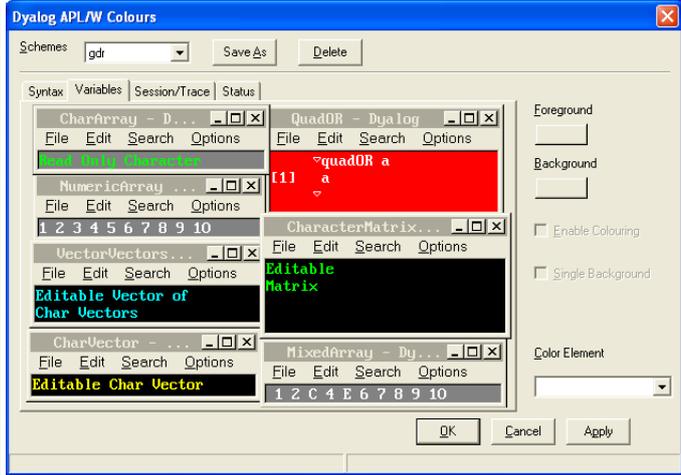
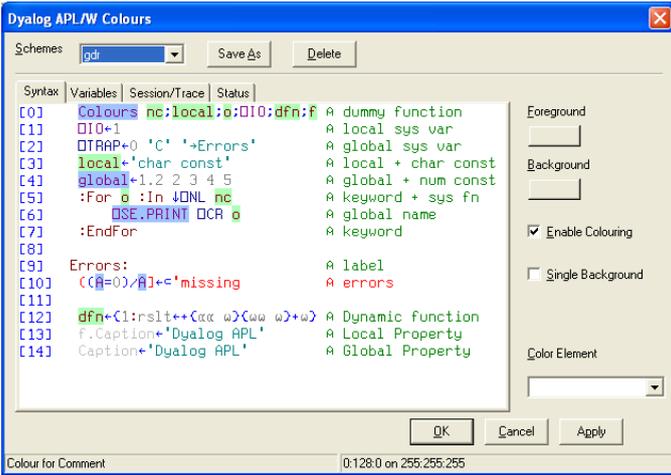
### §§ 4.1.3 Choosing syntax Colours

Syntax colouring is a valuable aid to reading and writing APL programs. The choice of colours should make some sort of sense to the programmer, viz you. It is particularly important to choose nice meaningful colours for global and local names. There are many possible criteria. Very pale background colours for some of the elements seem to work well.

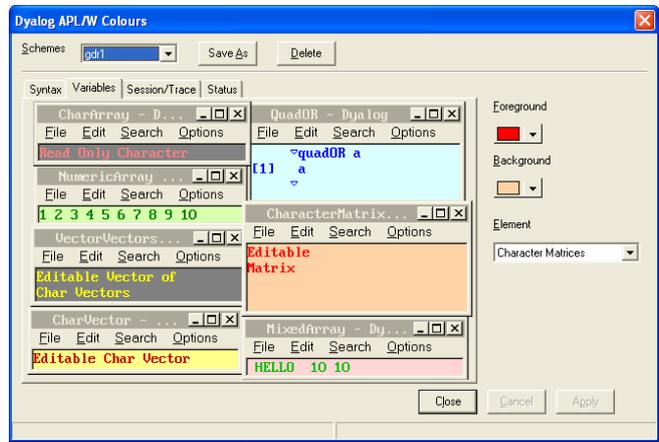
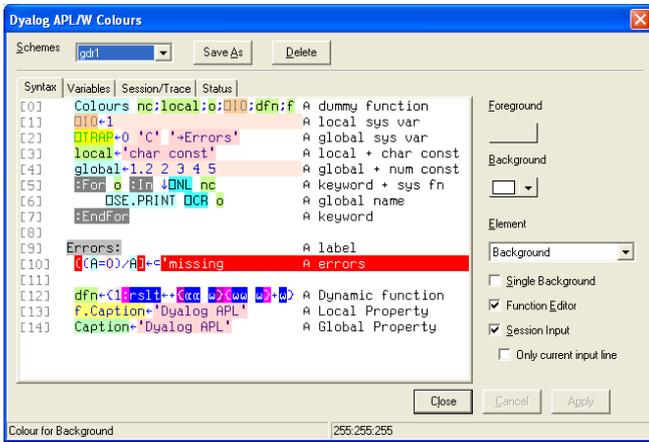
4.1.3.1 How would *you* colour the various syntactic elements in APL?

Below is a suggested scheme whose philosophy rests on a proposed correlation between name class ( $\square NC$ ) and approximate frequency of the colour. Elements of class 1, 2, 3 and 4 are aligned with the spectrum of colours: Black Red Orange Yellow Green Cyan Indigo Violet White, or any intermediate colours/intensities. (For this purpose name class 9 is regarded as name class 2.) Unclassified APL symbols may be placed in any suitable position in the 'class spectrum' from labels (1) to operators (4).

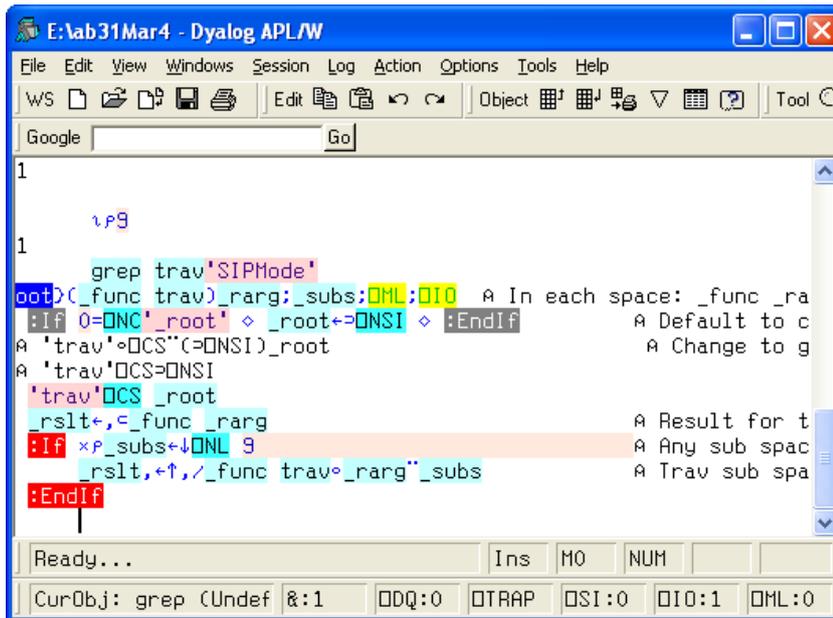
Class	Syntax Colour	Syntax Element
0	Gray (192)	Line Numbers
1	Black	Labels
1	Dark Red	Control Structures
	Red	Error - Unmatched parentheses, quotes, and braces
2.1	Orange	Character constants
2.1	Dark Yellow	Numeric constants
2	Black on Pale Red	Localised System Variables
2	Red on Black	Global System Variables
2.4	Black on Pale Yellow	Local GUI Property
2.4	Yellow on Gray	Global GUI Property
2∨3	Black on Pale Green	Local Names (functions and esp <sup>ly</sup> variables)
0	Dark Green	Comments
3∨2	Black on Pale Blue	Global Names (variables and esp <sup>ly</sup> functions)
3	Black on Pale Cyan	System functions
3.2	White on Light Indigo	D-Fn name
3∨4	Indigo	Primitives (functions/ops/special...)
4.2	White on Light Violet	D-Op (monadic)
4.2	Violet	D-Op (dyadic)
	White	Background



OR



The main Session window can also be coloured. See [Options][Colours...][Session/Trace] tab.



APL code for the Google bar appearing in the session above was kindly given free to [dyalogusers@yahoogroups.com](mailto:dyalogusers@yahoogroups.com) by Norbert Jurkiewicz and is discussed a little in Module 8.

## § 4.2 Inside the Session Object

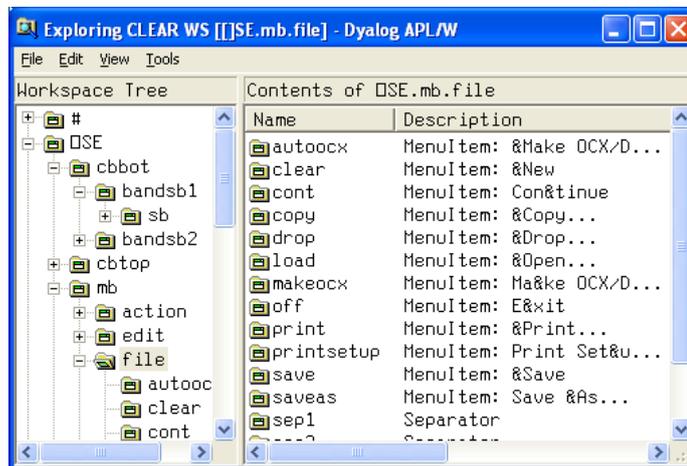
### §§ 4.2.1 Exploring the Workspace and `□SE`

The `□SE` window appears to be built from something like a multi-line `Edit` object or a `RichEdit` object just as the `Grid` object appears to be built from a matrix of `Label` or `Edit` or `Combo` .. objects. Normally we can only explore these superobjects to the point allowed by the object programmer who decides (as we shall see later) which properties, which methods and which events to *expose* to the outside world. These lists can be found from the `PropList`, `MethodList` and `EventList` properties of `□SE`. (Note that [Options][Object Syntax][Expose Session Properties], or the related registry parameter `PropertyExposeSE`, has to be set in order to expose the GUI names of the Session object.)

4.2.1.1 Change into the Session space and examine its properties. Assign `Posn`.

### §§ 4.2.2 Examining Session Menus and Buttons

The quickest way to gain an overview of the contents of `□SE` is by way of the Workspace Explorer, to be found in [Tools][Explorer...] or via the Explorer button, 'tipped' on the session image in §§ 4.1.1



4.2.2.1 Use the Workspace Explorer to investigate the object hierarchy of the Session object.

4.2.2.2 In the explorer, use right click [Set Session Space] to change into the space of the Explorer `ToolButton Bitmap`, `□SE.cbtop.bandtb3.tb.explorer.bm`, and display the value of the `Bits` and `CMap` properties.

4.2.2.3 Look at the value of the `Event` property of the `MenuItems` in `Menu` space `□SE.mb.file`. The `Select` event of most `MenuItems` in the Session has a bracketed keyword that performs some specific function outside of APL itself. These calls are specially crafted for the Session object although you can make use of some of them in your own applications if they are appropriate.

4.2.2.4 What happens if you change the `Event` property of the `□SE.mb.file.load MenuItem` from

```
Event ← ,c'onSelect' '[ChooseColors]'
```

to

```
Event ← 0 0
```

4.2.2.5 Replace the `Tip` on the `wsload ToolButton` with the word 'Open', and the `Hint` on the `load MenuItem` to 'Opens workspace file'.

## §§ 4.2.3 Miscellaneous Properties and Methods

Some properties *have* obvious meanings. (Can you see the verb *have* here as a scalar dyadic function with no result?☺)

4.2.3.1 Try changing `⊞SE.State` or `⊞SE.Size`.

Other properties have a more session-specific rôle. For example, the `CurObj` property can be useful in session-related functionality such as the `ToolButton` version of `varChar`. `⊞SE.CurObj` reports the name currently under the input cursor. Its value is reported in the session `StatusBar` and its significance may be appreciated by clicking on various names in the session window and viewing the message reported on the `StatusBar`.

The session `FontObj` property is useful, although the version 9 session is not best geared to non-fixed-width or large true-type fonts. version 11 is better, and obligingly includes a format bar in the session.

4.2.3.2 Write a function, such as that below, which loops round changing the `⊞SE.FontObj` property to each of the fonts in your `FontList` in turn. Trace `▽TestFonts▽`. Notice the effect in the session. Be prepared to close APL if the session becomes illegible (or hit Ctrl+Shift+Enter to get to line [9]).

```

    ▽ TestFonts;av;a;c
[1]   16 16ρav←⊞AV[1+⊞NXLATE 0]
[2]   a←>`#.FontList
[3]   a←a[⊞↑a]
[4]   c←1
[5]   Loop:→(c>ρa)ρEnd
[6]   ⊞SE.FontObj←c>a
[7]   c+←1
[8]   →Loop
[9]   End:⊞SE.FontObj←'dyalog std'▽
    
```

```
##
```

```
⌘ Parent of the current space
```

`##` is analogous to the use of `..` in DOS. `##` returns a ref, as can be verified from statements such as

```
P←##∘|⊞NC'P'↳9
```

4.2.3.3 Suggest a use for method `⊞SE.Create`, bearing in mind that runtime systems do have a session object, albeit initially empty.

4.2.3.4 Copy the function `▽DISPLAY▽` from workspace `..\ws\util.dws` into the session object and set `⊞PATH` to `'⊞SE'`. Why is this useful?

## § 4.3 Building the Session Object

### §§ 4.3.1 Tracing `▽BUILD_SESSION▽`

4.3.1.1 Load the supplied workspace `..\ws\buildse.dws`. Choose a country from the list

```
Trans.CODES↳'UK' 'FR' 'IT' 'FI' 'US' 'GR'
```

and type and trace `▽BUILD_SESSION▽` with a Rarg of the language key of your choice; English, French, Italian, Finnish, American or German. (This choice would normally match your innate keyboard language and the language set in [Control Panel][Regional and Language Options][Languages][Details].) Watch the component objects being built from the contents of the current (.DSE) session file (as identified in registry parameter `Session_File` or as set in [Options][Configure][Session]).

4.3.1.2 Practice tracing parts of the code and running others. Find the quickest keyboard-tracing route to some particular point in the code. (Set a `⎕STOP` on the target line by using your mouse when the cursor is in the first or second column on the left side of the edit window, first checking the editor menu items in [View].)

### §§ 4.3.2 Foreign Language Support

4.3.2.1 Build Sessions in other languages. Marvel at the changed *Menus*, *Hints* and *Tips*. Notice how standard menus and menu items make (almost blind) navigation possible. Any APL application may be made multi-lingual by the techniques employed in this workspace (see variable *Trans.STRINGS*). Note the [Session][Open] *MenuItem*, and also the [Log][Open] *MenuItem* which opens a .DLF log file that is maintained separately from the .DSE session file. The session file is, however, entangled with the contents of registry parameters that are described in the *Dyalog APL User Guide*.

Note that all the *Dyalog APL* manuals, the *User Guide*, the *Language Reference*, the *Object Reference*... are freely downloadable from [http://www.dyalog.com/documentation\\_library.htm](http://www.dyalog.com/documentation_library.htm).

### §§ 4.3.3 Adding useful Extensions

4.3.3.1 Add a new [File][Open] *MenuItem* whose purpose is simply to tie an APL component file. Assign the *onSelect* property of a *FileBox* object (with .DCF file filter) to the name of a function such as that below.

```
onSelect←'selectOpen'  
  
    ∇ selectOpen  
[1]  '##.FB'⎕WC'Filebox' 'Open'  
[2]  r←##.⎕DQ'FB'  
[3]  ⍎ Tie the file in r ∇
```

4.3.3.2 Add a *ToolButton* to the session that *DISPLAYS* the *CurObj-Value* pair in the session. (The workspace version of **varChar** makes use of `⎕SE.CurObj` in the APL development environment.)

4.3.3.3 Please ask for the next module on **Control Structures** 😊.

## Module5: Control Structures

### § 5.1 Logical Decisions and Jumps

In APL 1 and APL 2, program flow was controlled by branch ( $\rightarrow$ ) and also, sometimes, by execute ( $\downarrow$ ) or  $\square$ *SIGNAL* plus  $\square$ *TRAP*. In APL 3 program flow may be controlled by more readable control structures such as *:If*. There are 8 different types of control structures in Dyalog APL. They are defined by the control words *:If*, *:While*, *:Repeat*, *:For* and *:Select*, *:With*, *:Trap*, *:Hold*. Like line labels, these structures are only usable inside programs, and not in immediate execution mode.

#### §§ 5.1.1 The *:If* Statement

*:If* offers a simple readable alternative to a common traditional application of the branch arrow ( $\rightarrow$ ).

```
:If Prop  $\diamond$  ...  $\diamond$  :End           ⌘ If Prop is true execute ... code
```

If  $\neg Prop$ , ie  $Prop=1 \rightarrow 1$ , where *Prop* is a logical (Boolean) proposition and therefore either true(1) or false(0), then execute the code indicated by ..., otherwise end the *:If* statement.

```
:If Prop  $\diamond$  ...  $\diamond$  :Else  $\diamond$  ...  $\diamond$  :End   ⌘ If Prop is false execute second ...
```

If  $\neg Prop$ , then execute the code indicated by the first ... expression(s), otherwise therefore  $\neg Prop$ . In this case execute the second ... expression(s) before ending the *:If* statement.

```
:If Prop1  $\diamond$  ...  $\diamond$  :ElseIf Prop2  $\diamond$  ...  $\diamond$  :End
```

A *:If* statement may be embedded within another by means of the *:ElseIf* conditional.

##### 5.1.1.1 Rewrite

```
 $\downarrow$ (0= $\square$ NC'Forename')/'Forename←''''
```

in a *:If* statement.

#### §§ 5.1.2 Further truth Conditionals

```
:If Prop1  $\diamond$  :AndIf Prop2  $\diamond$  ...  $\diamond$  :AndIf PropN  $\diamond$  ...  $\diamond$  :End
```

Any number of *:AndIf* conditionals may be included after *:If* or *:Else* or *:ElseIf* conditionals.

```
:If Prop1  $\diamond$  :OrIf Prop2  $\diamond$  ...  $\diamond$  :OrIf PropN  $\diamond$  ...  $\diamond$  :End
```

Any number of *:OrIf* conditionals may be included after *:If* or *:Else* or *:ElseIf* conditional segments. But note that *:AndIfs* and *:OrIfs* may not be mixed within individual segments of code.

##### 5.1.2.1 Rewrite the function below in APL 1 (1<sup>st</sup> generation) language.

```
 $\nabla$  pass(A B C D E)
[1]   :If A
[2]   :AndIf B
[3]       'A and B'
[4]   :ElseIf C
[5]   :OrIf D
[6]   :OrIf E
[7]       'C or D or E'
[8]   :Else
[9]       'not A and not C'
```

```
[10]      :End
          ▽
```

Which notation is more legible? What other pros- and cons- can you identify?

5.1.2.2 Rewrite the expression

```
⊕(P∧Q) / '''P and Q'''
```

without using any of the symbols  $\oplus \rightarrow \wedge$  where  $P$  and  $Q$  are propositions. Replace  $(P \wedge Q)$  with another arbitrary logical expression involving logical connectives *and* ( $\wedge$ ), *or* ( $\vee$ ) and *not* ( $\sim$ ), eg  $((P \vee Q \vee R) \wedge \sim P \wedge Q)$ , and rewrite the formula in a *:If* statement control structure?

### §§ 5.1.3 The *:Select* Statement

*:Select* is another simplified alternative to branch ( $\rightarrow$ ).

```
:Select Arr ◊ :Case Arr1 ◊ ... ◊ :Case Arr2 ◊ ... ◊ ... ◊ :End   * Execute ... case
```

Execute the code in the *:Case* segment expression which satisfies  $\vdash Arr = ArrN$ . The last ... in this structure implies the possibility of more *:Case ArrX ◊ ...* code snippets.

```
:Select Arr ◊ :Case Arr1 ◊ ... ◊ :Case Arr2 ◊ ... ◊ ... ◊ :Else ◊ ... ◊ :End
```

5.1.3.1 Run the function below with various sorts of arguments, such as *WhatIs*  $\square$  *NULL* if in version 10+.

```
▽ WhatIs I
[1]      :Select I
[2]      :Case 1 ◊ 'I=1'
[3]      :Case 2 ◊ 'I=2'
[4]      :Else ◊ '((I≠1)∧(I≠2))∨(∼I=1)∧∼I=2'
[5]      :EndSelect
          ▽
```

and then replace *:Else* with a suitable *:CaseList* conditional qualifier, the precise definition of which is to be found in [Help][Language Help] or the [Dyalog APL Language Reference](#) manual.

## § 5.2 Looping Constructs

### §§ 5.2.1 The *:For* Statement

In many computer languages, a For statement provides a compact way to iterate over a range of values.

*:For* is another specific alternative to using branch ( $\rightarrow$ ).

```
:For Var :In Vec ◊ ..Var... ◊ :End   * Execute ..Var... for each Sc in ,Vec
```

In each iteration, *Var* takes the value of the next element of vector *,Vec*.

5.2.1.1 Rewrite the expression  $+ / NumVec$  in a *:For* statement, using  $\square$  *MONITOR* to compare efficiencies.

5.2.1.2 When might this looping mechanism sometimes be preferable to using operators such as each (*\*\**)?  
Hint: try tracing examples of both options.

5.2.1.3 Convert a looping statement such as

```
Loop:  ◊ ... ◊ → Loop × × i Bool
```

into a *:For* statement.

## §§ 5.2.2 Generalised `:For` Statements

The `Var` entry may be replaced by multiple variable names. In this case `Vec` is expected to be a vector of vectors and the  $N^{\text{th}}$  variable in the list of names is assigned at each iteration to the  $N^{\text{th}}$  element in the disclosed next element of the control vector.

```
:For Var1 Var2 .. :In VecNVec◊..Var1..Var2...◊:End      ⌘ Strand
```

In each iteration, `Var $N$`  takes the value of the  $N^{\text{th}}$  element of iteration subvector of the control vector. An example of a valid line in this case might be

```
:For V1 V2 V3 :In (1 2 3)(4 5 6)(..)...
```

An alternative definition is used if `:In` is replaced by `:InEach`. Again `Vec` is expected to be a vector of vectors but in this case the  $N^{\text{th}}$  variable in the list of names is assigned, at each iteration, to the next element in the  $N^{\text{th}}$  element of the control vector.

```
:For Var1 Var2 .. :InEach NVecVec ◊..Var1..Var2...◊:End  ⌘ Distribute
```

In each iteration, `Var $N$`  takes the value of the next element of vector  `$N > NVecVec$` . An example of a valid line in this case might be

```
:For V1 V2 V3 :InEach (1 4 ..)(2 5 ..)(3 6 ..)
```

In Module11 we shall see how a *collection object* may be treated as a `Vec` in a `:For` statement.

## §§ 5.2.3 `:Repeat` and `:While` Loops

```
:Repeat◊...◊:Until Prop      ⌘ Repeat execution of ... until ⊢Prop
```

This is necessarily an infinite loop unless proposition `Prop` can change from false to true in the ... process.

5.2.3.1 Write a 2-line function with the infinite loop

```
[1] :Repeat ⍺ [2] :Until 0 ⍺
```

Run the function and break the execution in a number of different ways. Now convert to a 1-line function

```
[1] :Repeat ◊ :Until 0
```

Try to break this loop. Be prepared to close APL. Repeat the experiment simply with `[1] →1`. Avoid such tight loops.

```
:While Prop◊...◊:End      ⌘ Execute ... while ⊢Prop
```

5.2.3.2 Loop round, executing some code, while proposition `Prop` is true (1), or `:Until Prop2` is true.

Note `:AndIf` or `:OrIf` may be included in the structure logic of `:While` and following `:Repeat`.

## § 5.3 Digging

### §§ 5.3.1 The `:With` Statement

`:With` is an alternative form to `□CS`.

```
:With Obj◊...◊:End      ⌘ Execute ... within object Obj
```

*Obj* may be the name of an object (string) or an object reference (value). The lines of code in ... are executed inside the space of *Obj*. The effect of *:With* is similar to that of `□CS`. Local variables in the outer space continue to be visible.

### 5.3.1.1 Write a function like

```

    ▽ drill
[1]     :With □SE
[2]         :With cbbot
[3]             :With bandsb1
[4]                 Dockable
[5]                     :End
[6]             :End
[7]     :End
    ▽
    
```

to drill into `□SE.cbttop.ilh.bm` and display the *Type* property at each level.

Within the Dyalog function editor, [Edit][Reformat] indents control structures and substructures according to the settings in [Options][Configure][Trace/Edit]. As in the case of the *:For* statement, *:With* extends to Collections, as described in Module11. *:With* also extends to unnamed namespaces, as described in Module11 too.

## §§ 5.3.2 Digging into SubSpaces

In an APL program one is usually working with local variables and functions all in the same space. It would therefore be tedious to prefix all names with the full space-qualified name, especially for deeply nested spaces. As we shall see when looking at *OLEClient* objects in Module7, the *:With* control structure plays an important role in identifying and changing the current space in a program.

## §§ 5.3.3 :Trap versus □TRAP

*:Trap* is a simplified version of `□TRAP`.

```

    :Trap ENum◇...◇:End           ⍝ Trap error(s) ENum and execute ...
    
```

When running code ... , in the event of an error having an error number which is in the list *ENum*, no default error action is taken and execution is passed to code after the end of the *:Trap* control structure, if there is any. This is similar to the action of something like `□TRAP←ENum 'C' '→1+□LC'`.

```

    :Trap ENum◇...◇:Else◇...◇:End ⍝ On error in first ..., do second ...
    
```

If an error of type *ENum* occurs in the first ... segment, then pass execution to the second segment and do not report the error. Further, disable the error trapping prior to processing the second segment.

```

    :Trap ENum◇...◇:Case ENum1◇...◇:Case ENum2◇...◇...◇:End ⍝ Split cases
    
```

The last ... in this structure implies the possibility of more *:Case ENumX◇...* code snippets. Furthermore, as in the *:Select* control structure, *:CaseList* and *:Else* segments may be used here too.

For a summary of the *:Hold* statement, see multi-threading in Module13.

### 5.3.3.1 Ask for the next module on OLE Servers 😊.

## Module6: In-Process OLE Servers

### § 6.1 Creating an OLE Server in a DLL

#### §§ 6.1.1 The *OLEServer* Object

Object Linking and Embedding (OLE), released in 1991, was the result of the evolution of Dynamic Data Exchange (DDE) that Microsoft developed for early versions of Windows. DDE was implemented in an early version of Dyalog APL through shared variable syntax. OLE is more powerful than DDE and can handle compound documents containing text, graphics, video, and sound.

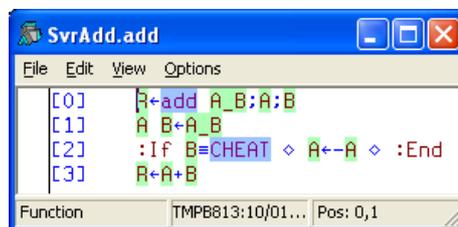
This new Microsoft platform for software components evolved to become an architecture known as the Component Object Model (COM), and later Distributed COM (DCOM). COM has been called, "*The centre of the Microsoft universe*" although the name was not emphasized until 1997. Introduced in 1993, COM is used to enable interprocess communication and dynamic object creation via any programming language that supports the technology. The term COM is often used in the software development world as an umbrella term that encompasses OLE, OLE Automation, ActiveX, COM+ and DCOM technologies.

In particular, OLE allows objects from one application to be embedded within another.

6.1.1.1 In a clear ws, create an object called *SvrAdd* of *Type OLEServer*. Rename the ws to *SvrAdd*.

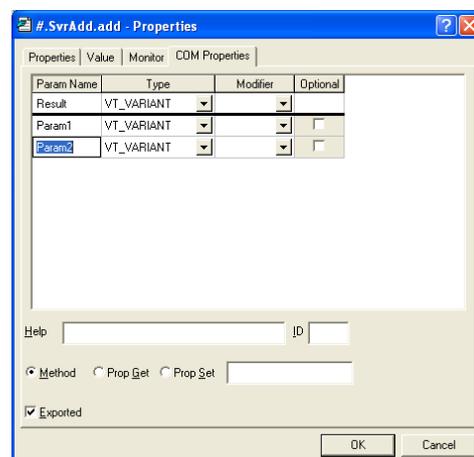
#### §§ 6.1.2 Exporting Variables and Functions as Properties and Methods

6.1.2.1 In space *SvrAdd*, write a monadic, result-returning function, *add*, such as that below, or, perhaps, a more meaningful example.



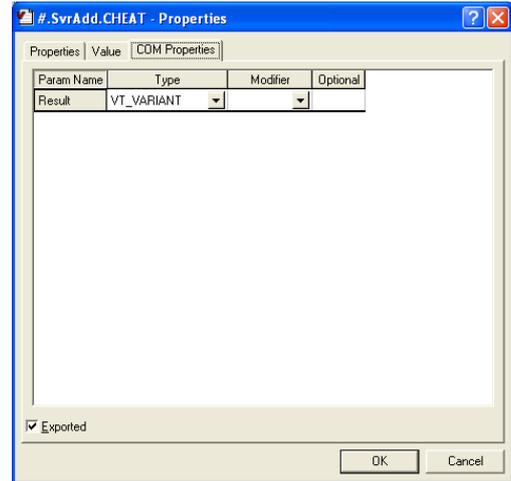
```
[0] ←add A_B:A:B
[1] A B←A B
[2] :If B=CHEAT ◇ A←-A ◇ :End
[3] A←A+B
```

Right click on the function *add* and select [Properties][COM Properties]. Check the Exported check box then click on Param1 and hit the down arrow key. Set all types to VT\_VARIANT, or any more specific type. The properties box should then look like that in the adjacent property page:



The Method radio button should be checked by default. Select OK to set the function argument and result type information as defined in your property page.

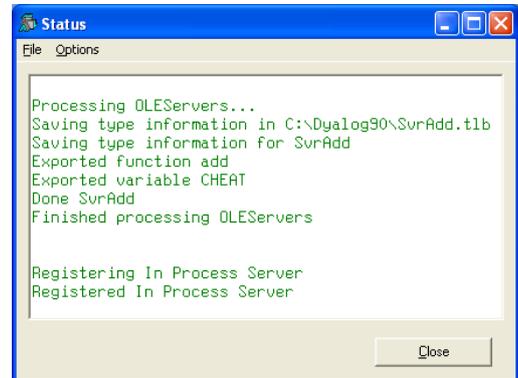
6.1.2.2 Create a variable called *CHEAT* with value zilde (⊖). Right click on the variable name and select [Properties] [COM Properties]. Check the Exported check box. Change the type of the result to VT\_VARIANT and hit OK to set the variable type information.



### §§ 6.1.3 Saving and registering your *OLEServer*

6.1.3.1 Check the menu item [File][Make OCX/DLL on )SAVE] (in version 9) and save the workspace. This returns a status information box if [Tools][AutoStatus] is checked.

At this point, Dyalog APL automatically updates the Windows registry and type libraries with all of the information needed to make your object accessible via COM. SvrAdd.TLB should be stored in a suitable directory.



Note that [File][Make OCX/DLL on )SAVE] has been replaced with [File][Export... ] in versions 10&11. (From version 10, in-process OLE Servers and ActiveX Controls may be bound with either the development or runtime Dyalog APL DLL.)

## § 6.2 Variable type Information

### §§ 6.2.1 Setting Method Information

Assuming `WX←1`, the above steps for exporting functions as methods could have been achieved under program control by

```
ExportedFns←←'add'
SetFnInfo(←'add'),←(' ' 'VT_VARIANT')←
('Param1' 'VT_VARIANT')('Param2' 'VT_VARIANT')⌘
```

Note that the names of optional parameters are surrounded by brackets, eg ('[Param2]')...

### §§ 6.2.2 Setting Property Information

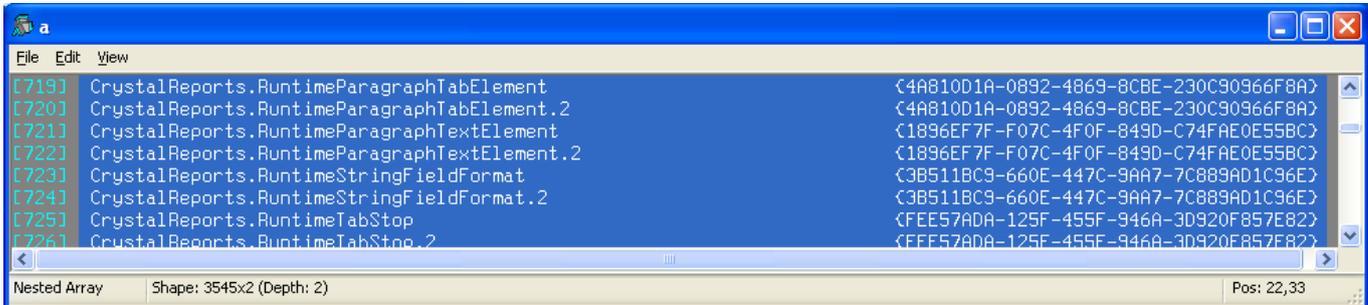
Similarly, variables can be exported as properties under program control.

```
CHEAT←⊖
ExportedVars←←'CHEAT'
SetVarInfo'CHEAT' 'VT_VARIANT'
```

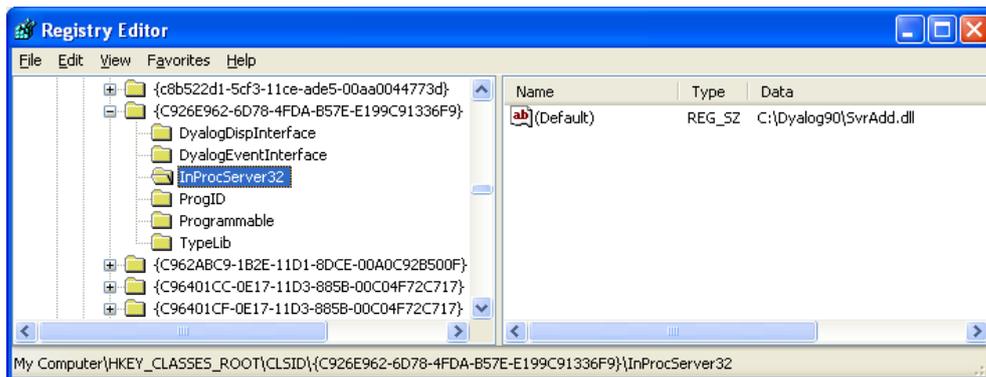
### §§ 6.2.3 Exploring the Registry Entry

The Root has a property called *OLEServers* which lists the OLE Servers registered on your personal computer.

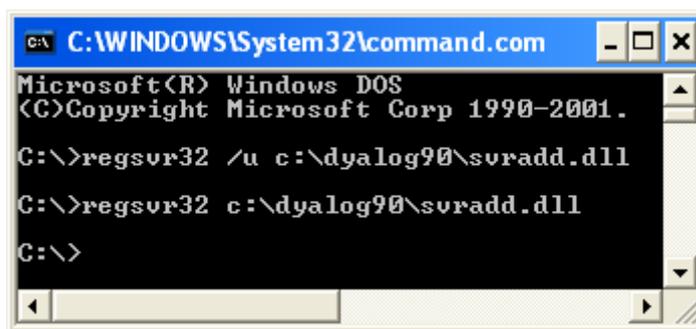
```
a←↑#.OLEServers
```



This list gives the unique class ID of each server. Looking in the registry at the classes under key HKEY\_CLASSES\_ROOT\CLSID reveals, amongst other things, the name of the DLL in which your server is stored.



The server may be unregistered and reregistered in a DOS box as required.



Note that Svradd.dll calls the dynamic link library version of Dyalog APL when it starts and Dyalog.dll must be visible to REGSVR32.EXE when the server is registered. In Dyalog version 10.0, DYALOG.DLL may be distributed free of charge as part of an application.

## § 6.3 Using your OLE Server

### §§ 6.3.1 The *OLEClient* Object

In a clear workspace, create an *OLEClient* object with the *ClassName* property set to dyalog.SvrAdd.

```
[CS 'SVR'[WC'OLEClient' 'dyalog.SvrAdd'
```

The object contains no functions or variables but *add* is reported as a method and *CHEAT* is on the list of properties.

```

) fns
) vars
) methods
add

```

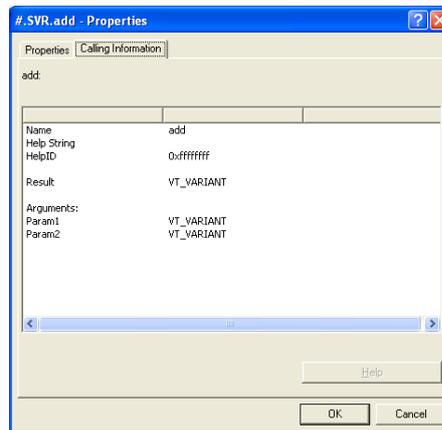
6.3.1.1 Try out your exported server method using various arguments. Set *CHEAT* to 5 and try *add* again.

6.3.1.2 Check the results of *GetMethodInfo 'add'* and *GetPropertyInfo 'CHEAT'*.

6.3.1.3 Investigate the workspaces CFILES.DWS and LOAN.DWS in conjunction with the explanations in the [Dyalog APL Interface Guide](#) and related sections of [www.dyalog.com](http://www.dyalog.com)

### §§ 6.3.2 Examining Type Libraries

The *calling information* regarding exported methods may be found by right-clicking on *add* and selecting [Properties][Calling Information].

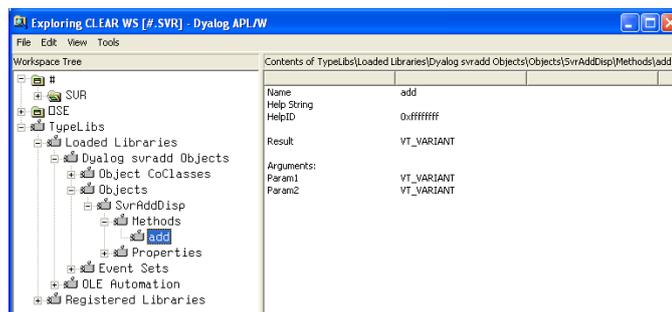


Or the same information may be obtained from the *OLEClient* method *GetMethodInfo* (see also *GetPropertyInfo*).

```
DISPLAY GetMethodInfo 'add'
```



Or the information may be obtained from the TypeLibs section of the workspace explorer.



Note that Int32[] would be a more precise, and therefore more efficient, alternative to VT\_VARIANT for these arguments and result.

### §§ 6.3.3 Calling an OLE Server from VB

Given the files SvrAdd.DLL, Dyalog.DLL and SvrAdd.TLB, any language that can access OLE servers from DLLs may be used to run your new in-process OLE server.

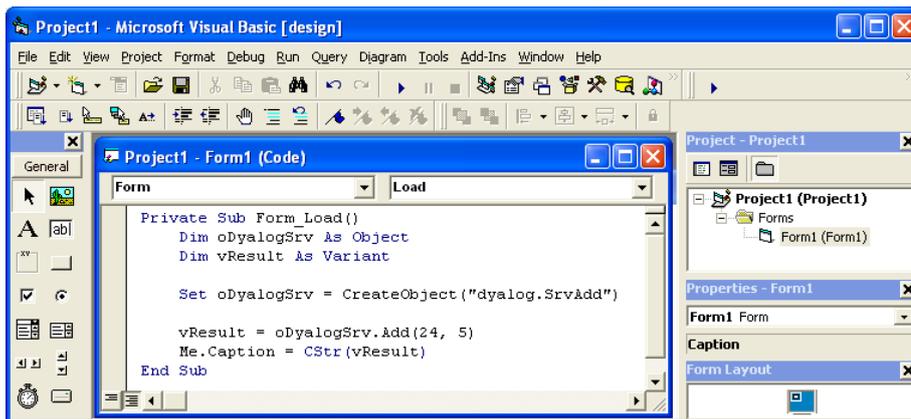
First the server has to be registered on the relevant machine, *eg* in an APL program or by the command line **C:\>regsvr32 c:\dyalog90\svradd.dll**

Then an instance of the SvrAdd object must be created and the **add** function called.

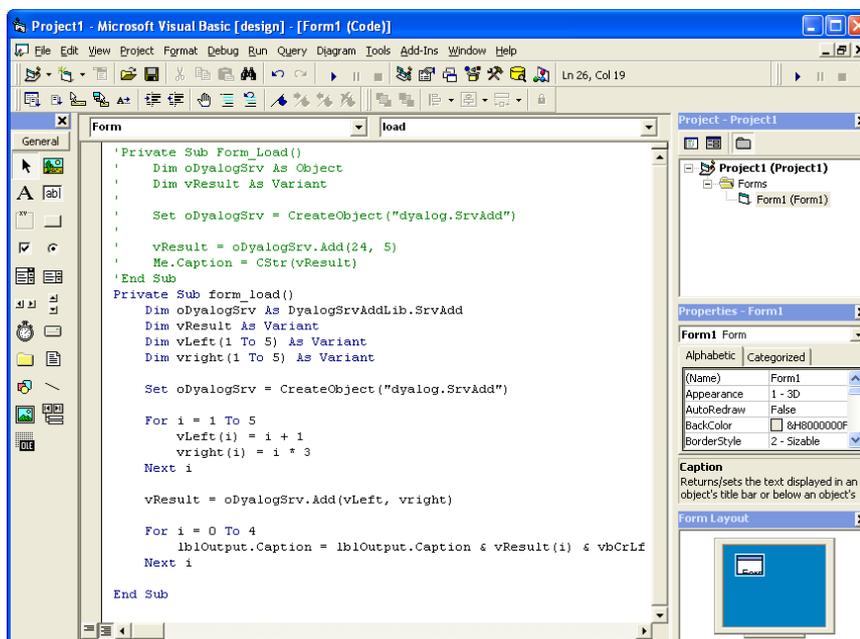
In VB a reference to the object is obtained using the CreateObject function  
**Set oDyalogSvr = CreateObject("dyalog.SvrAdd")**

then the add method may be called using VB dot syntax  
**vResult = oDyalogSvr.add(24, 5)**

In the following example, the result, for want of a better idea, is placed on the caption of a form.

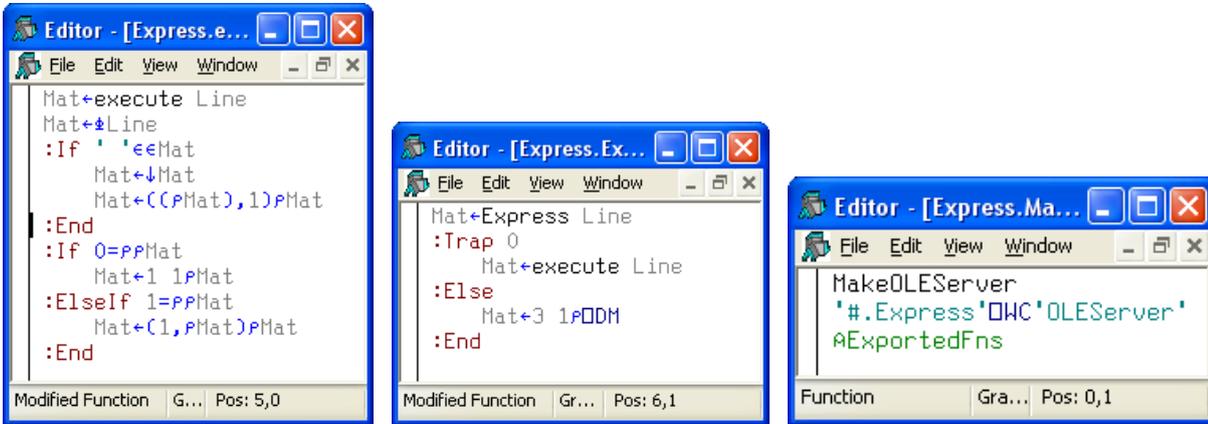


Another trivial VB example sets the form caption in a loop.



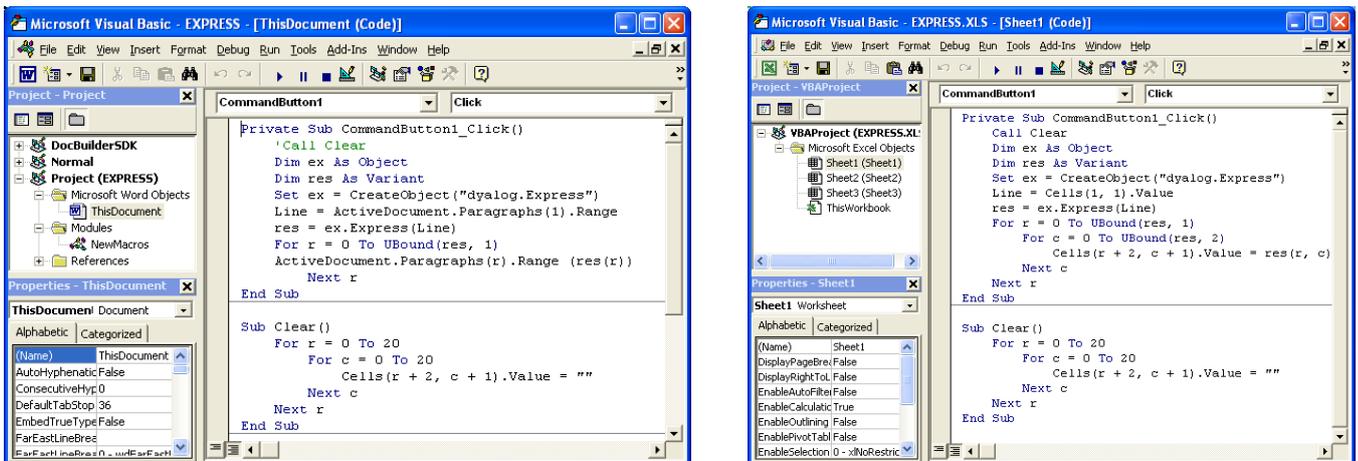
The following example is based on two simple APL functions in a namespace called `#.Express`, and a third function to turn the namespace into `Type OLEServer`.

This little example allows one to type a line of APL code into the first cell of an Excel spreadsheet and have a button on the spreadsheet execute the line. The result is placed on the ensuing lines of the spreadsheet.



The exported functions need to have their Result and Param1 types set to VT\_VARIANT before )SAVE.

The function `#.Express.Express` above is called by Visual Basic code behind Excel and the result is written to the cells of Excel in two different ways (below).



Note that later versions of Excel (from Excel 2000) use Value2 in place of Value in the VB code above.

6.3.3.1 Ask for the next module on **OLE Clients** 😊.

## Module7: OLE Clients

OLE Clients drive OLE Servers via *Object Linking and Embedding*. This provides the means for very powerful connections between applications; such as one between Dyalog APL and Microsoft Office.

### § 7.1 Inside Microsoft Word

#### §§ 7.1.1 Registry Entries, Object Models and Type Libraries

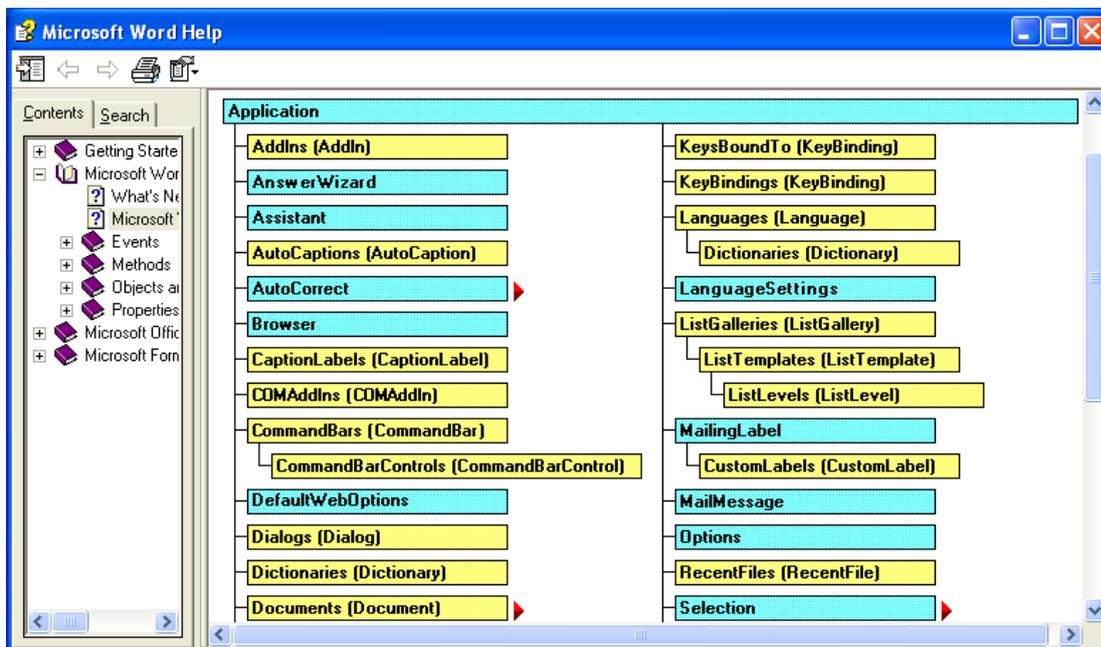
If Microsoft Office is installed on your computer, then Word.Application will appear in the list `#.OLEServers`. The corresponding class ID refers to that in the registry under HKEY\_CLASSES\_ROOT\Word.Application\CLSID. There the registry entry points to the most up-to-date version of Word currently installed, such as Word.Application.9.

The registry may be investigated further to find, for example, that HKEY\_CLASSES\_ROOT\CLSID contains key {000209FF-0000-0000-C000-000000000046}\LocalServer32 which contains the name of the program actually used for OLE automation:

C:\PROGRA~1\MICROS~2\Office\WINWORD.EXE /Automation

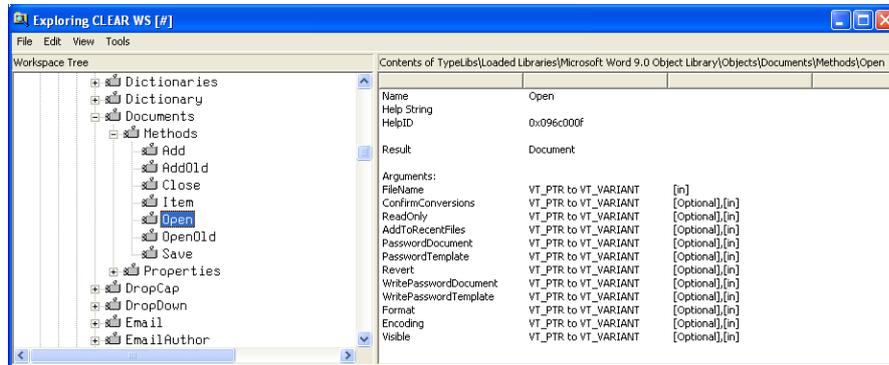
The Word program is vastly more complicated than our server of Module 6, and understanding its object structure and contents is a major challenge. The Word 9.0 VBA help file (VBAWRD9.CHM) is most helpful in this respect. The Word VBA help file is particularly geared to programmers wishing to call Word object methods and properties from different application environments such as VB or APL.

The Word object model reveals all the potential object hierarchies in a Word application. The properties and methods associated with each type of object are described in the help file, often with an example VBA call. This VBA code is close enough to an APL equivalent to be very useful as a starting template when programming Word-linked APL applications.



To access an OLE server, you create a namespace of `Type OLEClient` as an instance of the OLE server. The `ClassName` property of the `OLEClient` identifies the server and has to be set at create time. It takes the value `'Word.Application'` in the case of Microsoft Word.

7.1.1.1 Create an *OLEClient* object with *ClassName* set to *'Word.Application'*. Open the workspace explorer and browse the loaded type libraries. For example, explore the Microsoft Word Object Library and look at Objects\Documents\Methods\Open. Relate this to MS Word [File][Open].



7.1.1.2 Change into the *OLEClient* space, and compare the above with the result of `↑Documents.GetMethodInfo'Open'`

Thus there are a number of ways to find out the calling syntax and argument types for methods in Word.

### §§ 7.1.2 Digging into Word

7.1.2.1 In a clear workspace, start Word as an OLE Client and change space into the Word application.

```
□CS'WRD'□WC'OLEClient' 'Word.Application'
```

Set the *Visible* property to 1. Write a function called *▽show▽* which displays its Rarg in the session.

Set the *Event* property for all events to *▽show▽*.

```
Event←'All' 'show'
```

Look at the *EventList* property and try to fire an event that will 'show' in the session. In Word, select [File][Exit] and note the *Quit* event in the session.

7.1.2.2 In a clear workspace, trace the function below and identify the methods and properties being used.

```

▽ WordExample;WRD
[1]   :With 'WRD'□WC'OLEClient' 'Word.Application'
[2]   Visible←1
[3]   :With Documents
[4]     :With Add ∅
[5]     :With Content
[6]       Text←,(50 50ρ□A),3>□TC
[7]     :End
[8]     SaveAs'c:\myword.doc'
[9]   :End
[10]  :End
[11]  Quit ∅
[12]  :End ▽

```

At each change of space, check the name of the current namespace and the methods and properties available in that space. Right click on methods *Save*, *SaveAs* and *Quit* to view their calling information. In the workspace explorer, investigate the *#.WRD* object and its children.

Notice that some of the reported namespace names are surrounded by brackets, eg

```
)ns
```

```
#.WRD.[Documents].[_Document]
```

The brackets indicate that these namespaces have not actually been given any name at create time.

*Documents* is a property that returns a reference to a collection object. This reference is sufficient for *:With* to be able to deal with the collection, which is essentially a vector of objects (see Module11 on Arrays of Objects). *Add* is a method that returns a reference to a new *Document* object. The result could be assigned to a name if a name was required.

Information about properties in the current space may be found using *GetPropertyInfo* method of *OLEClient* objects; eg in the *Documents* collection, *GetPropertyInfo'Count'↪VT\_I4*.

### §§ 7.1.3 Demonstrating the Power of OLE

7.1.3.1 In a clear workspace, create an *OLEClient* for the OLE server Word.Application. Enter the *Documents* collection and *Open* file C:\myword.doc. Make Word *Visible*.

7.1.3.2 Enter the *Documents* collection and *Add* a new document. Trace the following function snippet which adds, fills and colours a *Table*.

```
:With Tables
  :With Add(#.WRD.Selection.Range,3,5)
    :For x :In 13
      :For y :In 15
        (Cell(x,y)).Range.InsertAfter'Cell(',(⌘x),',',(⌘y),')'
      :EndFor
    :EndFor
    Columns.AutoFit
    :With Rows
      :With Item 1
        Select
        Alignment←1
        :With #.WRD.Selection.Font
          Bold←1
          Color←256 256 256⊖255 127 0 ⌘ Orange :-)
        :End ⌘ font
      :End ⌘ row
    :End ⌘ rows collection
  :End ⌘ table
:End ⌘ tables collection
```

7.1.3.3 While still within the new *Document*, enter some text after the table using something like the following snippet, noting the parentheses and the reduced number of *:Withs*.

```
:With Paragraphs.(Item Count)
  Range.Text←1000⌘100↑20↓⌘AV
:End ⌘ paragraph
```

7.1.3.4 While still within the *Document*, add some suitable text to the first cell in the *Table* and convert the text into a *Table* (within a *Table*) as in the following snippet:

```
:With Range(0,0)
  Text←,(⌘(⌘''10),'#',''10 2⌘''130),3⇒⌘TC
  Select ⌘ this selects all
  DefaultTableSeparator←'#'
  ConvertToTable'#'
```

```

:With Tables
  :With Item 1
    :With Columns
      :With Item 1
        Select
          :With #.WORD.Selection.Font
            Bold←1
            Color←256 256 256∟φ0 0 255∟ Blue
          :End ∟ font
        :End ∟ a column
      :End ∟ a columns collection
    :End ∟ a table
  :End ∟ a tables collection
:End ∟ a range
  
```

Tip: A useful way of constructing such code, apart from valuable help from VBAWRD\*.CHM, is to record a macro in Word which takes the steps that you want your program to take, and then use **Alt+F11** to examine the macro VBA code.

Tip2: You might find that, after creating and destroying (erasing or expunging) an *OLEClient* and saving the WS, the size of the WS has grown considerably. This happens because the TypeLibs visible in WS Explorer have been saved too. They may be removed before saving by running the combination of Root methods

```
#.DeleteTypeLib">"#.ListTypeLibs
```

## § 7.2 Manipulating Microsoft Excel from the Inside

### §§ 7.2.1 Recognising the Object Model

The object model for Excel is very similar in appearance to that of Word. The model for Excel 9.0 is documented in help file VBAXL9.CHM. Clearly it is advisable to use the correct version of this help file for your particular Excel version.

As with Word, the registry entries for the Excel.Application *ClassName* can be investigated. More particularly, from the point of view of writing APL-Excel OLE applications, the TypeLibs in WS Explorer, or *GetMethodInfo*, *GetPropertyInfo*, *GetEventInfo* and *GetTypeInfo*, should be employed to obtain information about Excel functionality.

### §§ 7.2.2 Digging into Excel

7.2.2.1 Create an *OLEClient* with *ClassName* Excel.Application. Make the application *Visible*. Set the application *Caption* to eg 'My Excel'. Look at the values of the application properties *MemoryFree*, *MemoryUser*, *MemoryTotal*, *LibraryPath*, *TemplatesPath*, *Path*, *Name*, *UserName*, *Value*, *Version*, *Height*, *Width* and *WindowState*.

7.2.2.2 With the *Workbooks* collection, *Add* a new workbook and look at the values of properties *Author*, *Path*, *Name*, *FullName* and *UserStatus*. Run *GetMethodInfo* on methods *SaveAs* and *Close*. Save the file as something like 'C:\myexcel.xls' and run the method *Close*. Then *Quit* the application.

7.2.2.3 Write a function that will start Excel (visibly), enter the *Workbooks* collection and *Open* the *workbook* 'C:\myexcel.xls'. In the application space, assign the *Range* between A1 and B2 in the *ActiveSheet* to a name for this reference object and then assign the *Value* (or latterly *Value2*) property of the ref to a suitable matrix.

7.2.2.4 Given the alternative style of programming below, trace the function and explore the references.

```

▽ OLEExcel2;XL
[1]  □CS'#.XL'□WC'OLECLIENT' 'Excel.Application'
[2]  Visible←1
[3]  Wkb←Workbooks.Open'c:\myexcel.xls'
[4]  Wks←Wkb.Worksheets.Item 1
[5]  Rng←Wks.Range'A1:B10'
[6]  Rng.Value2←?10 2p100
[7]  Ch←Charts.Add 0
[8]  Ch.ChartType←xlColumnClustered
[9]  Ch.SetSourceData #.XL.Rng
[10] ▽
    
```

### §§ 7.2.3 Gaining full Control of Excel

Like Word, Excel is a very big program with many dark corners. Thankfully, the task of learning how to use the OLE interface mirrors quite closely the task of learning how to use Excel itself. (The task of learning WORDBASIC, used with the APL shared variable approach to DDE communication with Word, was closely tied to the menus of Word, but now, with OLE, much more of the detailed functionality of Word is exposed.) Each little feature in Excel that is incorporated into an APL program is a stepping stone for ever more ambitious and detailed communications with Excel.

The Word document version of the [Dyalog APL Object Reference](#) manual itself embodies an example of Word-APL OLE. Each description of an object in the manual opens with a section containing **Purpose, Parents, ..., Methods**. The contents of these sections are mustered and positioned in the Word document via OLE from within an APL workspace, thanks to the fruitful efforts of Peter Donnelly.

7.2.3.1 Write a function `▽putMatrix▽` which takes a matrix `Rarg` and an optional `Offset Larg` and places the matrix in an Excel worksheet, offset by the given number of rows and columns.

Tip: `□A,(,□A◦.,□A),(,□A◦.,□A◦.,□A)` generates the names of the first 18278 column names in an Excel worksheet (see a generalised version in the DFns Module12 ☺).

## § 7.3 Linking to other Servers

### §§ 7.3.1 Outlook

If Outlook.Application is in your list of `OLEServers`, then you can create an `OLEClient` with this `ClassName`. In that space you will find a method called `CreateItem`. This method returns an object whose type is determined by the `Rarg`. Available types are to be found in the list of Enums in the WS Explorer TypeLibs Loaded Libraries from the Microsoft Outlook 9.0 Object Library. The Enums section has an entry called `olItemType`. This contains a list of possible types and the Enum appropriate for each case. For example, to create a mail item use Enum `olMailItem` which has value zero. Thus the `CreateItem` method can take a `Rarg` of 0 or `olMailItem`. (`olMailItem` is an invisible keyword in the WS, not listed under `)VARS` or `)PROPS` and with `□NC'olMailItem'↳0`)

7.3.1.1 Enter the following two lines and assign the properties `Subject` and `Body` to suitable values. The `Body` of a message is a character string, as may be deduced from the result of `GetPropertyInfo'Body'`. Each new line in the `Body` should be terminated with a linefeed character. If `⊢□ML<3` then `⊢□TC[1+□IO]≡□AV[2+□IO]` and this is the linefeed character needed.

```

□CS'Outlook'□WC'OLEClient' 'Outlook.Application'
□CS CreateItem olMailItem
    
```

The current namespace contains a property that returns a *Recipients* collection. As usual, in this collection space there is an *Add* method that returns an object of appropriate type. The *Add* method type library calling information describes the Rarg of *Add* as VT\_BSTR. It is in fact an enclosed character string which corresponds to an entry in your Outlook address book or a raw email address. For example

```
Recipients.Add='Karen Shaw'
```

will check your address book and resolve the entry, if possible. If you are lucky, it might be resolved to [karen.shaw@monadic.com](mailto:karen.shaw@monadic.com) or possibly [briony.williams@dyadic.com](mailto:briony.williams@dyadic.com) or even to [pauline.brand@triadic.com](mailto:pauline.brand@triadic.com). Or you could add the recipient's email address directly as

```
Recipients.Add='karen.shaw@dyadic.com'
```

Any number of recipients may be added in this way.

You can tell if a name was resolved successfully from the result of the niladic method; *Resolve*. If it does not get resolved properly and *!~Resolve* then the message may be removed by means of the *Remove* method in the *Recipients* collection. The Rarg of this method is the item number of the recipient object in the collection. If there is only one recipient object in there, then Rarg is 1.

At this point the *Type* of the message may be changed. (Note the *Type* keyword conflict and its resolution.) The default *Recipients.Type* is 1, or *olTo*, but it could be any of a number of *Types*, their Enums being those in the group *olMailRecipientType*. (Double clicking on these key words in the APL session displays their contents.)

The names of those to whom the message will be addressed is returned by the *To* property in the unnamed [\_MailItem] namespace. If some of the recipients were of *Type olBCC* then the property *BCC* returns that list of names to be blind carbon copied (see VBAOUTL\*.CHM).

At this point all that needs to be done is to run the niladic, non-result returning method *Send*.

7.3.1.2 The mail item namespace contains a property called *Attachments* that returns a collection of attachment objects. Attach a file to an email by calling the *Add* method of this collection, after checking the method's calling information using the property sheet obtained by right-clicking on the method name in the session, or by way of the WS Explorer.

```
Attachments.Add'C:\myword.doc'
```

## §§ 7.3.2 Microsoft Internet Explorer

Have you ever wanted an APL function that takes a url as its argument and returns the retrieved html text as its result? The InternetExplorer.Application may be used for this, as Tommy Johansen has adeptly shown to the [dyalogusers@yahoogroups.com](mailto:dyalogusers@yahoogroups.com) mailbox group.

7.3.2.1 Start Internet Explorer as an OLEServer by

```
□CS'IE'□WC'OLEClient' 'InternetExplorer.Application'
```

Your program might need a delay (□DL) of a few seconds at this point to give time for the server to initialise properly. There is a *Busy* property in the application that should be queried after each significant action and a short delay included in the program *:While Busy.Visible* may be set to 1 if you wish to see the IE application activity.

7.3.2.2 Take any Internet address of interest and *Navigate* to that address, eg

```
Navigate'http://www.simcorp.com'
```

or

```
Navigate'http://finance.yahoo.com/q/ecn?s=IBM'
```

or

```
Navigate' http://finance.yahoo.com/p?v&k=pf_1'
```

or

```
Navigate' http://news.bbc.co.uk/1/hi/business/7206270.stm'
```

7.3.2.3 In order to obtain the body section of the resulting html, first get the *DispHTMLDocument* object returned by the *Document* property and then get the *DispHTMLBody* object returned by the document's *body* property. Note the lower case spelling of *body* and of many of the object's other properties and methods. The *outerHTML* property of the *DispHTMLBody* object contains the <BODY>..</BODY> character string.

### §§ 7.3.3 Beyond

It is possible, through OLE, to link to any of the servers listed by the Root property, *#.OLEServers*. For example, if you have DAO.dbEngine installed then it is possible to read data from Microsoft Access files directly into an APL workspace. Or if you have CrystalReports installed you can communicate directly with that application. All sorts of applications, from computer-animated *synthespians* (examples of which may be downloaded from <http://www.microsoft.com/msagent/downloads/developer.aspx>) to music and radio players, may be incorporated into your APL applications by object linking and embedding.

As a final simple example, the call to `⎕CMD` to open Notepad

```
⎕CMD'Notepad' ''
```

may now be replaced by

```
'WSS'⎕WC'OLEClient' 'WScript.Shell'  
WSS.Run'Notepad'
```

or

```
WSS.Exec'Notepad'
```

Remember to clean up the workspace by

```
#.DeleteTypeLib''>'#.ListTypeLibs
```

before saving if you don't want to save all the TypeLibrary information in the workspace.

7.3.3.1 Ask for the next module on **ActiveX Controls** 😊.

## Module8: ActiveX Controls

In 1996, Microsoft renamed the OLE 2.0 technology to ActiveX. An ActiveX Control is like an OLE Server but it is stored inside an OCX file rather than a DLL file. The significant new feature is that an ActiveX Control may be instantiated as a GUI object inside another GUI application.

### § 8.1 Creating an ActiveX Control in an OCX

#### §§ 8.1.1 The *ActiveXControl* Object

Since an ActiveX control is intended to be the child of another GUI object, the Dyalog APL *ActiveXControl* object, unlike the *OLEServer* object, cannot be created as a child of the Root. It must be created as the child of a *Form*. This *Form* object is the notional container object for the ActiveX. When the *ActiveXControl* is created within another application, the *Form* will be replaced by the relevant GUI parent from within the application. (Some details about the actual container object involved in any particular environment may be found from the *ActiveXContainer* object that is returned by the *Container* property of the *ActiveXControl*.)

8.1.1.1 Create a *Form* with an *ActiveXControl* child object.

```
'F'⊂WC'Form'('BCol' 120 20 230)
⊂CS'AXC'F.⊂WC'ActiveXControl'
```

The *ClassID* property contains a unique identifier for this control. *Container.BCol* should return the *Form BCol*.

#### §§ 8.1.2 The *Create* Callback

An *ActiveXControl* is created whenever it is first used. A function is usually required in order to initialise a control. This function may, for example, create the children of the ActiveX as soon as the ActiveX exists. Two events are supplied for initialisation code, *Create* and *PreCreate*.

8.1.2.1 Assign the *onCreate* property of *F.AXC* to function *∇cre8∇*, where

```
∇ cre8
[1]   'C'⊂WC'Calendar'
∇
```

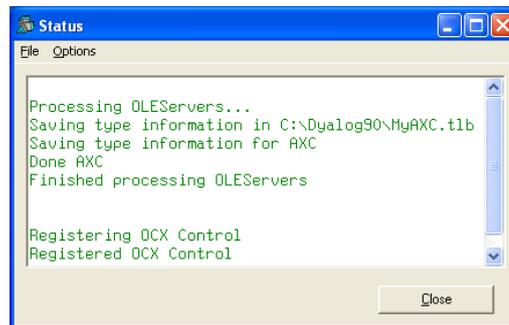
This will create a *Calendar* child of the *ActiveXControl*. Experiment with a more complicated callback such as:

```
∇ cre8;BB
[1]   :With 'BB'⊂WC'BrowseBox'
[2]       StartIn←'C:\'
[3]       onFileBoxOK onFileBoxCancel←1
[4]       Caption←'Resource Brower'
[5]       HasEdit←1
[6]       Msg←⊂DQ''
[7]       :If 'FileBoxOK'≡2>Msg ◇ Dir←Target
[8]       :Else ◇ Dir←'' ◇ :EndIf
[9]   :EndWith
∇
```

### §§ 8.1.3 Creating the OCX on )SAVE

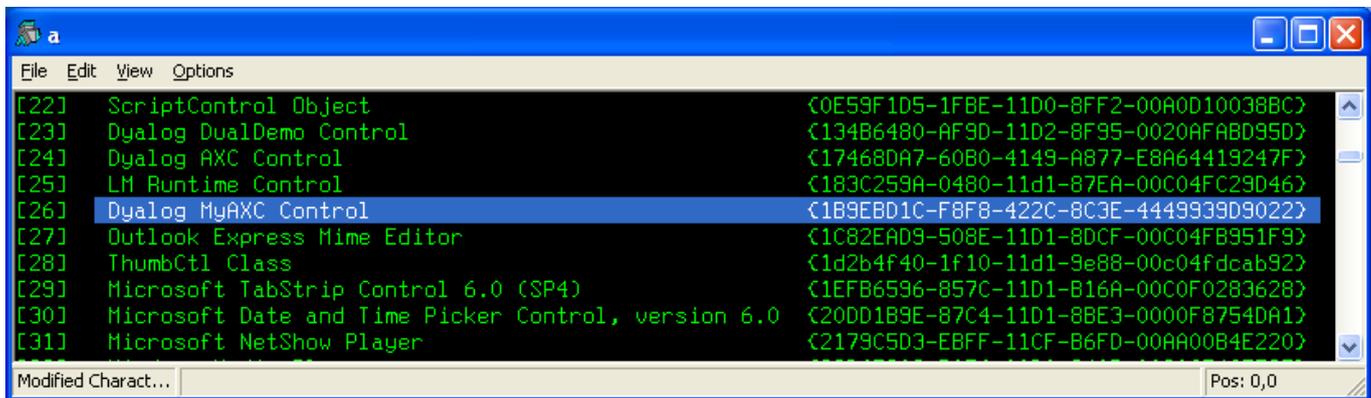
With the above ActiveX we have not introduced any functions to be exported as methods (or properties), nor any variables to be exported as properties. If we had functions or variables to be exported for use by the external application then we may set their calling structure using *ActiveXControl* methods *SetFnInfo* or *SetVarInfo* under program control. An external event may also be declared using *SetEventInfo*. Alternatively, the entire control might be constructed manually and the calling information set in the property sheets obtained by right-clicking on the name and selecting [Properties][COM Properties].

8.1.3.1 Check that [File][Make OCX/DLL on )SAVE] is checked and )SAVE the workspace as MyAXC.DWS. This generates type information in a TLB file, just as is done for an *OLEServer*. The OCX Control thus created is registered in the Windows registry in a similar way to an OLE Server.



There now exists a MyAXC.OCX file as well as the MyAXC.DWS file, and there is a new entry in the list of OLEControls:

`a←↔↑#.OLEControls`



This list of controls contains the unique identifier found in the *ClassID* property of the *ActiveXControl*. This ClassID resides in the Registry.

8.1.3.2 Run **REGEDIT.EXE** and find details of Dyalog.MyAXC.MyAXCctrl.1 under key \HKEY\_CLASSES\_ROOT\Dyalog.My AXC.MyAXCctrl.1\CLSID. In particular, notice mention of the file MyAXC.OCX under key InProcServer32.

8.1.3.3 Write an ActiveX control which reads and writes a character matrix in component 2 of an APL component file (component 1 should always be reserved for some sort of a description). Create the file if it doesn't exist. Display the component for editing in a multi-line *Edit* object.

An OCX file represents a set of ActiveX controls that are ready to run when supplied with the particular Dyalog APL Dynamic Link Library DYALOG.DLL that was in use when the OCX file was saved.

## § 8.2 Using your ActiveX Control

### §§ 8.2.1 Creating an Instance from an *OCXClass*

The way that you use an ActiveX control in Dyalog APL is by way of an *OCXClass* object. An *OCXClass* object is created from some registered control by setting the *ClassName* of the *OCXClass* at create time to the name of the control precisely as given by *#.OLEControls*. The name given to the *OCXClass* object is then the name of a new *Type* of object, instances of which may be created in the usual way as children of appropriate parents.

8.2.1.1 In a CLEAR WS create a new class (*Type*) of object and an instance of this new class on a *Form*.

```
'MyAXC'⊂WC'OCXClass' 'Dyalog AXC Control'
'Frm'⊂WC'Form'
'Frm.Inst'⊂WC'MyAXC'
```

Change some of the properties of this instance.

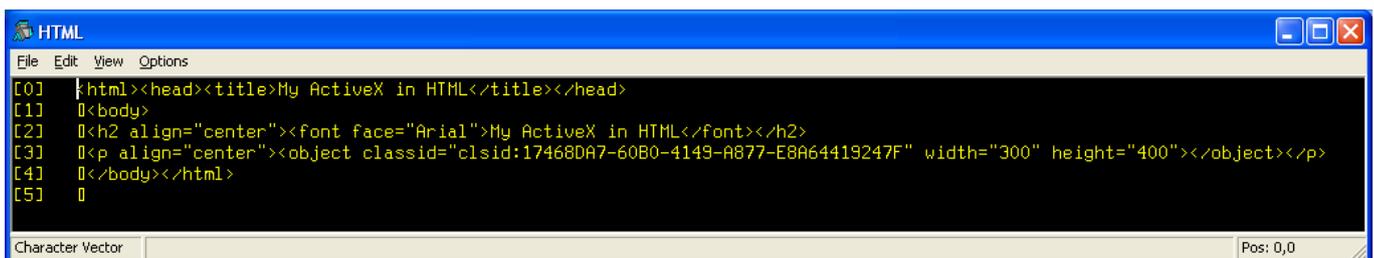
This control may be called from any OLE-aware application via a suitable language such as VBA or JavaScript.

8.2.1.2 Load the supplied DUALBASE workspace and follow the instructions in chapter 14 of the [Dyalog APL Interface Guide](#) (which may be freely downloaded from [www.dyalog.com](http://www.dyalog.com) [Download zone]). Further instructions on the Dyalog Dual Control can be found in the APL98 course notes in APL98OCX.ZIP which may also be downloaded from [Products][Dyalog for Windows][Writing ActiveXControls].

### §§ 8.2.2 Using Controls in IE 6.0

ActiveX controls can be incorporated into HTML pages by reference to the classid of the control.

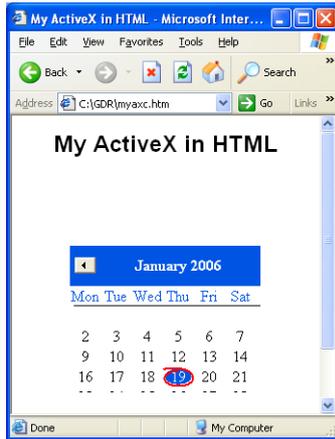
8.2.2.1 Create a native file called MyAXC.HTM containing the HTML string in the window below, using the *ClassID* of your object. The squish-quad represents carriage return/linefeed, ie `⊂AV[ 3 2+⊂IO]`.



```
HTML
File Edit View Options
[01] <html><head><title>My ActiveX in HTML</title></head>
[11] <body>
[21] <h2 align="center"><font face="Arial">My ActiveX in HTML</font></h2>
[31] <p align="center"><object classid="clsid:17468DA7-60B0-4149-A877-E8A64419247F" width="300" height="400"></object></p>
[41] </body></html>
[51] </>
Character Vector Pos: 0,0
```

The source and display look something like:

```
<html>
  <head>
    <title>My Active X Control</title>
  </head>
  <body>
    <h2 align="center">
      <font face="Arial">A X C</font>
    </h2>
    <p align="center">
      <object
        classid="17468DA7-60B0-4149-A877-E8A64419247F"
        width="300" height="400">
      </object>
    </p>
  </body>
</html>
```



More information on how to do this is to be found in <http://support.microsoft.com/kb/q159923/> where licensing of controls is also discussed.

### §§ 8.2 3 Using Controls in Visual Basic and VBA

In APL98OCX.RTF and in chapter 14 of the *Dyalog APL Interface Guide* you will find explanations of how to call the Dual Control example from VB, VBScript and IE.

## § 8.3 Browsing registered OLE Controls

### §§ 8.3.1 Having a quick Look

A snippet from a function by Dick Bowman shows how you might go about investigating the controls in your, possibly very large, list of `#.OLEControls`.

```
:For Item :In >''. 'WG'OLEControls'
  Item
  'Inst'WC'OCXClass'Item
  'F'WC'Form'
  'F.X'WC'Inst'
  :Trap 11
    2 NQ'F.X' 'ShowHelp' 'Type'
  :End
:End
```

### §§ 8.3.2 Having a deeper Look

For a deeper look into the registered controls on your list, the supplied workspace OCXBROWS.DWS may be used to investigate available controls.

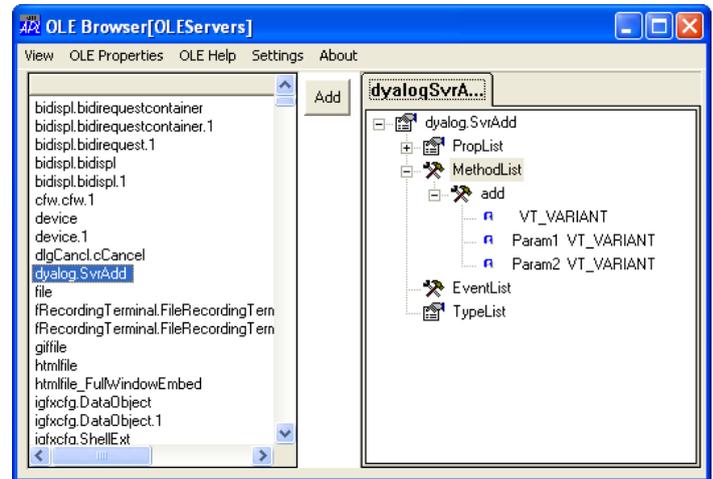
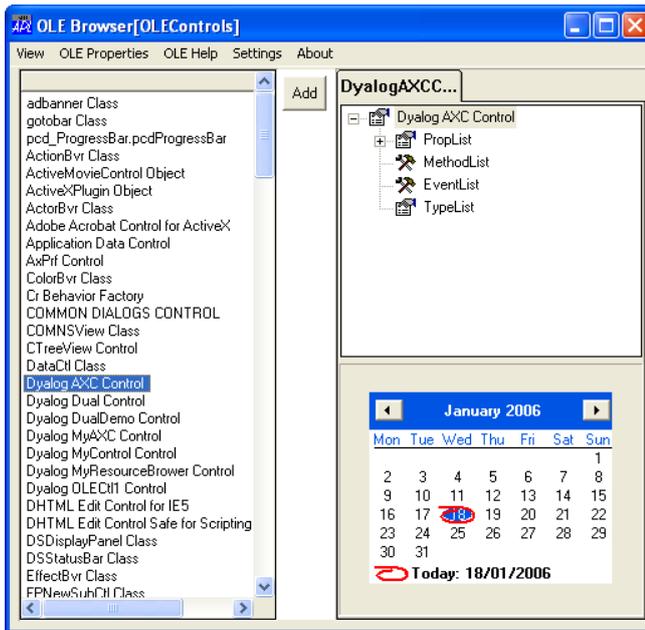
8.3.2.1 XLOAD the workspace OCXBROWS.DWS and trace the `⊞LX, ⊖CLASSES.LISTCLASSES`. Skip over line [17] and trace into line [18]. Select 'Dyalog MyAXC Control' and hit [Details]. Trace into and examine line [41].

Hint: Normalise `Form` to continue beyond line SHOWCLASS[15].

Alexander Balako has kindly donated his workspace OLEBROWS.DWS as a free download in [www.dyalog.com](http://www.dyalog.com) [Download Zone].[OLEBROWS]. This workspace enables exploration of both the ActiveX controls in your list `#.OLEControls`, and the OLE servers in your list `#.OLEServers`.

8.3.2.2 In OLEBROWS.DWS, attempt to trace the `⊞LX, ⊖OLEBrowser.MAIN`, and note the use of the `Create Event` on line [1].

Hint: Put a `⊞STOP` on line [4] and run the function past line [3].



### §§ 8.3.3 Trying some Examples

The Dyalog APL *ActiveXControl* object allows you to package an application as an ActiveX control. The application then comprises at least two files; your OCX file and the Dyalog APL dynamic link library file, DYALOG.DLL, to which your OCX was linked when created. Both files have to be visible when the OCX is registered. Also the full path name of the OCX is recorded in the registry so it cannot be moved around easily. It is conventional to place OCX and DLL files in the ..\Windows\System32\ directory.

Let us call some controls from an APL workspace. If you have the VideoSoft FlexArray control on your list then you can start it with:

```
'FAC'⊂WC'OCXClass' ':-) VideoSoft FlexArray Control'
'Frm'⊂WC'Form'
'Frm.Flex'⊂WC'FAC'
```

8.3.3.1 You probably have Windows Media Player on your list. Create an instance of this control on a *Form* and examine the instantiated object from the inside. Research the *FileName* and *QueueEvents* properties of the class. In Dyalog APL version 10, the *Grid* object can have controls inside cells. Run the following lines of code and explore the possibility of playing 25 tunes at once!

```
'WMP'⊂WC'OCXClass' 'Windows Media Player'
'F'⊂WC'Form'('Coord' 'Pixel')
H W←(F.Size-4)÷5
'F.G'⊂WC'Grid'(5 5p'')(0 0)
F.G.CellHeights←H ◊ F.G.CellWidths←W
F.G.Size←F.Size
F.G.ShowInput←1
F.G.TitleHeight←0 ◊ F.G.TitleWidth←0
'F.G.WMP'⊂WC'WMP'
F.G.Input←'F.G.WMP'
```

The Google search bar, donated by Norbert Jurkiewicz, as mentioned in Module4, can be summarised, in essence, in the following function which takes any Google search string as its Rarg. The function creates the Microsoft Web Browser *MSWB* class object and an instance of the class on a suitably sized *Form*. The *DocumentComplete* event is set to 1 in order to terminate the ensuing `⎕DQ` when the entire document has been obtained. Then come the crucial lines. We invoke the objects *Navigate* method with a carefully crafted argument:

```
http://www.google.com/search?ie=UTF-8&oe=UTF-8&sourceid=deskbars&q=',XXX
```

The *XXX* is a character string that contains a search request string such as one would type into Google. The rest is the URL-specific command line suitable for a general Google search.

```

▽ searchFor XXX;To
[1]  'MSWB'⎕WC'OCXClass' 'Microsoft Web Browser'('QueueEvents' 0)
[2]  'F'⎕WC'Form'('Coord' 'Pixel')('OnTop' 1)('Border' 0)
[3]  F.Size←400 600
[4]  F.EdgeStyle←'Recess'
[5]  'F.IE'⎕WC'MSWB'
[6]  F.IE.Posn←0 0
[7]  F.IE.Coord←'Prop'           Ⓜ MUST USE PROP to look ok
[8]  F.IE.Size←100 100
[9]  F.IE.onDocumentComplete←1 Ⓜ exit ⎕DQ when finished
[10] To←'http://www.google.com/search?'
[11] To,←'ie=UTF-8&oe=UTF-8&sourceid=deskbars&q=',XXX
[12] F.IE.Navigate To
[13] ⎕DQ'F'
[14] F.IE.Refresh             Ⓜ MUST REFRESH to look ok
▽

```

8.3.3.2 Explore the possibility of viewing 25 Internet sites simultaneously in a 5 by 5 *Grid*.

8.3.3.3 Please ask for the next module on `⎕NA` 😊.

## Module9: C Function Access

Pre-.NET Dynamic Link Libraries (DLLs) are libraries of compiled functions. These functions may be accessed and run from within an APL workspace by means of the system function `⊞NA`. Details may be found in the *Dyalog APL Language Reference*.

### § 9.1 Declaring “dataTypes” of Arguments and Results

#### §§ 9.1.1 Quick View of DLLs and their Contents

The main sources of useful compiled C functions for general APL applications are to be found in the files `advapi32.dll`, `gdi32.dll`, `kernel32.dll` and `user32.dll`. These files reside in the `..\windows\system32\` directory under Windows XP. The Windows utility *QuickView* which used to be bundled with the Accessories of Windows 98 is no longer supplied in later versions of Windows. This facility was very useful as it allowed one to find out what functions are included in any given DLL.

However, a full list of usable Windows functions is given in the MSDN library, at <http://msdn.microsoft.com/library/>, under [Win32 and COM Development][Development Guides][Windows API][Windows API][Windows API Reference], where functions from various DLLs are listed by name or by category. As is often the case with Microsoft documentation, unless you know what you are looking for, the volume of practically un navigable technical information can be disheartening. Nevertheless there are numerous other sources of this information in books, such as *Microsoft Windows 32 API Programming Reference, Volumes 1 and 2* from Microsoft Press, and in various places on the Internet.

#### §§ 9.1.2 The Meaning of the right Argument of `⊞NA`

`⊞NA CVec`

⌘ Fixes function as defined by CVec

The character vector `Rarg` to `⊞NA` contains a number of distinct parts. Essentially, there are 4 separate parts in the string.

1. The first describes the variable `dataType` of the result. This element may be elided if there is no result from the C function, or if none is required.
2. The second part is the name of the file containing the compiled C function. This may be the full path name if the DLL is not in a visible directory such as `..\system32\`.
3. The third part, separated from the second by a bar (`|`), is the name of the function to be called from the DLL.
4. The fourth, and most complicated part, contains the specification of the variable `dataTypes` of the elements of the (right) argument to be supplied to the function after it has been fixed from the DLL.

For example, the following `CVec` refers to a function called `SystemParametersInfoA` which is found in library `User32.dll`.

```
'I4 User32|SystemParametersInfoA I4 I4 >{I4 I4 I4 I4} I4'
```

The basic function result is a 4 byte integer and the argument to be supplied has 4 elements. The first, second and last are 4 byte integers and the third consists of a string of 4 byte integers which are to be used to capture the memory contents of a useful set of data indicated by C code pointers.

9.1.2.1 Fix the function `SystemParametersInfoA` in a clear workspace and display the result of the call

```
SystemParametersInfoA 48 0 (0 0 0 0) 0
```

### §§ 9.1.3 Discovering C Function Syntax

Let us start with a very simple, but very useful, example. The function **GetSystemMetrics** takes an integer argument and returns an integer result. The meaning of the argument and result can be found in <http://msdn.microsoft.com/en-us/library/ms724385.aspx> .

According to this documentation, "the **GetSystemMetrics** function retrieves various system metrics (widths and heights of display elements) and system configuration settings. All dimensions retrieved by **GetSystemMetrics** are in pixels." The calling syntax is given as:

```
int GetSystemMetrics(  
    int nIndex  
);
```

and the single parameter argument, *nIndex* is defined as "the system metric or configuration setting to retrieve." There then follows a table of possible values and their meaning. Int is a 32 bit signed integer.

9.1.3.1 Define function *GetSystemMetrics* in your workspace and determine the meanings of the first 20 calls.

```
⎕NA'I4 user32|GetSystemMetrics I4'  
GetSystemMetrics¨i20  
1024 17 17 26 1 1 3 3 17 17 32 32 32 32 20 1280 968 0 1 17
```

Hint: See function *#.WDesign.GetSystemMetrics* in the supplied workspace WDESIGN.DWS for a good short explanation of each metric.

## § 9.2 Examples of C Function Calls

### §§ 9.2.1 Simple Examples

Another simple useful example of an API call is the function **GetCurrentDirectory** which retrieves the current directory for the current process. Its syntax is documented as:

```
DWORD GetCurrentDirectory(  
    DWORD nBufferLength,  
    LPTSTR lpBuffer  
);
```

In this case there is a result described as a DWORD, and a 2-parameter argument described as a DWORD and an LPTSTR. The meanings of these parameters are defined as "the length of the buffer for the current directory string..." and "a pointer to the buffer that receives the current directory string..."

This translates to

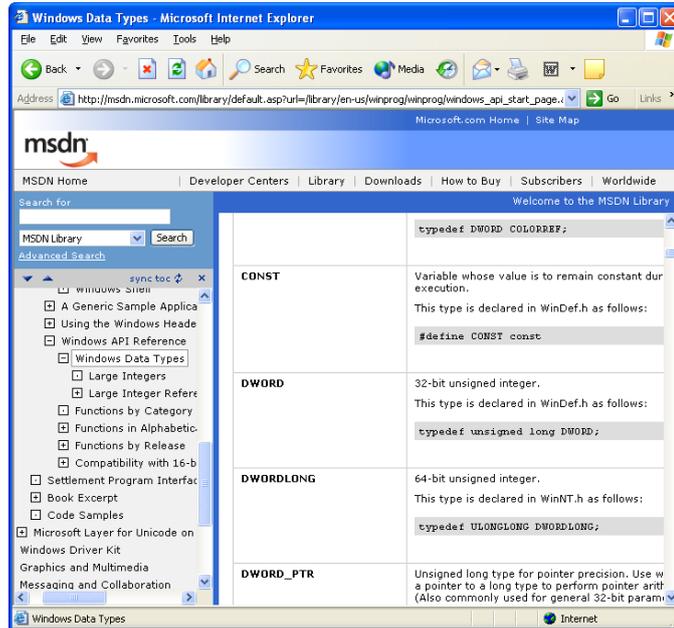
```
⎕NA'kernel32|GetCurrentDirectoryA U4 >0T'
```

because a DWORD is defined (see below) as a 32 bit (4 byte) unsigned integer, which translates to *U4*, and LPTSTR is a pointer to a null-terminated character string, which translate to *>0T* in *⎕NA* syntax. The *>* indicates that the contents of the pointed memory location assumed by the template argument will be used and overwritten by pointer-type *output* from the C function. The zero implies a null-terminated string, and the *T* means char - an 8-bit Windows (ANSI) character. **GetCurrentDirectoryA** is the ANSI version of the function and **GetCurrentDirectoryW** is the Unicode version.

So a call such as

```
GetCurrentDirectoryA 100 200  
C:\Dyalog90
```

fills in an output buffer at pointer position 200 with an ANSI string of up to 100 characters long, the last character being a terminating null character.



There are 8 bits in a byte (and, not coincidentally, 8 wires in the keyboard cable). Each bit can be a **1** (voltage ON) or a **0** (voltage OFF) so there are  $2 * 8 \hookrightarrow 256$  possible combinations of bits in a byte.

$$\begin{matrix} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ \hline 8 \text{ bits} & = & 1 \text{ byte} \end{matrix}$$

**ASCII** characters use only 7 bits, giving  $2 * 7 \hookrightarrow 128$  combinations. This used to be sufficient space for all the common letters and symbols on telex and teletype terminal keyboards. The 8<sup>th</sup> bit was often used in communications for a parity check. **ANSI** characters use all 8 bits and therefore allow 256 distinct characters to be defined (hence the length of `AV`). **Unicode** characters generally take 16 bits (2 bytes), giving  $2 * 16 \hookrightarrow 65536$  distinct combinations. Therefore Unicode allows at least 65,536 different characters to be defined.

If interpreted as a number, then 2 bytes can represent any number between 0 and 65535 for (unsigned) `U2`, or any number between -32768 and 32767 for (signed) `I2` in which the first bit is used for the sign.

$$\begin{matrix} (16 \rho 2) \tau (2 * 15) - 1 & \text{a Biggest positive number} \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ (2 * 15) - 1 & \hookrightarrow & 32767 \end{matrix}$$

$$\begin{matrix} (16 \rho 2) \tau 2 * 15 & \text{a Smallest negative number (2's complement)} \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 * 15 & \hookrightarrow & -32768 \end{matrix}$$

The `GetCurrentDirectoryA` function returns the current working directory as a string via a pointer reference. Be sure to allocate enough space for the string or you might get a GPF! Incorrect coding of `AV` function argument parameters is the most common cause of `SysError 999` crashes in Dyalog APL.

## §§ 9.2.2 More complex Examples

We shall look at 3 functions in the ADVAPI32.DLL library, which together give read and write ability to and from the Windows Registry. They are therefore very useful in many APL applications. The functions are **RegCreateKeyExA**, which returns a handle to a given registry key, creating it if it does not already exist, **RegQueryValueExA**, which "retrieves the type and data for a specified value name associated with an open registry key", and **RegSetValueExA**, which "sets the data and type of a specified value under a registry key."

These three functions, and others, are well described in a workspace kindly supplied by Alex Kornilovski, to be found in the [Download][AKUTILS] section of [www.dyalog.com](http://www.dyalog.com), or in the very large collection generously provided by Ray Cannon to be found in [Download Zone][Pocket Dyalog][padlls.dws].

The *GetEnvironment* method of Root with argument '*IniFile*' returns, by default, the Dyalog registry subkey of the HKEY\_CURRENT\_USER key.

```
#.GetEnvironment'IniFile'↵'SOFTWARE\Dyalog\Dyalog APL/W 11.0'
```

RegCreateKeyExA takes this key and returns a handle to it for use with the other 2 functions. The function syntax is described as:

```
LONG RegCreateKeyEx(  
  HKEY hKey,  
  LPCTSTR lpSubKey,  
  DWORD Reserved,  
  LPTSTR lpClass,  
  DWORD dwOptions,  
  REGSAM samDesired,  
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
  PHKEY phkResult,  
  LPDWORD lpdwDisposition  
);
```

which we may implement as

```
[NA'ADVAPI32.dll|RegCreateKeyExA U <0T I <0T I I I >U U'
```

Notice *I* and *U*, without a numeric qualifier, implies width 2 for 16-bit DLLs or width 4 for 32-bit DLLs.

```
Key←2147483649      ⍎ HEX 0x80000001 = HKEY_CURRENT_USER  
SubKey←#.GetEnvironment'IniFile'  
Access←983103      ⍎ HEX 0xF003F    = KEY_ALL_ACCESS  
  
Rarg←Key SubKey 0 '' 0 Access 0 0 0  
|RegCreateKeyExA Rarg↵600
```

Read this as "it happens to be true that *RegCreateKeyExA* applied to the above (enumeration) key and subkey returns the handle 600." We need to use 600 for the next function call. (The special numbers quoted above are slightly more meaningful in their HEX representation.)

9.2.2.1 Compare *Rarg* with the C function syntax above and the description of the parameters in <http://msdn2.microsoft.com/en-us/library/ms724844.aspx> .

Symbol < indicates a pointer pointing at *input* to the DLL function.

The **RegQueryValueEx** function retrieves the type and data for a specified value name associated with an open registry key, identified by its handle.

```
LONG RegQueryValueEx(
    HKEY hKey,
    LPCTSTR lpValueName,
    LPDWORD lpReserved,
    LPDWORD lpType,
    LPBYTE lpData,
    LPDWORD lpcbData
);
```

This syntax may be translated to

```
□NA'I ADVAPI32.dll|RegQueryValueExA U <0T I =I >0T =I4'
```

The result is a LONG which we can identify as integer *I* or *I4*. The key is an unsigned integer, *U*. The name of the value of interest is to be input and is interpreted as a null-terminated character string.

```
Key←600      a Handle
ValueName←'log_file'
DataType←1 a String data type (REG_SZ)
```

Given the □NA specification above we expect *RegQueryValueExA* to return a 4 element vector representing the result (*I*), the dataType (*=I*), the data (*>0T*) and the number of bytes used (*=I4*). Note that the equals sign (=) is used to specify parameters which are both input (<) and output (>) pointers.

```
DISPLAY RegQueryValueExA Key SubKey 0 DataType 255 255
+-----+
|      +-----+      |
| 0 1 |C:\Dyalog90\default.dlf| 24 |
|      +-----+      |
+ε-----+
```

If we were not concerned with anything but the data value, we might use the specification

```
□NA'ADVAPI32.dll|RegQueryValueExA U <0T I <I >0T <I4'
```

but it is obviously not advisable to completely ignore error flags.

The complementary function, **RegSetValueEx**, sets the data and type of a specified value under a registry key. The C function has syntax declared as:

```
LONG RegSetValueEx(
    HKEY hKey,
    LPCTSTR lpValueName,
    DWORD Reserved,
    DWORD dwType,
    const BYTE* lpData,
    DWORD cbData
);
```

This may be translated as

```
□NA'I ADVAPI32.dll|RegSetValueExA U <0T I I <0T I4'
```

9.2.2.2 Call functions *RegCreateKeyExA*, *RegQueryValueExA* and *RegSetValueExA* with suitable arguments to access, replace and create some registry entries.

### §§ 9.2.3 Other API Calls

The workspace SQAPL residing in the WS directory of your Dyalog APL installation is a very useful example of a system written in C and linked to APL via `□NA`. The system allows access to ODBC data sources and is described in chapter 16 of the [Dyalog APL Interface Guide](#).

9.2.3.1 In library User32.DLL there is a function called `SetCursorPos` which moves the cursor to the specified (X,Y) screen coordinates, in pixels. The C function syntax is specified as

**BOOL SetCursorPos(int X, int Y);**

Define this function in you workspace and check that it works as expected.

9.2.3.2 In library User32.DLL there is a function called **FindWindowA**. This function can determine if another application is currently running on your system. It accepts two string arguments, one for the class name of the application, and another for the window title bar caption. The result is an unsigned integer giving the handle to a window. The first argument is also an unsigned integer that can be given the value zero. The second argument is a null-terminated string containing the caption of the window. Define this function in your workspace. Create a *Form* in the workspace with some specific *Caption*. Look at the *Handle* property of this *Form* and compare that with the result of the function `FindWindowA`.

9.2.3.3 As demonstrated by Thomas Gustaffson, the function `WinExec` in Kernel32.DLL runs a specified application and may be used to replace a call such as

```
□CMD'Notepad' ''
```

The first parameter argument of `WinExec` is a pointer to a null-terminated character string that contains the command line for the application, the second argument is an integer such that 1 means "show window".

Define this function and use it to replace the `□CMD` command above. Note that, as in the `□CMD` case, the first parameter must be surrounded by double-quotes if there are any spaces in the string.

There are a number of working examples in the supplied workspaces QUADNA.DWS, NTUTILS.DWS and WDESIGN.DWS. QUADNA contains a particularly interesting example, `ChooseColor`, which requires a pointer to a structure which itself contains a pointer to an array. Notice that this workspace makes calls to the library `DYALOG32.DLL`. There are many other examples of `□NA` calls in freely available workspaces such as those kindly supplied by Alex Kornilovski and Ray Cannon.

Sometimes the main difficulty with utilizing external functions is not in the construction of `dataType` specifications but in the interpretation of the final result. For example, `GetVersion` should be defined as taking no arguments and returning an integer result. To decipher the meaning of this result requires appropriate documentation, as can be seen from the code snippet below:

```
code←GetVersion
code←(32ρ2)⊥code
code←(⊃code)(2⊥~8↑code)
:Select code
:CaseList (0,3 4 5)
    R←'Windows NT/2000/XP'
:Case 1 4
    R←'Windows 95/98'
:Case 1 3
    R←'Win32s with Windows 3.1'
:Else
    R←'? '
:End
```

`GetVersion` has been superseded by `GetVersionExA` in the same library. The new function returns a more complex structure and may be fixed by

```
⎕NA'I kernel32|GetVersionExA = {I4 I4 I4 I4 I4 T[128]}
```

### § 9.3 Harnessing large C Libraries

#### §§ 9.3.1 Fastest Fourier Transform in the World

FFTW is a free C subroutine library for computing the discrete Fourier transform in one or more dimensions, of arbitrary input size, and of both real and complex data, as describes in the owner's web site <http://www.fftw.org/>.

The C function DFT.C below has been written on top of some of the principle calls to FFTW in order to create a function with relatively straight-forward arguments for a `⎕NA` call. It is therefore probably not still the Fastest Fourier Transform in the World, but it is nevertheless a very useful addendum to Dyalog APL.

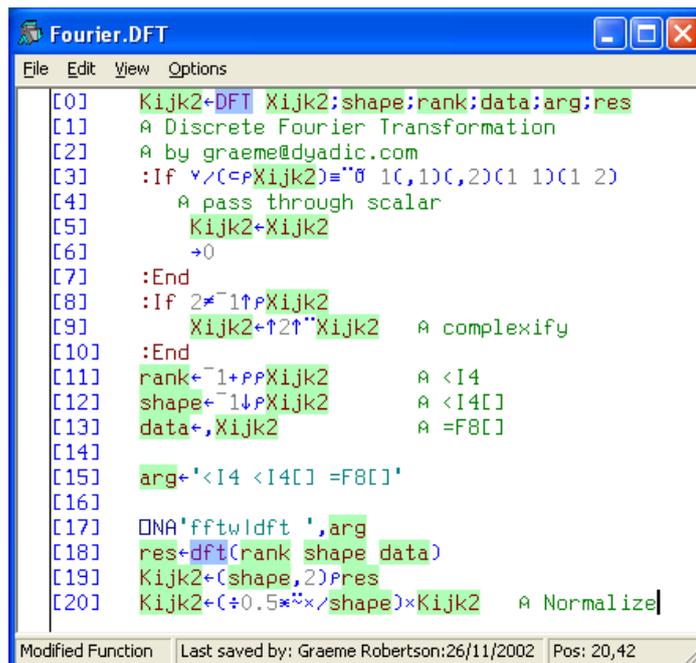
```
DFT.C
/* dft.c discrete fourier transform */

#include <fftw.h>

__declspec(dllexport) void dft(int *rank, const int *shape, double *data)
{
    fftwnd_plan plan;
    plan = fftwnd_create_plan(*rank, shape, FFTW_FORWARD, FFTW_IN_PLACE);
    fftwnd_one(plan, (void*)data, 0);
    fftwnd_destroy_plan(plan);
}
```

This function and the inverse function IDFT.C are to be found in the supplied file FFTW.DLL.

9.3.1.1 Given the above C function header, compose the right argument of `⎕NA` and compare with line [15] in the function below.



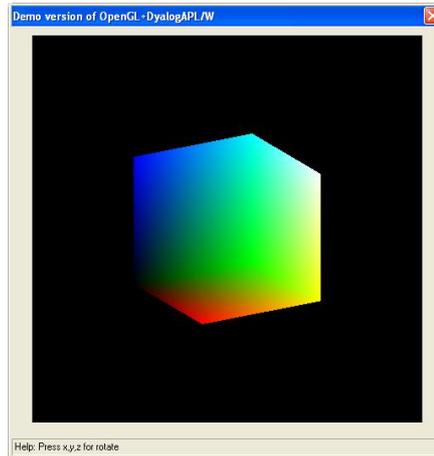
```

[0]  Kijk2←DFT Xijk2;shape;rank;data;arg;res
[1]  A Discrete Fourier Transformation
[2]  A by graeme@dyadic.com
[3]  :If ∨/⊖⊖Xijk2≠"0 1(,1)(,2)(1 1)(1 2)
[4]  A pass through scalar
[5]  Kijk2←Xijk2
[6]  →0
[7]  :End
[8]  :If 2≠⊖1↑Xijk2
[9]  Xijk2←⊖1↑Xijk2 A complexify
[10] :End
[11] rank←⊖1+⊖⊖Xijk2 A <I4
[12] shape←⊖1↓Xijk2 A <I4[]
[13] data←Xijk2 A =F8[]
[14]
[15] arg←'<I4 <I4[] =F8[]'
[16]
[17] ⎕NA'fftwldft ',arg
[18] res←dft(rank shape data)
[19] Kijk2←(shape,2)res
[20] Kijk2←(÷0.5×⊖/shape)×Kijk2 A Normalize
  
```

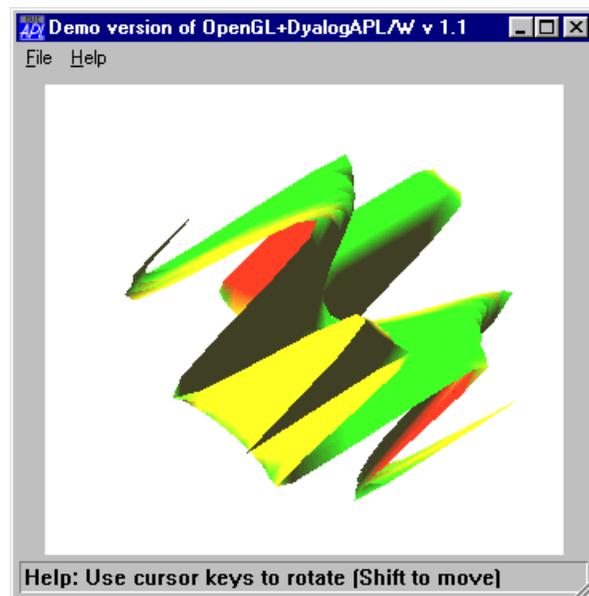
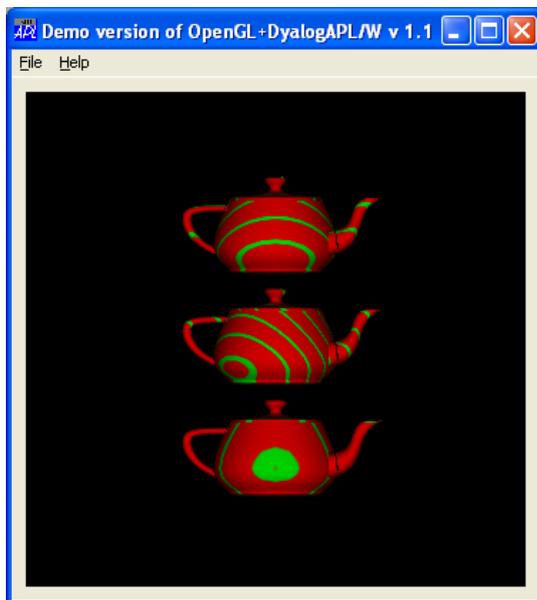
### §§ 9.3.2 Open Graphics Library

The OpenGL graphics library is an interesting application for  $\square NA$ . OpenGL is described in <http://www.opengl.org/documentation/>.

Alexander Skomorokhov, Alexei Zalivin and Alexander Kornilovski have kindly provided code that covers many of the OpenGL calls. Their workspace, DEMOGL.DWS, may be downloaded from the public download section of [www.dyalog.com](http://www.dyalog.com). It contains a number of static and dynamic examples.



Alexander Kornilovski has supplied a further workspace, GLAUX.DWS - also downloadable, and with some more complex examples.



9.3.2.1 Explore these workspaces, paying particular attention to the arguments of the 200 or so  $\square NA$  calls.

### §§ 9.3.3 Linear Algebra Package

There are many other freely available sources of code that can be accessed via  $\square NA$ . Some are single functions and some are large and complex applications, such as Petsc, the Portable, Extensible Toolkit for Scientific Computation, downloadable from <http://www-unix.mcs.anl.gov/petsc/petsc-2/>.

As a final example we consider LAPACK, the Linear Algebra Package which is downloadable free from <http://www.netlib.org/lapack/> (under GNU license agreement).

LAPACK contains hundreds of C functions for real and complex matrix manipulation. Most of these functions are defined in order to support 2 major goals. The main goal is general computation of eigenvectors and eigenvalues from real or complex square matrices. The other is the equivalent of monadic and dyadic domino ( $\boxplus$ ) for real *and complex* matrices.

A subset of these functions has been carefully chosen to cover the essentials of these two goals and thus provide the basis of two proposed new APL primitive functions which we denote  $\ominus$  and  $\boxminus$ .

See *APL81 Proceedings* for first mention of symbol  $\boxminus$  for this purpose.

Monadic  $\ominus$  is *complex matrix inverse* and dyadic  $\ominus$  is *complex matrix divide*. These functions are modelled by the ambivalent APL function, *Domino*, to be found in supplied workspace MATH.DWS. The convention adopted here is that complex numbers are represented by enclosed 2 element vectors.

The primary purpose of LAPACK is the calculation of monadic  $\boxminus$  (*eigen*) - the computation of the eigenvectors and eigenvalues of real or complex square matrices. This function is implemented in the APL function *Eigen* to be found in MATH.DWS. Complex numbers are again represented as enclosed 2 element vectors.

Given some complex square *i* by *i* matrix,  $A_{ii}$ , an **eigenvector** of  $A_{ii}$  is a vector whose direction is unchanged by the application (matrix multiplication) of  $A_{ii}$ . The corresponding **eigenvalue** is the scaling factor, which may be complex. In other words, given

$$E_{ji} \leftarrow \boxminus A_{ii}$$

then

$$\vdash(A_{ii} + .x \ 1 \ 0 \ \downarrow E_{ji}) \equiv \text{fuzz}(i \ i \ \rho E_{ji}[1;]) x \ \uparrow 1 \ 0 \ \downarrow E_{ji}$$

where *x* represents complex multiplication

```

    v R←A x W;Sign
[1]   Sign←2 2ρ1 1 1 1
[2]   R←+/Sign×0 1⊖(2↑A)∘.×2↑W v

```

and *fuzz* is a fuzzy operator to cope with a little algorithm inexactitude.

```

    v R←A(f fuzz)W;CT
[1]   CT←2*132 A=2.328E-10
[2]   R←(A+01)f W+01 v

```

Of particular scientific interest are Hermitian matrices (*H*), defined by  $\vdash H \equiv \phi \boxminus H$ , where  $\phi$  (or perhaps  $\epsilon$ ) might be defined as the complex conjugation primitive function. Hermitian matrices are important because their eigenvalues are **real** numbers, as are the results of all quantitative measurements. Thus Hermitian matrices are actually used to represent measurement operations in modern physics.

At this point let us consider an operator of a different character – a monistic niladic operator - that takes a matrix left operand and returns a related matrix result. These operators are intended to generalise to matrices of functions as outlined in § 11.3.

$Arr_2 \leftarrow Arr_1^T$	$\boxplus$ Transpose array $Arr_1$
$Arr_2 \leftarrow Arr_1^{-1}$	$\boxminus$ Inverse of array $Arr_1$
$Arr_2 \leftarrow Arr_1^*$	$\boxtimes$ Complex conjugate of array $Arr_1$

$$Arr_2 \leftarrow Arr_1^\dagger$$

A Complex transpose of  $Arr_1$

The function  $\nabla EV \nabla$  below is a canonical version of the function  $\nabla Eigen \nabla$  in MATH.DWS. Line [11] calls function  $\nabla ZHEEV \nabla$ .

```

Eigen.EV
File Edit View Options
[0] Eji2←EU Aii2 A Eigenvalues;Eigenvectors of Aii
[1] A by graeme@dyadic.com
[2] :If real Aii2
[3]   :If symmetric Aii2
[4]   :AndIf 2≠P Aii2
[5]     Eji2←DSYEU Aii2 A real symmetric
[6]   :Else
[7]     Eji2←DGEEU Aii2 A real non-symmetric
[8]   :EndIf
[9] :Else
[10]  :If hermitian Aii2
[11]    Eji2←ZHEEU Aii2 A complex hermitian
[12]  :Else
[13]    Eji2←ZGEEU Aii2 A complex non-hermitian
[14]  :EndIf
[15] :EndIf
Modified Function Last saved by: Dyadic:06 August 1999 11:04:26 Pos: 0,8

```

$\nabla ZHEEV \nabla$  is a cover function for the LAPACK function  $zh\text{eev}_\text{}$  which is fixed from the supplied LAPACK.DLL on line [19] of  $\nabla ZHEEV \nabla$  and called on line [22].

```

Eigen.ZHEEV
File Edit View Options
[0] Eji2←ZHEEU Aii2;arg;res;Ei;Eii;re;pair
[1] A complex hermitian matrix eigenvalues/vectors
[2] A Aij Ejk = Ek Ejk A Ek ∈ Real
[3] A Aii +.× Eik = Ek ×[2] Eik A i=j=k=1..N
[4] DSHADOW'JOBZ UPLO N A LDA W WORK LWORK RWORK INFO'
[5]
[6] JOBZ←'U' A1 <C1
[7] UPLO←'L' A2 <C1
[8] N←P Aii2 A3 <I4
[9] A←,2 1 3@Aii2 A4 =F8[]
[10] LDA←N A5 <I4
[11] W←N A6 >F8[]
[12] WORK←2×(2×N)-1A2×LWORK A7 >F8[]
[13] LWORK←(2×N)-1 A8 <I4
[14] RWORK←(3×N)-2 A9 >F8[]
[15] INFO←0 A10 >I4
[16]
[17] :If 0=DNC'zheev_'
[18]   arg←'<C1 <C1 <I4 =F8[] <I4 >F8[] >F8[] <I4 >F8[] >I4'
[19]   DNA'lapack.dll|zheev_',arg
[20] :End
[21] :Trap 11
[22] res←zheev_(JOBZ UPLO N A LDA W WORK LWORK RWORK INFO)
[23] :Else
[24]   ('unknown error')DSIGNAL 11 A→ try ZGEEU
[25] :End
[26] :If 0≠1↑res
[27]   ('error ',#↑1↑res)DSIGNAL 11
[28] :End
[29] Ei←↑2↑'',2>res
[30] Eii←2 1 3@N N 2p1>res
[31] Eji2←Ei;Eii
Modified Function Last saved by: Graeme Robertson:26 November 2002 12:10:28 Pos: 1,48

```

9.3.3.1 Compare the `⊠NA` call in `▽ZHEEV[19]▽` with the snippet of C code in file ZHEEV.C below taken from the corresponding uncompiled LAPACK function `zheev_`.

```
ZHEEV.C
#include "f2c.h"

/* Subroutine */ int zheev_(char *jobz, char *uplo, integer *n, doublecomplex
    *a, integer *lda, doublereal *w, doublecomplex *work, integer *lwork,
    doublereal *rwork, integer *info)
{
/* -- LAPACK driver routine (version 2.0) --
   Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
   Courant Institute, Argonne National Lab, and Rice University
   September 30, 1994
...

```

Specification of the right argument to `⊠NA` can be arbitrarily complex. Errors in the specification can cause Dyalog APL to crash with a System Error 999 and therefore care should be taken when constructing `⊠NA` calls, although version 11 has significantly improved the error reports from `⊠NA`.

Without a left argument `⊠NA` fixes a function whose name is that of the C function involved. Alternatively, `⊠NA` can take a left argument of a character string containing any legal user-defined name for the function to be fixed in the workspace.

9.3.3.2 Ask for the next module on **runtime applications** 😊.

## Module10: Stand-Alone Applications

You have written your Dyalog APL application and all you want to do now is dish it out. "HOW DO YOU DO THAT?"

### § 10.1 Building GUI Applications

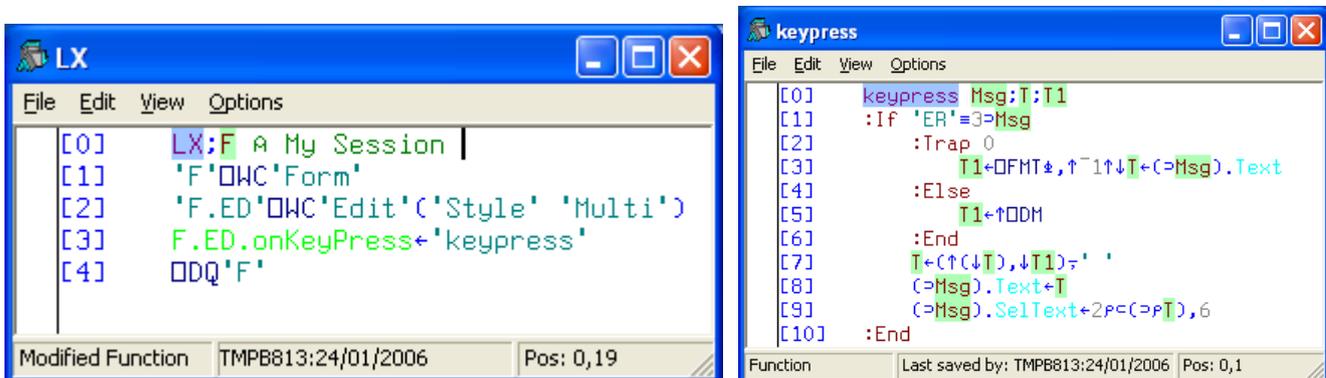
#### §§ 10.1.1 The bare Minimum

Your application code may be in a workspace and the interpreter may be the runtime executable, Dyalogrt.exe. For details of this outdated scenario, see the document "Run-Time Applications in Dyalog APL/W Versions 7 and 8" which is available from the DyalogUsers yahoo repository. This document also applies, essentially unchanged, to version 9.

From version 10 onwards, the smallest possible packaged Dyalog APL application requires two files, an executable (EXE) file containing your code and the Dyalog APL dynamic link library, Dyalog10rt.DLL, to which it is bound. Together, these two files can constitute a complete application whose only reference to APL is the name of the Dyalog DLL. The application may use its own registry section and an application icon can be incorporated into the executable. This is explained in §10.2.

The program should trap all errors, but just in case the program should exit before `OFF`, the configuration parameters `RunTimeTitle` and `RunTimeError` ought to be given appropriate entries in your registry file. This registry file will be described in §§10.2.2.

In this section we prepare an application for export. The runtime application agreement with Dyalog Limited clearly precludes the possibility of writing a Session-replacement cover for the Dyalog APL language kernel such as `LX ↦ F ↦ keypress` that executes ( $\pm$ ) raw APL code below.

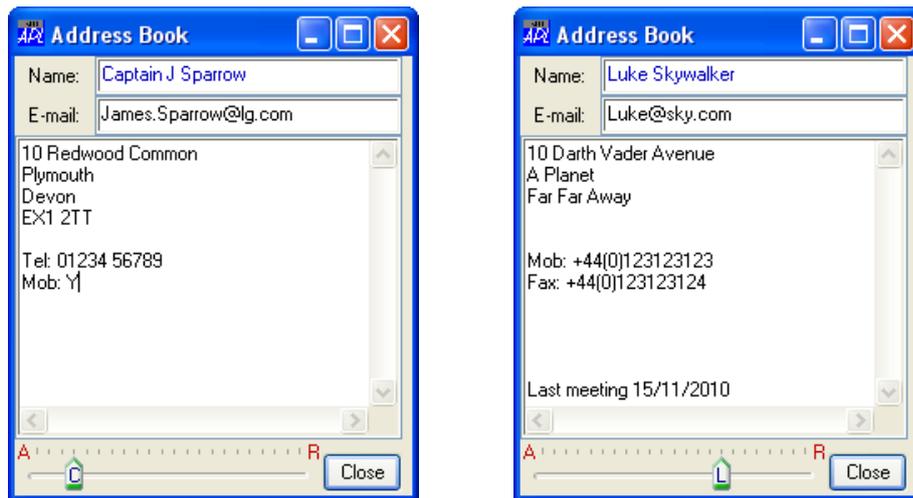


So we need an *IDEA!* for an application of our own. To assist in the quest we provide a small starter application that you might adopt and adapt to your liking.

10.1.1.1 Load workspace `AddrBk.DWS` and try using the address book. Add a new entry by dragging the `TrackBar` to the far right and entering a name. Drag the `TrackBar` back to the appropriate letter to view your new entry. Edit the entry at will.

Note: Remove mention of `ShowsIP`, `OKButton` and `SIPResize` to run on a PC.

Two of your friends and associates might be:



### §§ 10.1.2 Completing the Address Book Application

You might have noticed that your entries are lost as soon as you exit the workspace ☹.

10.1.2.1 In order to make this little application useful, replace the functions `vaddrbk.GetMyStorev` and `vaddrbk.PutMyStorev` with APL component file read and write/create storage functions.

### §§ 10.1.3 Enhancing your Address Book

There are many other ways that you might wish to personalise and improve the Address Book application:

- Use a different extension (eg .AB) rather than .DCF for your component file name.
- Include `↵FileBox` in `[File][Open]` to select a different address book.
- Add `□TRAP ↵ErrorLog` to deal with errors. (Start `ErrorLog[1]` with `□TRAP←0 'S'`.)
- And/or email errors to your HQ (see §§7.3.1) with information such as..  
`□AN □AI □DM □SI □XSI □TS □FNAMES □WA □TID ..`, but note `≠□WSID≡''` in an EXE.
- Create a Help file and add a `[Help][Help]` menu item.
- Add `[Help][About]` to identify yourself and your product.
- Change the system `Icon` defined in `vaddrbk.Icons32v` to your own design.  
 Tip: Consider using supplied workspace BMED.DWS.
- Make the left and right arrow keys move the `TrackBar Thumb` left and right. Etc...

## § 10.2: Making runtime Executables

Up until version 10, you could build *free* stand-alone applications with Dyalog APL. (In version 11 the runtime interpreter is *not free*.) An important practical difference between version 10 and previous versions is that you can completely avoid the difficulties involved in correctly specifying the command line required to initialise a version 9 (and prior) run-time system.

### §§ 10.2.1 Files to Include

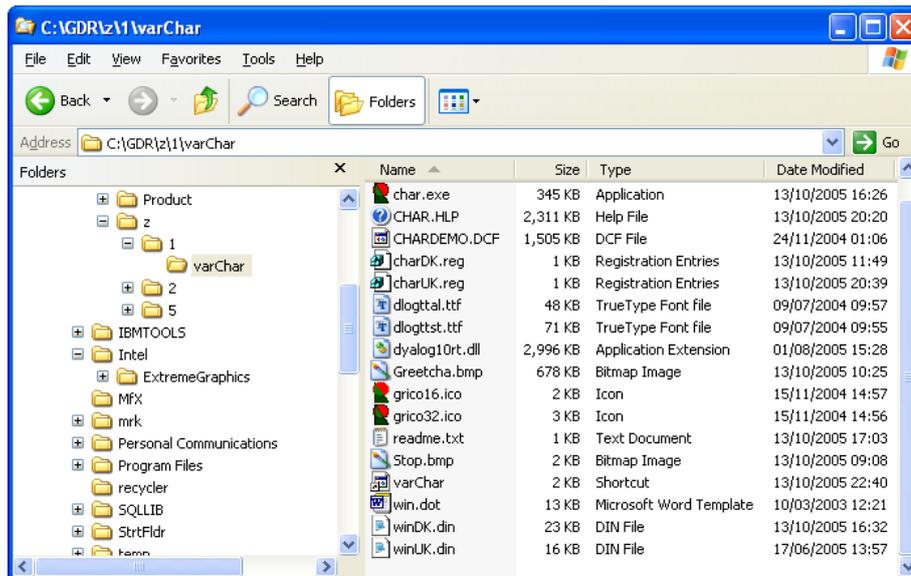
The only files that you have to include with a minimal runtime application in versions 10 and 11 are the application executable and the bound Dyalog runtime DLL. There is no need to include input (.DIN), output (.DOT) or APL font files, unless the application is explicitly going to use APL characters.

It is, however, often mandatory to include a registry (.REG) file, described in the next section.

Various other files are often necessary:

- APL component files or native files for data storage,
- the application's help file,
- application-specific bitmaps and icons.

An example version 10.0 run-time application called **varChar** is included in this APL3&4 course. The files involved in this application are shown below. **Char.exe** with **Dyalog10rt.dll** is the core of the system. There are registry files and corresponding keyboard files for Danish and UK users. There is a demo DCF file and a help file. There are odd font, bitmap and icon files.



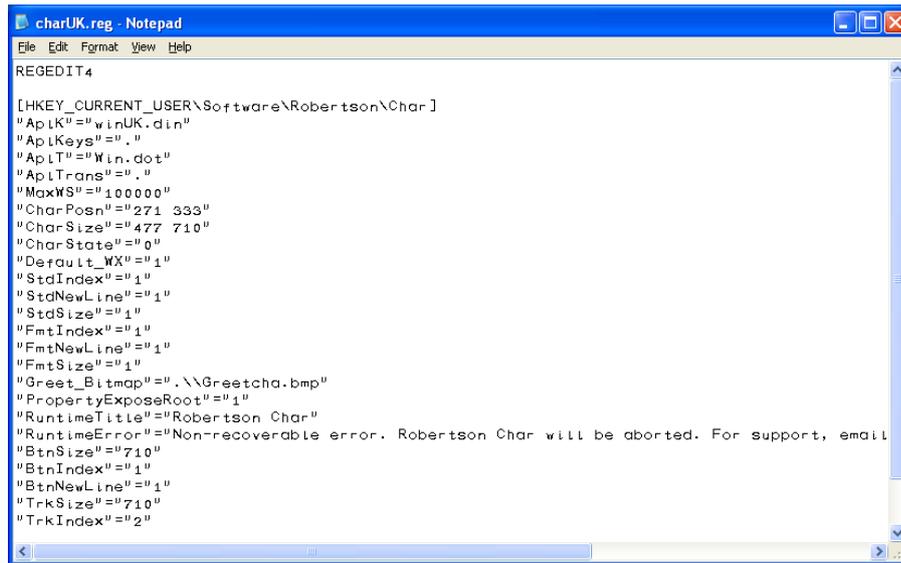
Tip: If your keyboard is other than Danish or UK then you should copy and rename your keyboard (.DIN) file to be winUK.din and copy it over the one in **varChar**.

## §§ 10.2.2 Infiles and the Windows Registry

The Infile parameter refers to a registry subkey in the HKEY\_CURRENT\_USER section of the Windows Registry. The default subkey in version 10.0 is \Software\Dyadic\Dyalog APL/W 10.0 but you can choose your own names to put under the \Software\ section.

REG files, which are simple text files and can therefore be edited in Notepad, should begin with REGEDIT4 (or Windows Registry Editor Version 5.00) to identify the file as containing registry information. The next line identifies the section into which the information to follow should be added. If this section does not exist then it will be created. The name of the section typically contains the company name and the application name, *eg* [HKEY\_CURRENT\_USER\Software\MyCompany\MyApplication]

10.2.2.1 Use REGEDIT.EXE [File][Export] to export a small section of your registry. View the resulting file in Notepad. Note that names and values of string entries (of type REG\_SZ) are surrounded by double-quotes. Note also that backslash characters (\) have to be doubled up (\\) inside quotes.



To install information from a REG file into the registry, you simply need to double-click on the file in Windows Explorer. Alternatively you can run command

```
regedit c:\myapp\myapp.reg
```

in the command prompt. Or you can run the command

```
regedit /s c:\myapp\myapp.reg
```

to silently install the entry (without the popup message boxes). (The /d switch deletes a key.)

This can be done in APL under program control using `⎕CMD` (or WinExec), first making a call to `#.GetEnvironment 'Inifile'` to check whether some particular registry entry already exists. This can make the whole process of establishing a registry entry completely transparent to the user.

the MAXWS parameter used to be one of the most significant entries in the registry file. MAXWS determined the maximum size of the workspace and often had to be set to a large value for big applications. However, the DLL now dynamically allocates more memory as required and so this parameter is less important from version 10 onwards. (Also note that `⎕WSID` is **empty** in a .EXE file.)

One simple but nevertheless valuable parameter is `greet_bitmap`. If this parameter is set to the name of a bitmap file (eg "greet\_bitmap"="c:\\myapp\\mybmp.bmp") then that bitmap will be displayed while the workspace is loading, and until the statement `#.GreetBitmap 0` is executed in your `⎕LX` after everything has been initialised and your application is ready for user activity. Sometimes this is too quick!

10.2.2.2 Make a 200 by 400 pixel bitmap (using the `FileWrite` method of a `Bitmap` object) and make a 32 by 32 pixel icon (using the `FileWrite` method of an `Icon` object.)

10.2.2.3 Create a personalised registry file with the `greet_bitmap` parameter set to your product bitmap file.

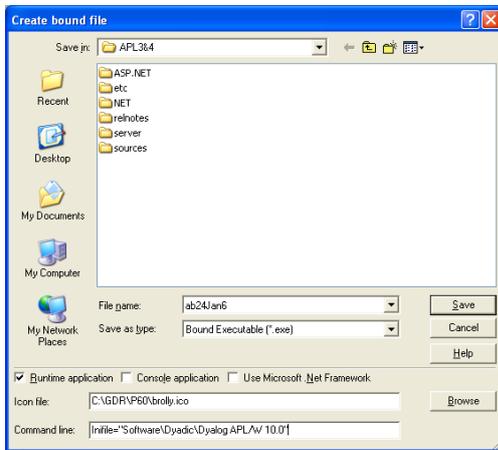
10.2.2.4 Add `⎕CMD'RegEdit /s ...'` to the Address Book `⎕LX`, to be run if and only if

```
⌈'C:\\.\\MyBitmap.bmp'###.GetEnvironment'greet_bitmap'
```

10.2.2.5 Add the statement `#.GreetBitmap 0` to the Address Book `⎕LX` at a suitable point.

### §§ 10.2.3 The [File][Export] Menuitem

10.2.3.1 XLoad your Address Book workspace in Dyalog version 10. Run [File][Export...]. Uncheck [Use Microsoft .Net Framework]. Enter your icon file name under *Icon file* and Infile under *Command line*.



On *Save* you have an executable program bound to a dynamic link library. If the DLL is placed in the \Windows\System32\ directory then it should always be visible to the executable, otherwise it should be copied and moved around with the EXE file (that was your workspace).

10.2.3.2 Test your product and sell it to your friends ;-)

## § 10.3 Aspects of Pocket APL

### §§ 10.3.1 Pocket Platforms

Dyalog Pocket APL runs under Microsoft Pocket PC 2002 and 2003 operating systems which run on a number of Personal Digital Assistants (PDAs) including Toshiba e740 and HP Jornada 560 Series.



← Toshiba e740



← HP Jornada 560 Series

For a current list of suitable devices, see <http://www.dyalog.com/> [Products][Pocket Dyalog]. The Dyalog web site and *Dyalog APL Object Reference* are the primary sources of documentation for Pocket APL.



## §§ 10.3.2 Creating the executable Program

The Address Book GUI was built using the WDESIGN workspace. You might like to modify the Address Book GUI using WDESIGN. To do this you should follow the instructions in the *Dyalog APL User Guide* where a brief description of the distributed WDESIGN workspace is given.

10.3.2.1 Remove the comment symbols on lines [1] and [3] of `▽addrbk.RUN▽` and save the workspace.

10.3.2.2 XLoad the workspace in **Pocket APL** on your **iPAQ** and select [File][Make Executable] to create an executable file. Tick the [Runtime Executable] checkbox and untick the [Evaluation Executable] checkbox to create a basic product.

## §§ 10.3.3 Building a distributable Application

A pocket APL runtime application requires the Pocket APL Runtime Engine, Dyalog10.DLL, which should be installed in the PDA Windows directory. This Pocket APL Runtime Engine must be purchased by the user of your application from [www.handango.com](http://www.handango.com) for around \$5.

Instructions as to how you can package and distribute an application yourself through [www.handango.com](http://www.handango.com) is give in [www.dyalog.com](http://www.dyalog.com) [Products][Pocket Dyalog].

Apart from the APL runtime DLL, all that you need to distribute is the system executable (plus any other extraneous files required by your application, such as supporting APL component files).

Note if you are an individual holding personal information only for domestic reasons (eg an address book or Christmas card list) then you are not required to comply with the UK Data Protection Act 1998. Phew! ☺.

10.3.3.1 Consider joining the [pocketapl@dyalog.com](mailto:pocketapl@dyalog.com) mailbox group. That is the end of Day 1. Please come back for more tomorrow. ☺

## Module11: Advanced Dot Syntax

### § 11.1 Object Variables

#### §§ 11.1.1 Stranding Object Properties

##### §§§ 11.1.1.1 Stranded Vectors

Since the advent of *floating array* second generation APLs, which we have described generically as APL 2 in APL1&2.PDF, variables (and unnamed parenthesised expressions) may be stranded together to form nested vectors of enclosed elements simply by juxtaposing array expressions.

**Strand (Vector) Notation:** A series of two or more adjacent array expressions results in a vector whose elements are the enclosed arrays resulting from each expression. (See *Language Reference* p12.)

Aside: In *fixed array* second generation APLs, pioneered by **Ken Iverson** in **SharpAPL** and **J**, strand notation is entirely avoided. Instead a new canonical primitive function, *link* (:), is introduced which encloses the Larg and catenates the result to Rarg, enclosing Rarg if it isn't already a vector of enclosed elements. In *fixed* notation, enclosing a scalar **is** possible.

Given three variables *f* for first, *s* for second and *t* for third,

```

f←43564          a Numeric scalar (NumSc)
s←'sdgg' 'sg' 'sgsg' a Vector of character vectors (VecCharVec)
t←3 3ρ1         a Numeric matrix (NumMat)
f s t          a Strand (VecEncArr)
43564  sdgg  sg  sgsg  1 1 1
                        1 1 1
                        1 1 1
  
```

Strand notation has been generalised to *strand assignment*. The above 3 assignments can be achieved in one single statement:

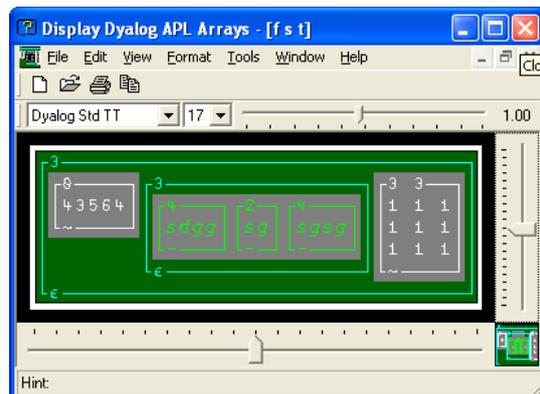
```

f s t←43564('sdgg' 'sg' 'sgsg')(3 3ρ1)
DISPLAY f s t          a Display view of strand
  
```

```

.→-----
| 43564 | .→---. .→-. .→---. | ↓1 1 1 | | | | | | | | |
|       | |sdgg| |sg| |sgsg| | |1 1 1| |
|       | |-----' |-----' | |1 1 1| |
|       | 'ε-----' | '~-----' |
|ε-----
  
```

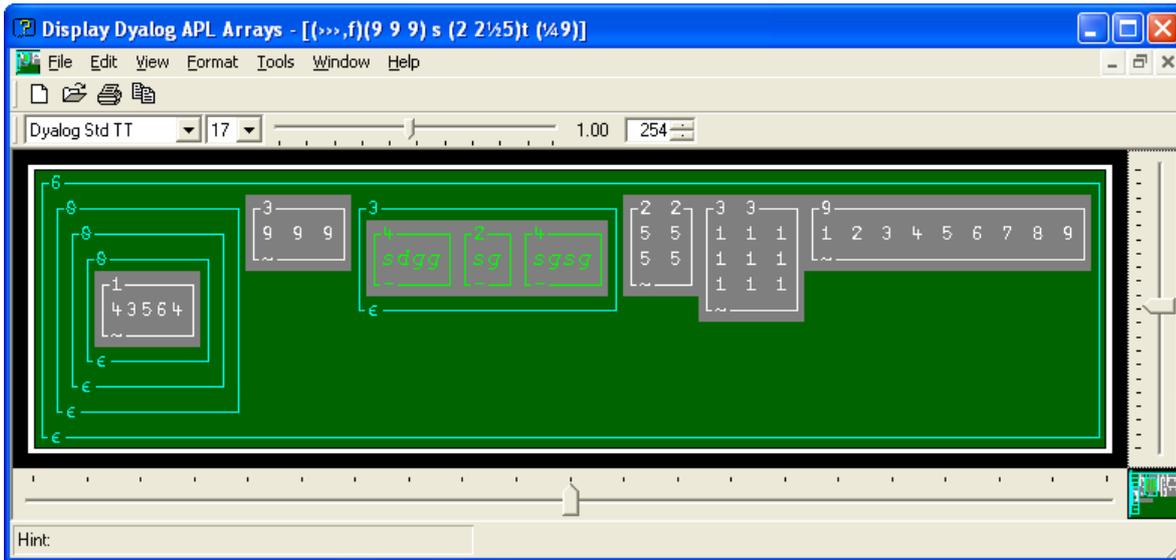
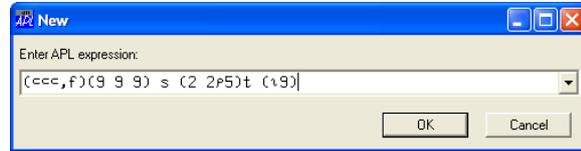
The **varChar** view of this strand is:



The resulting display shows clearly that the strand *f s t* results in a three element nested vector of which the first element is a simple numeric scalar, the second element is a vector of character vectors and the third element is a simple numeric matrix.

11.1.1.1 Produce the example below in **varChar** and describe the elements of the six element stranded vector. Experiment with other strands.

```
(ccc,f)(9 9 9)s(2 2ρ5)t(19)
```



Since object properties are essentially variables residing in the object's space, we should be able to strand these into vectors of properties, and indeed we can.

11.1.1.2 Create a *Form*, *F*, and enter *F*-space. List its properties. Strand some of its properties, eg

```
ρ←Accelerator AcceptFiles Active AlphaBlend AutoConf  
0 0 0 1 256 3  
5
```

Assign a new *Posn* and *Size* in a single expression.

It often happens, especially in a C code environment, such as in `□NA` function calls, that arguments to and results of functions consist of amorphous strands of disparate variables. This general scenario is such a common occurrence in application code that, as of Dyalog version 10.1, it is possible to use **extended function header syntax**. With this syntax, the header line of a function may use strands of variable names in place of single names for arguments and results. For example a function that takes a date *Rarg* and returns the next day's date could have header line:

```
▽(Day Month h Year)←nextDay(Day Month h Year)▽
```

§§§ 11.1.1.2 Vectors of .. Vectors of Stranded Name Vectors

```
$V..V←SV..VArr      ⍲ Assigns nested array SV..VArr to a compatible strand
```

Assignment is extended to pervade nested strands of names ( $S^{V..V}$ ) to the left of the arrow. This allows multiple naming of parts of a structure in a single assignment. This name structure ( $S^{V..V}$ ) is entirely constructed of lists of names. The depth of the overall vector structure ( $V..V$ ) is determined by the placing

of parentheses. We give this new form of name-structure the generic symbol,  $\S^{V..V}$ , to indicate it is a vector-name-strand structure of arbitrary depth.

The conformability rules are similar to those for scalar (pervasive) dyadic primitive functions such as '+'. If  $a\ b\ c$  and  $d$  are stranded names to left of an assignment arrow then the data on the right of the assignment arrow must either be a scalar, in which case it is extended to conform to the stranded structure, or a vector whose structure reflects the stranded structure of the names on the left.

```
(a b)(c d)←(1 2)(3 4)⇒ ((a←1)(b←2))((c←3)(d←4))
((io ml)vec)←0 av ⇒ ((io←0)(ml←0))(vec←av)
(a(b(c d)e)f g)←1(2(3 4)5)6 7 ⇒ a b c d e f g←17
(a(b(c d)e)f g)←1(2((3 3)(4 4))(c3 3p5))6 7
```

If a simple scalar is encountered at an earlier stage in the correspondence, then that scalar is extended to cover the structure beneath it, as is the case below:

```
(a b)(c(d(e f)))←1 (3 4)
```

This is a way of creating named arrays from a structure corresponding to vectors of .. vectors of names.

```
((first last) sex (street city country))←DATA
```

Each name in the structure may be space-qualified so we can step from

```
(first last)←'Star' 'Chakahwata'
```

to

```
(A.first A.last)←'Star' 'Chakahwata'
```

where

```
A.first↳'Star'
A.last↳'Chakahwata'
```

This is another step in the logic behind the extension of second to third generation APL notation.

### §§§ 11.1.1.3 Name Strands in :For Loops

The :For loop control structure allows multiple control variables using distributed assignment.

```
:For  $\S^{V..V}$  :In VecC $\S^{V..V}$ Arr◇.. $\S$ ...◇:End ⌘ Do.. $\S$ ... for element C of VecC $\S^{V..V}$ Arr
```

In this case the :For statement loops round once for each of the C elements of Vec<sub>C</sub> $\S^{V..V}$ Arr. C symbolises the notional loop counter. (Vec<sub>C</sub> $\S^{V..V}$ Arr is ravelled if it is not already a vector.) The structure of each element ( $\S^{V..V}$ Arr) should be compatible with the name strand structure  $\S^{V..V}$ , in which the (V..V) indicates an arbitrary depth vector of vectors .. of vectors. So the vector structure of names  $\S^{V..V}$  reflects the vector structure of each element of vector Vec<sub>C</sub> $\S^{V..V}$ Arr. The expression or expressions within the :For loop will most probably make reference to some of the names from the nested structure of names  $\S^{V..V}$  and this likely scenario is indicated symbolically by .. $\S$ ... .

For example,

```
:For a b c :In (1 2 3)(3 4 5)(5 6 7)(7 8 9) ⌘ In this case C=4
  a b c ⌘ on first loop |a b c≡1 2 3
:EndFor
```

OR

```
:For a(b(c d)) :In (1(2(3 4)))(5(6(7 8)))(15(16(17(2 2p18)))) ⌘ C=3
  a b c d ⌘ on first loop |a b c d≡1 2 3 4
:End
```

Alternatively,

```
:For §V..V :InEach SV..VVecCArr ⋄..§...⋄:End   ⌘ Do ..§... for each C in SV..VVecCArr
```

In this case, on the C<sup>th</sup> loop a strand consisting of the C<sup>th</sup> element of each of the vectors in S<sup>V..V</sup>Vec<sub>C</sub>Arr should have a structure compatible with the structure of the vector of vectors .. of vectors of names in §<sup>V..V</sup>. (In this version of the :For loop, each element in the data vector is more likely to have a uniform structure.)

```
:For a b c :InEach (1 3 5 7)(2 4 6 8)(3 5 7 9)   ⌘ In this case C=4
      a b c   ⌘ on first loop |a b c≡1 2 3
:EndFor
```

Formally, the alternatives may be presented as a table:

:For	§	:In	Vec <sub>C</sub> Arr
:For	§ <sup>V</sup>	:In	Vec <sub>C</sub> S <sup>V</sup> Arr
:For	§ <sup>V..V</sup>	:In	Vec <sub>C</sub> S <sup>V..V</sup> Arr
:For	§	:InEach	Vec <sub>C</sub> Arr
:For	§ <sup>V</sup>	:InEach	S <sup>V</sup> Vec <sub>C</sub> Arr
:For	§ <sup>V..V</sup>	:InEach	S <sup>V..V</sup> Vec <sub>C</sub> Arr

A corresponding table of examples may be written:

:For	aa	:In	A <sub>1</sub> A <sub>2</sub> ... ⇒ aa←A <sub>i</sub> on i <sup>th</sup> loop round
:For	aa bb	:In	(A <sub>1</sub> B <sub>1</sub> )(A <sub>2</sub> B <sub>2</sub> ) ... ⇒ aa bb←A <sub>i</sub> B <sub>i</sub>
:For	aa(bb cc)	:In	(A <sub>1</sub> (B <sub>1</sub> C <sub>1</sub> ))(A <sub>2</sub> (B <sub>2</sub> C <sub>2</sub> )) ...
:For	aa	:InEach	A <sub>1</sub> A <sub>2</sub> ... ⇒ aa←A <sub>i</sub> on i <sup>th</sup> loop round
:For	aa bb	:InEach	(A <sub>1</sub> A <sub>2</sub> ...)(B <sub>1</sub> B <sub>2</sub> ...) ⇒ aa bb←A <sub>i</sub> B <sub>i</sub>
:For	aa(bb cc)	:InEach	((A <sub>1</sub> A <sub>2</sub> ...)((B <sub>1</sub> B <sub>2</sub> )(C <sub>1</sub> C <sub>2</sub> )...))

where aa bb and cc are valid variable names and A<sub>1</sub> B<sub>2</sub> etc.. are arrays to be assigned.

11.1.1.3.1 If aa bb cc are replaced by Posn, Size and Caption properties of a Form, compare the data required in :In and :InEach loops.

## §§ 11.1.2 Stranding Objects

### §§§ 11.1.2.1 Pure Vectors of Namespace Objects

GUI objects are namespaces but namespaces are not necessarily GUI objects. The term namespace, or space, covers both flavours. Spaces are like variables and can be arguments and/or results of functions.

In Dyalog APL there are numeric variables, character variables, and object variables. The strand

```

ρ←# SE
# SE
2

```

exemplifies a simple 2 element pure vector of spaces. Notice that with the introduction of vectors of objects naturally comes the subtle generalisation of the primitive function *shape* ( $\rho$ ).

```
IntVec←ρRVec
```

⌘ Shape of vector containing references to spaces

11.1.2.1.1 Create a 3 element strand of GUI objects.

11.1.2.1.2 Write a dummy function such as

```
▽ enterMouse Msg
```

```
[1] Msg ▽
```

Attach this callback to the *MouseEnter Event* of a *Form* either by

```
Event←'onMouseEnter' 'enterMouse'
```

or, better, by

```
onMouseEnter←'enterMouse'
```

but not by superseded statements such as

```
'F'⊂WS'Event' 'onMouseEnter' 'enterMouse'
```

```
'F'⊂WS'Event' 'MouseEnter' 'enterMouse'
```

or

```
Event←'MouseEnter' 'enterMouse'
```

Verify by tracing into the callback that ( $\Rightarrow$ Msg) is an object reference (of dataType *RSc*). Notice the natural generalisation of *pick* ( $\Rightarrow$ ) to apply to a vector containing refs.

```
RSc←⇒RVec
```

⌘ Discloses first element (given  $\vdash ML < 2$ )

Use of the event prefix "on", as described in §§2.2.2, causes the first element in the message (*Msg*), which is automatically generated for the right argument to any callback, to be a ref to the object rather than a character vector containing the name of the object.

The distinction between the name of an object and a ref to the object is clearly important. The distinction between the name of an array and the array itself is also important in understanding the *data representation* system function  $\vdash DR$ .  $\vdash DR$  applies to the array itself. A ref might or might not be named.

An object may have many *named* references to itself and so the real name of the object becomes a moot point. The question is analogous to the Platonic question as to which is the 'real' number 1, *A* or *B*, in the expression  $A \leftarrow B + 1$ . Indeed, is *A* numeric, or just a named reference to a number? Is a named ref less valid than the name given to an object in  $\vdash WC$ ? We argue from simplicity that it should not be.

Notice that *Msg* appears to be a normal APL 2 nested vector, but actually two out of three of its elements are objects. (*Msg* contains dataTypes *RSc CVec RSc*.) APL 3 vectors can be composed of numbers, characters *and* refs. Henceforth we shall assume a named ref to be a proper name of an object.

The following lines place 100 *Buttons* on a *Form*, position them and set the individual *Captions*.

```

'F'⊂WC'Form'
FV←⊕('F', '( $\vdash$ .B'), '⊕'⊕100)⊂WC'Button'
FV.Posn←,45×⊕10 10
FV.Caption←⊕100 3ρ⊂A

```



11.1.2.3.1 Are the 3 spaces created by ( $\square NS'' \ni \rho c' ' ) \square \square \square$  identical or just similar (isomorphic)?

The `:With` control structure accepts a ref, `RSc` (which can be a *collection*), or a string, `CVec`, containing the name of a space. It also, therefore, accepts an unnamed namespace. This can be useful for localising more or less complex lines of code.

```
:With □NS''
  CVec←'This string is ephemeral.'
:End
```

`:For` also applies to *collection* objects as found, for example, in Word and Excel. Collection objects encourage irregular syntax in VB because they always have an `Item` method and this method name may be elided in VB code. Thus **Application.Workbooks (1)** in VBA works exactly the same as **Application.Workbooks.Item (1)**. This simplified syntax is emulated in `:For` as applied to a collection wherein each *Item* in *Count* is automatically instantiated sequentially in the loop. (Dyalog version 11.0 goes even further in incorporating this VB anomaly.)

```
:For ඩ්වය :In Documents ෦ Documents is a Word Collection Object
  ඩ්වය.Name
:End
```

```
:For ඒඩ්වය :In Sheets ෦ Sheets is an Excel Collection Object
  ඒඩ්වය.Name
:End
```

### §§ 11.1.3 Arrays of .. Arrays of Objects

#### §§§ 11.1.3.1 Reshaping Object Vectors

Generalisations of the APL primitive function *pick* ( $\triangleright$ ) to select an object from a nested vector, or *shape* ( $\rho$ ) to obtain the shape of a vector of refs, are natural extensions that can go almost unnoticed.

Object spaces are essentially a new type of scalar since  $'F' \square WC' Form' \diamond \theta \equiv \rho F \downarrow 1$  (but  $\square NC' F' \downarrow 9!$ ). We can now assign objects to names,  $G \leftarrow F$ , including strands,  $FGF \leftarrow F G F$ , and then  $\square NC' FGF' \downarrow 2$  which is more comprehensible than name class 9.

```
NAME←RARR ෦ Creates name for object reference array
```

Beware of the fact that assignment cannot change the class of an existing variable (although this is ameliorated in version 11). Therefore if `G` had already existed as a class 2 object then assignment  $G \leftarrow F$  above would have given a **SYNTAX ERROR**.

The extension of dyadic *reshape* ( $\rho$ ) is harder to miss than that of monadic *shape* because it allows you to generate matrices and higher rank arrays of objects. Indeed, you can make arbitrary nested arrays of arrays of .. arrays of object spaces, numbers and characters.

```
RARR2←IVecρRARR1 ෦ Reshapes array containing references to spaces
```

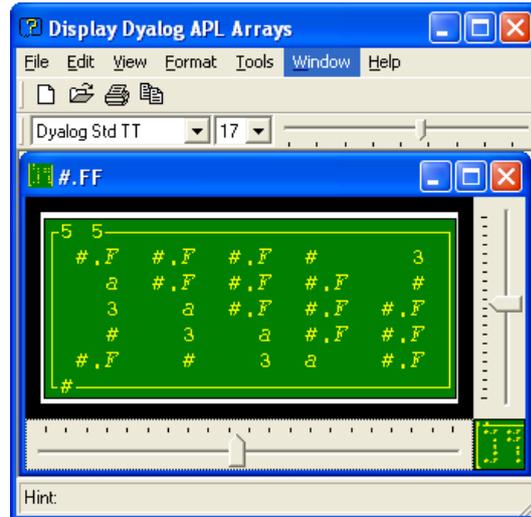
For example, consider the matrix

```
FF←5 5ρF G F # 3 'a' ෦ Simple mixed RefArr
```

In **varChar**, *FF* displays as shown. Notice that the display form of *G* is *#.F*. This is the case even after *F* has been erased. *F* and *G* are references to the same object.

11.1.3.1.1 By creating a variable inside *F*-space, and examining the contents of *G*-space, demonstrate that *F* and *G* are not separate objects. Expunge the *Form* by expunging **all** references to it.

Aside: There could be 11 different colour codings for the different types of mixtures. Currently, **varChar** just picks the colour of the first element for exotic mixtures.



*Dyadic*  $\square NS$ , with an empty right argument, returns the full path name of the new (or existing) space.

$CVec_2 \leftarrow CVec_1 \square NS()$	Ⓐ Returns the full name of space, named in $CVec_1$
---	---

Note  $\vdash () \equiv \emptyset$

11.1.3.1.2 Create a rank 3 array of unnamed, and another of named, namespaces.

Unless an unnamed namespace (or array of unnamed namespaces such as  $2 \ 2 \rho \square NS''c''$ ) is given a name in some way then it evaporates to nothing on completion of the statement in which it was created. However, if spaces are named by some reference on assignment,  $RA \leftarrow 2 \ 2 \rho \square NS''c''$ , or individually as in

```
2 2 ρ 'ABCD' □ NS''c''
```

```
#.A  #.B  
#.C  #.D
```

then, like GUI spaces that are named on creation, as in for example,

```
ρ''EFGH' □ WC''c' Form' ↪ 3 3 3 3
```

they each do not evaporate until explicitly expunged by some particular action or mechanism.

### §§§ 11.1.3.2 Generalised Primitives

A number of other primitive and system functions have been extended to handle space arrays. Where appropriate, they take arguments of arrays containing spaces and return arrays containing spaces.

$CMat \leftarrow \overline{\varphi} RArr$	Ⓐ Returns <i>CMat</i> of character display forms
---	--

$RArr_2 \leftarrow IArr \triangleright RArr_1$	Ⓐ Picks from array (depending on $\square IO$ )
--	---

$EncSc \leftarrow c RArr$	Ⓐ Encloses an array containing references to spaces
---------------------------	---

$VecEncRArr \leftarrow BVec c RArr$	Ⓐ Vector of enclosed arrays of refs (given $\vdash \square ML < 3$ )
-------------------------------------	--

$RArr_2 \leftarrow \uparrow RArr_1$	Ⓐ Mixes nested array to higher rank (given $\vdash \square ML < 2$ )
-------------------------------------	--

$RArr_2 \leftarrow IntVec \uparrow RArr_1$	Ⓐ Takes (not yet overtakes) from array containing refs
--	--

For overtake, we would need an *identity element*, eg @, for the *take* function when applied to spaces.

$RARR_2 \leftarrow \downarrow RARR_1$	▫ Splits nested array to lower rank
$RARR_2 \leftarrow IntVec \downarrow RARR_1$	▫ Drops (not yet overdrops) from array containing refs
$RARR_2 \leftarrow \phi RARR_1$	▫ Reverses array containing references to spaces
$RARR_2 \leftarrow IArr \phi RARR_1$	▫ Rotates according to simple integer array, $IArr$
$RARR_2 \leftarrow \ominus RARR_1$	▫ Reverses array $RARR$ along first axis
$RARR_2 \leftarrow IArr \ominus RARR_1$	▫ Rotates $RARR$ along first axis according to $IArr$
$RARR_2 \leftarrow \diamond RARR_1$	▫ Transposes all axes of $RARR_1$
$RARR_2 \leftarrow IVec \diamond RARR_1$	▫ Transposes $RARR_1$ according to axis positions $IVec$
$VecRARR \leftarrow , ArrRARR$	▫ Ravels arbitrarily nested array to nested vector
$RARR_3 \leftarrow RARR_2 , RARR_1$	▫ Catenates conformable arrays along last axis
$RARR_3 \leftarrow RARR_2 ; RARR_1$	▫ Catenates conformable arrays along first axis
$ISc \leftarrow \equiv RARR$	▫ Depth of array containing references to spaces
$BSc \leftarrow RARR_2 \equiv RARR_1$	▫ Whether arrays of refs all point to the same spaces
$BSc \leftarrow RARR_2 \neq RARR_1$	▫ Whether refs don't all point to the same spaces
$BARR \leftarrow RARR_2 = RARR_1$	▫ Pervasive scalar equality of elements
$BARR \leftarrow RARR_2 \neq RARR_1$	▫ Pervasive scalar inequality of elements
$RARR_2 \leftarrow IVec / RARR_1$	▫ Replicates array containing references to spaces
$RARR_2 \leftarrow IVec \uparrow RARR_1$	▫ Replicates along first axis
$RARR_2 \leftarrow IVec \setminus RARR_1$	▫ Expands array containing references to spaces
$RARR_2 \leftarrow IVec \setminus RARR_1$	▫ Expands along first axis
$RVec_2 \leftarrow RVec_1 \sim RARR$	▫ $RVec_1$ without elements in $, RARR$

The extension of APL primitive function *without* ( $\sim$ ) is implemented in Dyalog APL version 11.

$IArr \leftarrow RVec \iota RARR$	▫ Index of $RARR$ in $RVec$
-----------------------------------	-----------------------------

$BArr \leftarrow Arr \in RArr$

Ⓐ Finds  $Arr$  in  $RArr$

$RArr_2 \leftarrow RArr_1[Index]$

Ⓐ Elements from  $RArr_1$  according to  $Index$  spec.

The  $Index$  specification may be *simple indexing*, *choose indexing* or *reach indexing*. The three corresponding flavours of *indexed assignment* also apply to arrays containing references to objects.

The operators *reduce* ( $/$ ), *reduce first* ( $\neq$ ), *scan* ( $\backslash$ ), *scan first* ( $\neq$ ), *each* ( $\ddot{\cdot}$ ), *compose* ( $\circ$ ), and the *axis* operator ( $[ ]$ ) have all been generalised to deal with arrays containing references to objects. It could be argued (see § 3.3 and New Foundations in *Vector Vol.20 No.1 p132*) that the product operator ( $\cdot$ ) has also been generalised.

11.1.3.2.1 Examine the structure of matrix  $(\# \square SE) \circ . . , (\# \#)$ .

As with the introduction of nested arrays in APL 2, the introduction of arrays containing references to objects is so natural that it is obvious how to generalise the definition of many primitive APL functions and operators, especially the structural functions. There remain some candidate functions such as *type* ( $\in \omega$ ), *membership* ( $\alpha \in \omega$ ), *unique* ( $\cup \omega$ ), *union* ( $\alpha \cup \omega$ ) and *intersection* ( $\alpha \cap \omega$ ), which are not yet implemented but which would seem to have natural generalised definitions too. There are others, such as *take* ( $\alpha \uparrow \omega$ ), *drop* ( $\alpha \downarrow \omega$ ), *without* ( $\alpha \sim \omega$ ) and *find* ( $\alpha \in \omega$ ), which have been partially generalised and are still to be fully generalised.

### §§§ 11.1.3.3 Generalised System Functions

A number of system functions have been generalised to accommodate object references.

$CVec \leftarrow \square CS RSc$

Ⓐ Changes to space  $RSc$  from space named in  $CVec$

$MsgVec \leftarrow \square DQ RVec$

Ⓐ Dequeues events associated with objects in  $RVec$

$Arr \leftarrow IntSc \square NQ RSc CVec \dots$

Ⓐ Enqueues event or method in  $CVec$  of object  $RSc$

$CMat \leftarrow \square FMT RArr$

Ⓐ Character matrix of display forms

$RArr \square FAPPEND Tie$

Ⓐ Appends  $RArr$ , which can include  $\square OR$  of spaces

$RArr \square FREPLACE(Tie Cpt)$

Ⓐ Replaces  $RArr$ , which can include  $\square OR$  of spaces

Further generalisations of system functions and variables doubtless will appear in later versions of Dyalog. For example,  $\square NS$ ,  $\square NSI$  and  $\square PATH$  are candidates. Even  $\square CT$  could be generalised to soften equality of namespaces, for example by ignoring the contents of column 2 of the result of  $\square AT$ . And  $\square WC$  is mortally challenged by  $\square NEW$ , to be described in Module20.

The question naturally arises as to how to make a **deep** copy of a space. Direct assignment only creates a **shallow** copy, *ie* it creates a pointer or ref to the single copy. A deep copy of a namespace may be created using a combination of  $\square OR$  and  $\square WC$ .

$CVec_2 \square WC \square OR CVec_1$

Ⓐ Clone space named in  $CVec_1$  to name in  $CVec_2$

For example, given

```

a←[]ns''◇ a.x←33    ⌘ Creates a namespace containing variable x

'b'[]wc []or 'a'    ⌘ Clones the namespace containing variable x
b.x↪33              x in b is 33
a.x↪33              as is x in a
b.x←4               ⌘ Assigns x in b to 4
a.x↪33              x in a is still 33
b.x↪4               but x in b is now 4
  
```

11.1.3.3.1 Experiment with some of the above generalisations on mixed and nested arrays containing refs.

## § 11.2 Understanding (...).( ...)

### §§ 11.2.1 Expanding Array.Strand

#### §§§ 11.2.1.1 New Rules

**Rule 4: Dots bind tighter than strands. (Strands bind tighter than indexing brackets...) (Indexing brackets bind tighter than rational primitive functions...)...**

**Rule 4** helps one to interpret correctly the order of execution of 3<sup>rd</sup> generation APL statements involving *dot* syntax. The rule may be expressed simply as the order of precedence of dot binding w.r.t. strand binding. APL 1 claimed no special hierarchy of binding strengths amongst functions, nor, separately, amongst operators. There were, however, some anomalous cases like `∘.` and `[]` brackets which should be eliminated (see [K.E.Iverson](#), *Rationalised APL*, IPSA Research Report No.1). Instead anomalous cases have been replicated and others have been introduced, moving APL ever closer to standard multi-rule traditional computer programming languages like FORTRAN, C, VB, VB.NET and C<sup>#</sup>.

This rule allows one to take a next step and rewrite

```
A.first A.last←'Andy' 'Shiers'
```

as

```
A.(first last)←'Andy' 'Shiers'
```

both of which imply that

```
⊢A.first↪'Andy'
⊢A.last↪'Shiers'
```

This is another step in the logical extension of second to third generation APL notation. **Rule 4** implies that parentheses are required above because

```
A.first last←'Kai' 'Jaeger' ⌘ That other hero!
```

implies that

```
⊢A.first↪'Kai'
⊢last↪'Jaeger'
```

Therefore we *do* need parentheses around the name strand `first last` if we mean to imply that both variables be in namespace `A`.

Exactly the same applies to GUI objects and properties. According to **Rule 4**, dots bind tighter than strands and therefore

```
F.Accelerator AcceptFiles↪(F.Accelerator)(AcceptFiles)
```

which is probably *not* what we intend, in which case the parent of `F` probably does not contain an `AcceptFiles` variable. Instead we must write

```
F.(Accelerator AcceptFiles)↳(F.Accelerator)(F.AcceptFiles)
```

Then, for example,

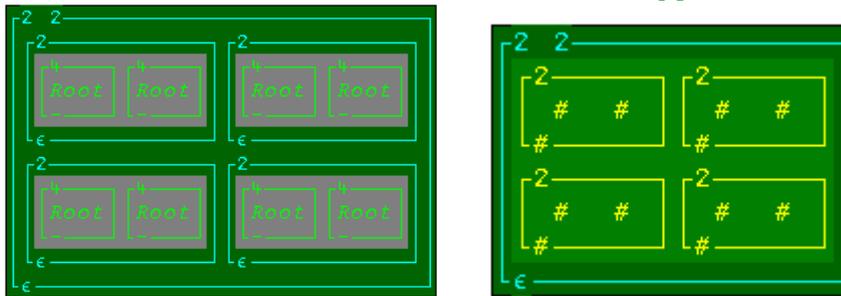
```
F.(Accelerator AcceptFiles Active AlphaBlend AutoConf)
0 0 0 1 256 3
```

The syntax rules (*ie* parsing rules) of the APL language should be clearly distinguished from the semantics (*ie* meaning) associated with specific tokens. The semantic details regarding execution of symbols representing primitive functions and operators involve **algorithms** that define the meaning of the symbols. This should be kept distinct from rules of **grammar**. We therefore define the rules for expanding the various forms of dot syntax in terms akin to the definition of a new dualistic niladic *dot* operator.

$$S_D S^{V..V} Arr \leftarrow \tilde{n}_D . \S^{V..V} \quad \text{A Variable strand pervades deep space nesting}$$

If to the left of the *dot* is an array of refs,  $\tilde{n}_D$ , of arbitrary depth,  $D$ , and to the right is a stranded vector of vectors .. of names,  $\S^{V..V}$ , then the name strand pervades the array of refs so that the structure of the result is the structure of the array of refs ( $S_D$ ) and, within that, the structure of the name strand ( $S^{V..V}$ ), with the value ( $Arr$ ) of each name returned at that point in the nesting.

11.2.1.1.1 Check that the structure of the result of  $((\# \#) \circ . . (\# \#)) . Type$  is consistent with  $\tilde{n}_2 . \S$ .



Here is a final rule which helps one to read and write 3<sup>rd</sup> generation APL statements involving *dot* syntax and which we have inadvertently assumed above.

**Rule 5: The parenthesised expression in  $F.(...)$  is executed in  $F$ -space.**

As with strand notation, this rule seems natural in many circumstances, but it is extra to APL 1 & 2. Note the exception that  $\vdash \# . 1 \neq \# . (1)$ .

The expression  $(...)$  might not return a result. This causes a **VALUE ERROR** in the case of a niladic function such as  $F.(Detach)$ . In this case the unnecessary error can be avoided by using  $F.Detach$ .

If *dot* had been treated as a dualistic niladic operator (with a valid right operand  $(...)$ ), then **Rules 4 and 5** might both have been unnecessary. See *New Foundations in Vector Vol.20 No.1* for discussion of a rational alternative. Henceforth we shall consider new issues involving the interpretation of execution of *dot syntax* structures as **intrinsic properties of the dot symbol** rather than as new APL syntax rules.

Nevertheless, this new rule (**Rule 5**) is a valuable and profoundly useful addition to **Rules 1 & 2**. It is very useful to be able to execute arbitrary expressions in a distant space. For example;

```
□SE.(□IO □ML←1 0)
┆(ρ''□SE.(□CR''↓□NL 3))≡(12 40)(19 102)
```

and also an expression such as

```
#.FORM1.GROUP2.SUB3.SLP<#.FORM1.GROUP2.SUB3.GMP
```

simplifies to

```
#.FORM1.GROUP2.SUB3.(SLP<GMP)
```

GUI object properties and methods are just special cases of variables and functions and as such almost everything that has been said about variables and monadic functions with respect to *dot* syntax applies equally to properties and methods.

11.2.1.1.2 Use **varChar** to examine the properties of a *Calendar* on a *Form* by way of expressions like `#.F.CAL.(CircleToday Border)` and `⊣ 10ρ#.F.CAL.(⊣`PropList)`.

The object to the **left** of the dot *might not be a simple scalar object*, but instead may be an arbitrary *pure* (ie not mixed) array of objects. In this case **the expression to the right of the dot pervades the nested array of refs to its left**. When an array of refs is dotted with a variable name or an object property name then the name pervades the array.

$$(X\ Y).\boxed{IO} \hookrightarrow (X.\boxed{IO})(Y.\boxed{IO})$$

which is the symmetric complement of

$$X.(\boxed{IO}\ \boxed{CT}) \hookrightarrow (X.\boxed{IO})(X.\boxed{CT})$$

which itself follows from **Rule 5**.

$$(2\ 3\rho U\ V\ W\ X\ Y\ Z).\boxed{IO} \quad \text{a matrix of space origins}$$

```
1 1 1
1 1 1
```

$$\rho(F.(C\ D\ E)).Type \hookrightarrow 3 \quad \text{a get 3-vector of object Types}$$

$$(F1.(B1\ B2)\ F2.(B3\ B4)).Caption \quad \text{a get 4 Button Captions}$$

```
France Germany Spain Portugal
```

When an array of refs is dotted with a **strand of variable names** (or a strand of object property names) then the strand pervades the array. We need parentheses in order to strand the 2 variables  $V_1$  and  $V_2$ , and then this strand pervades  $X$  and  $Y$

$$(X\ Y).(V_1\ V_2) \hookrightarrow (X.(V_1\ V_2))(Y.(V_1\ V_2)) \hookrightarrow (X.V_1\ X.V_2)(Y.V_1\ Y.V_2)$$

11.2.1.1.3 Create two *Forms*  $F$  and  $G$  and examine the structure of the results of expressions such as `(2 2ρ#.F).(Type State)` or `((F F)(G G G)).(Size (Posn Caption))`

Aside: Might we have hoped that these would be outer products written  $S_D S^{V..V} Arr \leftarrow \bar{n}_D \circ .\S^{V..V}$ ? Discuss.

### §§§ 11.2.1.2 Parsing Rules

We are going far beyond the usual VB dot syntax, in the APL direction. But before we explore the full generality of APL *dot* syntax, let us recapitulate the evolution of APL syntax as encapsulated in our list of grammatical rules extracted from key language ingredients.

As was regrettably the case with strand notation and **Rule 3** in APL 2, the simple grammar of APL 1 is considerably complicated by the introduction of *dot notation* in APL 3. Beautifully simple formal syntax rules such as **Rule 1** relating to function parsing interpretation, and **Rule 2** relating to operator parsing interpretation, are supplemented by other *ad hoc* heuristic ‘rules of precedence’ defining the order of execution of special new unclassified tokens in a line. The opportunity to completely obviate *strand notation* caused a split in the APL community in the mid 1980's. An opportunity to rationalize APL *dot notation* by way of a strict interpretation of dot as a dualistic niladic operator, which would reduce **Rule 4** to **Rule 2**, has also been lost, as have other opportunities to rationalize along the way. In this regard **J** from Jsoftware is probably the most rational (and, unfortunately, most illegible) dialect of APL today.

In seeking a rule-based view of APL, many irrational and extraneous features of APL have been ignored in an attempt to present a simple unified view. Let us include in a long list some other rules that we might consider to be essential APL 1-3.

**Rule 0: System commands begin with a right parenthesis (as what else could?) and do their own thing (but this is not APL proper...).**

**Rule 0.1: Enter numeric vectors by using standard number formats and spaces between elements, or enter character vectors by surrounding a string in single quotes. This is the real beginning and is very natural.**

**Rule 1: Function sequences execute from right to left. (This is the usual *APL Rule* and follows advanced mathematics.)**

Perhaps because Indo-European languages are read from left to right, most infix mathematical functions (such as minus) are left-associative; that is, a series of functions of the same precedence is evaluated from left to right. However, prefix functions (such as *log* and *tan*) are usually right-associative. APL adopts right-associativity universally for all functions.

**Rule 1.1: There are a number of rules for function header syntax (and for the del ( $\nabla$ ) or other function editors...).**

$\nabla Niladic$	$\nabla R \leftarrow Niladic$
$\nabla Monadic W$	$\nabla R \leftarrow monadic W$
$\nabla A dyadic W$	$\nabla R \leftarrow A dyadic W$

There are header rules for localising variables and shy results and ambivalence and name strands ...

**Rule 1.2: There are rules for *semicolon* (;) indexing and indexed assignment (ameliorated by squish-quad ( $\square$ ), the *index* function in IBM APL2 and Dyalog version 11)...**

**Rule 1.3: Indexing brackets bind tighter than rational primitive functions.**

Hence `7 8[1]*2`  $\hookrightarrow$  `14` and not a *SYNTAX ERROR*

**Rule 1.4: Right arrow ( $\rightarrow$ ) can be used niladically and monadically. (This breaks the *metagrammatical rule* that symbols may be employed both monadically and dyadically (ambivalently), but not either of these and niladically.)**

Otherwise for example `++7` would be ambiguous as the `+` on the left may be interpreted monadically or niladically giving different results.

**Rule 1.5: Labels are immediately followed by a colon (:). A label may be used at the beginning of a line in a program to hold dynamically the line number as a class 1 variable.**

**Rule 1.6: Comments ( $\rho$ ) at the end of a line may be used to hold arbitrary text (somewhat obviated in SharpAPL by the introduction of *lev* ( $\rightarrow$ ) and *dex* ( $\leftarrow$ ) – see the Sharp APL Reference Manual).**

Ken Iverson himself seemed to enjoy using pure APL to add comments in the following manner (even prior to *lev*):

```
3+4,0ρ'Here we add 3 to 4.'↵7
```

**Rule 1.7: Diamonds can be used in a line (and in executable strings) to separate statements.**

<sup>11.2.1.2.1</sup> Try expressions such as that below. This sort of trick can make APL even less compilable.

```
⊖⊖AV[16ρ254 81 88 245] ρ ⊢⊖IO≡1
```

**Rule 2: Operator sequences execute from left to right.**

It is easy to find non-associative functions (eg  $\vdash((A-B)-C) \neq A-(B-C)$  where eg  $A \ B \ C \leftarrow 2 \rho \ \cdot \cdot \ 1 \ 2 \ 3$ ).

Non-associative operators are also possible. They reveal the default order of execution of operator sequences.

$$\begin{aligned} \vdash(A(-.x). \times C) \neq A-.(\times.x)C & \quad \text{ie } (-.x). \times \text{ is not the same as } -.( \times.x) \text{ but,} \\ \vdash(A(-.x). \times C) \equiv A-.x. \times C & \quad \text{ie } (-.x). \times \text{ is the same as } -.x. \times \end{aligned}$$

**Rule 2.1: There is a special rule for outer product ( $\circ$ ) syntax. Rationalisation is compounded by introduction of the *jot* ( $\circ$ ) operator. (Jot could have been usefully defined as, for example, enclose zilde in order to eliminate this rule.)**

**Rule 2.2: Symbols slash (/) and slope (\) can be both functions and operators. (This breaks the second metagrammatical rule that symbols may be either functions or operators but not both.)**

**Rule 2.3: There are a number of rules for operator header syntax (and for the del ( $\nabla$ ) or other editor...).**

$\nabla(f \text{ monisticMonadic}) W$	$\nabla R \leftarrow (f \text{ monisticMonadic}) W$
$\nabla A (f \text{ monisticDyadic}) W$	$\nabla R \leftarrow A (f \text{ monisticDyadic}) W$
$\nabla(f \text{ dualisticMonadic } g) W$	$\nabla R \leftarrow (f \text{ dualisticMonadic } g) W$
$\nabla A (f \text{ dualisticDyadic } g) W$	$\nabla R \leftarrow A (f \text{ dualisticDyadic } g) W$

And rules for shyness... Neither nihilistic operators nor operators that return niladic derived functions figure in 1<sup>st</sup> or 2<sup>nd</sup> generation APLs. (See APL Linguistics in *Vector Vol.2 No.2* for a general classification scheme.)

**Rule 2.4: The axis operator has special rules, similar to bracket indexing (see Rules 1.2 and 1.3).**

**Rule 2.5: There are some individual rules surrounding the syntax for the (multiply-classified) assignment arrow, including choose assignment, modified assignment and function assignment...**

**Rule 3: Strands bind tighter than indexing brackets.**

**Rule 3.1: There are different rules for different control structures, but all of them have to start with a colon followed by a keyword, take an arbitrary number of lines, and end with an :End(optionally immediately followed by the initial keyword).**

**Rule 3.2: Special uses of symbols  $\bar{E} \square \square \uparrow \alpha \omega \Delta \underline{\Delta} \_ \circ \rho \diamond ; : \nabla$  (apart from those mentioned above).**

**Rule 4: Dots bind tighter than strands.**

**Rule 4.1: There are a number of new rules associated with the definition of DFns and DOps in Module12, including proliferation of paired symbols  $\alpha \alpha \omega \omega \nabla \nabla ::$ , analogous to ## in §4.2.3.**

**Rule 5: The expression inside the parentheses in  $F.(...)$  is executed in  $F$ -space.**

**Rule 5.1: There are new 'rules' associated with the expansion of dotted structures.**

As rules proliferate their identification becomes harder. Ultimately it is the parser code that determines the rules and therefore there should be a move to focus on the details of the Dyalog APL parser to identify exactly what the rules are. In any event, we consider it to be very important to explicitly enunciate the major rules of APL because *the reader, not just the machine, has to be able to parse a line* accurately if (s)he is to understand it.

11.2.1.2.2 In which spaces are *k*, *l* and *m* most likely to be found (ignoring `PATH`) in the expression

```
#.A[k].B[l].C[m]
```

Compare the assumed locations of spaces *A B C* and *D* in expressions

```
#.A.B C.D
#.A.B C[2]
#.A.(B C).D
```

### §§§ 11.2.1.3 Generalised Strand Assignment

Assignment into an **array of refs dotted with a variable name** (or object property name) requires either a scalar argument, which experiences scalar extension, or an array of conformable shape and structure to the shape and structure of the array of refs. Assignment is pervasive.

```
(X Y).IO←0      ↪ (X.IO←0)(Y.IO←0)
(X Y).IO←0 1    ↪ (X.IO←0)(Y.IO←1)
(F1 F2).Caption←'F1' 'F2'    ⌘ Set both Form Captions.
```

Assignment into an **array of refs dotted with a stranded structure of variable names** (or object property names) requires either a scalar argument, which experiences scalar extension, or an array of conformable shape and structure to the shape and structure of the array of refs dotted with the strand. The whole strand pervades each element of the ref array. Strand assignment is *totally pervasive*.

```
F.(Caption OnTop)←'The End' 1
(X Y).(first last)←('Søren' 'Kierkegaard')('Dan' 'Baronet')
```

Scalar extension can occur at various levels depending on the structure of the data array.

```
(X Y).(IO ML)←0      ⌘ Scalar extension of scalar 0.
(X Y).(IO ML)←<0 0  ⌘ Scalar extension of scalar <0 0.
(X Y).(IO ML)←2ρ<0 0  ⌘ No scalar extension required.
```

$\tilde{n}_D . \mathcal{S}^{V..V} \leftarrow S_D S^{V..V} Arr$	$\lrcorner$ Strand assignment pervades deep space nesting
--	---

If an array of refs,  $\tilde{n}_D$ , is to the left of a *dot* and a stranded vector of names,  $\mathcal{S}^{V..V}$ , is to the right then the name strand pervades the array of refs. Data assigned to the expanded set of names given by  $\tilde{n}_D . \mathcal{S}^{V..V}$  must have a structure that mirrors the structure of  $(V..V)$  within an outer structure of depth *D*,  $S_D S^{V..V} \dots$ . Each element within this container structure may be any arbitrary enclosed array. So the structure of the data,  $S_D S^{V..V} Arr$ , has the structure of the array of refs  $\tilde{n}_D$ , and within that the structure of the name strand  $\mathcal{S}^{V..V}$ , with the value of each name assigned to the corresponding arbitrary data array *Arr* in  $S_D S^{V..V} Arr$  at that point. If a scalar is encountered in the data at an earlier stage in the correspondence, then that scalar is extended to cover the structure beneath it, as is the case in an ordinary strand assignment without ref arrays:

```
(a b)(c(d(e f)))←1 (3 4)
```

Note there is no resulting difference between the following two assignments

```
F1.(Posn Size)←(55 40)(25 58)
F1.(Posn Size)←(55 40)(25 58)
```

However, a significant general difference between  $\tilde{n}_D . \mathcal{S}^{V..V} \leftarrow \mathcal{S} \dots$  and  $\tilde{n}_D . (\mathcal{S}^{V..V} \leftarrow \mathcal{S} \dots)$  lies in the space location, or locations, of names ( $\mathcal{S}$ ) in expression  $\dots \mathcal{S} \dots$ .





```

ρ ⋅ ⋅ ⋅ ⋅ ⋅ ⋅ (3 1ρA B C).(2 2ρÁ Â).(a b c (c c,d))
                                1                                1
                                1                                1
                                1                                1
                                1                                1
                                1                                1
                                1                                1

```

We can assign a value more than once - *eager*, as opposed to *lazy* evaluation defined in Wikipedia - to variable  $V$  in every leaf space in

```
(A B C ⋅ ⋅ , B C D).(A B C ⋅ ⋅ , B C D).(a b c ⋅ ⋅ , b c d).V←42
```

or

```
(3 1ρA B C).(2 2ρA B).(a b c (c c,d)).X←3 1ρ9 7 4
```

giving

```

(3 1ρA B C).(2 2ρA B).(a b c (c c,d)).X
9 9 9      9      9 9 9      9
9 9 9      9      9 9 9      9
7 7 7      7      7 7 7      7
7 7 7      7      7 7 7      7
4 4 4      4      4 4 4      4
4 4 4      4      4 4 4      4

```

As seemed to be the case when APL 1 first appeared, and again when APL 2 first appeared, the new possibilities for APL 3 seem endless. Two new idioms should be mentioned:

```

RSC←(⊖⊖CS' ').##           ⌘ Returns scalar ref to parent space

```

```

RSC←@.##                   ⌘ Returns scalar ref to current space

```

Remember  $\vdash @ \equiv \square NS()$  where  $\vdash () \equiv \emptyset$ .

11.2.2.1 Rewrite expression  $(\# \#) . (\# \#) . (\# \#)$  in 5 different ways without using dots.

§§§ 11.2.2.3 Generalised Modified Assignment

An arbitrary dotted variable (or stranded variable) structure may be used in a modified assignment.

```

ñD. §V..V f2←SDSV..V Arrr           ⌘ Modified strand assignment of dotted variable structure

```

In this case  $\tilde{n}_D. \S^{V..V} f_2 \leftarrow S_D S^{V..V} Arrr \Rightarrow \tilde{n}_D. \S^{V..V} \leftarrow \tilde{n}_D. \S^{V..V} f_2 S_D S^{V..V} Arrr$

If  $X1$ ,  $Y1$  and  $Z1$  are variable names then an arbitrary name strand may be modified by a dyadic function and a conformable array argument. For example,

```
(X1 Y1 Z1)+←1 2 3           ⌘ X1+←1 ⋄ Y1+←2 ⋄ Z1+←3
```

or

```
(X1(Y1 Z1))+←1 2           ⌘ X1+←1 ⋄ Y1+←2 ⋄ Z1+←2
```

If these names are in space  $N1$ , then **Rule 5** suggests that it should be possible to write

```
N1.(X1(Y1 Z1))+←1 2           ⌘ N1.X1+←1 ⋄ N1.Y1+←2 ⋄ N1.Z1+←2
```

from outside space  $N1$ . Generalising further, we would expect that

```
(N1 N2).(X1(Y1 Z1))+←1
```

should increment variables  $X1$ ,  $Y1$  and  $Z1$  in both  $N1$  and  $N2$  by 1, or that

```
NO.(N1 N2).(X1(Y1 Z1))+←(1(2 3))(4(5 6))
```

would be equivalent to

```
NO.N1.(X1 Y1 Z1)+←1 2 3 ⋄ NO.N2.(X1 Y1 Z1)+←4 5 6
```

This is indeed the case in Dyalog version 11.0. Further than this, selective modified assignment has been partially extended in a natural way to selective modified assignment including objects.

Create 2 *Forms* each with 2 *Buttons*

```
(⊥''FG'⊠WC''c'Form').(⊥''B1' 'B2'⊠WC''c'Button')
```

and in the space of each button create 2 variables, *a* and *b*, with some assigned values

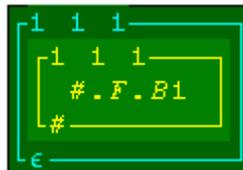
```
(F G).(B1 B2).(a b)←↓↓2 2 2ρ⊖8
```

Then, selection can apply to the namespace references in access and assignment:

```
(1 0/F G).(0 1/B1 B2).(a b)↓,c,c3 4
(1 0/F G).(0 1/B1 B2).(a b)←,c,c9 10
(1 0/F G).(0 1/B1 B2).(a b)↓,c,c9 10
```

The ref array may be any shape and structure,

```
zz←(1 1 1ρF G).(1 1 1ρB1 B2)
```



as long as the shapes and structures correspond with those of the assigned data,

```
(1 1 1ρF G).(0 1/B1 B2).(a b)←1 1 1ρc,c33 34
(1 1 1ρF G).(0 1/B1 B2).(a b)↓1 1 1ρc,c33 34
```

However, the Dyalog implementation is not yet complete for one would expect the following to work

```
X←(1 1 1ρF G).(0 1/B1 B2).(a b)
(1 1 1ρF G).(0 1/B1 B2).(a b)←X
(1 1 1ρF G).(0 1/B1 B2).(a b)+←X
```

**RANK ERROR**

although modified assignment with scalar extension works already in version 11:

```
(1 1 1ρF G).(0 1/B1 B2).(a b)+←1
(1 1 1ρF G).(0 1/B1 B2).(a b+←1)
```

11.2.2.3.1 Discuss with your colleagues these and further generalisations of assignment and the appropriate class of the APL primitive assignment arrow.

## §§ 11.2.3 Expanding Array.Function

### §§§ 11.2.3.1 Array.Niladic

A niladic function may be dotted with an array of object references, in which case the function is executed in every leaf in the array of references.

For example, we can create a non-simple ref array from

```
'ABCDE'⊠WC''c'Form'
≡RARr←(A(B C(D E)))↓-3
```

and execute niladic system function  $\boxed{WA}$  on each *Form*.

```
RARr.⊠WA
```

```
100758844 100758556 100758540 100758368 100758352
```

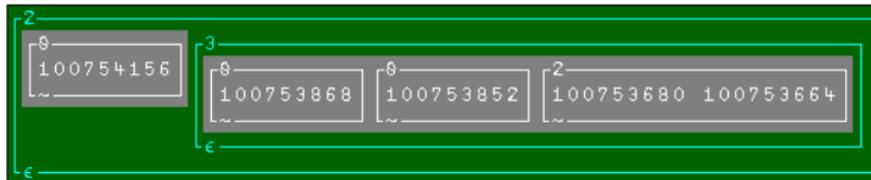
OR

$$(A \ B) . \square WA \Rightarrow (A . \square WA) (B . \square WA)$$

OR

$$\rho RArrr . \square TS \hookrightarrow 2$$

When such a structure ( $RArrr$ ) is *dotted* with a variable name or niladic function, the variable name or function name pervades the namespace structure and the structure of the result (eg of  $RArrr . \square WA$ ) reflects the structure of the namespace array.



A function in a namespace executes within that namespace; and normally only sees other functions and variables in that same namespace.

$$R_D \leftarrow \tilde{n}_D . f_0 \quad \text{where } f_0 \text{ pervades nested structure } \tilde{n}_D$$

A nested array of namespace references,  $\tilde{n}_D$  (depth  $D$  and data type  $RArrr$ ) is *pervaded* by a niladic function in an analogous way to the way in which primitive scalar functions *pervade* their nested arguments.

### §§§ 11.2.3.2 Array.Monadic

A monadic function dotted with a space array *pervades* the space array structure in the same way as a niladic function or an arbitrary array expression does. The arguments, on the other hand, are *distributed* to spaces according to the space structure in the way that data is distributed to each space in the single name assignment  $\tilde{n}_D . \S \leftarrow S_D Arrr$  where no strand expansion takes place.

$$R_D \leftarrow \tilde{n}_D . f_1 W_D \quad \text{where Run } f_1 \text{ in space } \tilde{n}_x \text{ with argument } W_x$$

In the general case of  $\tilde{n}_D . (\dots)$  where  $(\dots)$  is a function expression that evaluates to a monadic function, the items of its argument array(s) are *distributed* to each referenced function. The structure of the argument  $W_D$  mirrors the namespace structure  $\tilde{n}_D$  (and also reflects the *argument rank* of the function  $f_1$ ). The structure of the result  $R_D$  also reflects the namespace structure  $\tilde{n}_D$  (and the structure of the result when  $f_1$  is applied to a typical argument – *ie* the *result rank* of  $f_1$ ).

For example, a monadic scalar function, dotted with a vector of refs, may take a vector argument. The function is then applied to element  $I$  of the argument in element  $I$  of the space vector:

$$R1 \ R2 \leftarrow (N1 \ N2) . f_1 W1 \ W2 \Rightarrow \vdash (R1 \equiv N1 . f_1 W1) \wedge (R2 \equiv N2 . f_1 W2)$$

Note that variables  $W_D$  and  $R_D$  are taken to exist in the covering space, whereas  $f_1$  is assumed to exist in all leaf spaces in  $\tilde{n}_D$ . If the arguments and results are intended to exist in the leaf spaces, then the expression  $\tilde{n}_D . (R \leftarrow f_1 W)$  could be used.

#### 11.2.3.2.1 Examine the depth and structure of the results of

```

\sE . (cbbot cbtop mb popup tip NumEd) . \NL 2
\sE . (cbbot cbtop mb popup tip NumEd) . \NL 3
\sE . (cbbot cbtop mb popup tip NumEd) . \NL < 2 3
\sE . ((cbbot cbtop mb) (popup tip NumEd)) . \NL < < 2 3
    
```

```
⊖SE.(cbbot cbotop mb popup tip NumEd).(⊖NL'')=2 3
and
```

```
(#.F #.G).GetTextSize 'ab'
(#.F #.G).GetTextSize 'abc'
(#.F #.G).GetTextSize<'abc'
```

given that *F* and *G* are GUI *Forms* in the Root space.

### §§§ 11.2.3.3 Array.Dyadic

A dyadic function dotted with a space array *pervades* the space array structure in the same way as a niladic function, a monadic function or an arbitrary array-expression does. The arguments, on the other hand, are *distributed* to spaces according to the space structure in the way that data is distributed to each space in the single name assignment  $\tilde{n}_D. \S \leftarrow S_D Arr$  where no strand expansion takes place. This distribution takes place for both left and right arguments.

$$R_D \leftarrow A_D \tilde{n}_D . f_2 W_D$$

⌘ Run  $f_2$  in space  $\tilde{n}_x$  with arguments  $A_x$  and  $W_x$

In the general case of  $\tilde{n}_D. (...)$  where  $(...)$  is a function expression that evaluates to a dyadic function, the items of its argument array(s) are *distributed* to each referenced function. In the dyadic case, there is a 3-way distribution amongst left argument, reference array (operand) and right argument.

For example, a dyadic scalar function, dotted with a vector of refs, may take vector arguments. The function is then applied to element *I* of both arguments in element *I* of the space vector:

$$R1 \ R2 \leftarrow A1 \ A2 (N1 \ N2) . f_2 \ W1 \ W2 \Rightarrow \vdash (R1 \equiv A1 \ N1 . f_2 \ W1) \wedge (R2 \equiv A2 \ N2 . f_2 \ W2)$$

#### 11.2.3.3.1 Consider 2 *Forms* in a CLEAR WS.

```
'FG' ⊖WC''c'Form'
```

Create a *Button* and a *Label* on each *Form* via a dyadic, space-qualified use of  $\ominus WC$

```
'BL'(F G).⊖WC''c'Button'Label'
```

Set the *Captions* on the child objects with space-qualified property assignment

```
(F G).(B L).Caption←('FB' 'FL')('GB' 'GL')
```

Change the positions of all the children using modified assignment

```
(F G).(B L).Posn×←(.5 .4)(.3 .2)
```

Set the *Dragable* property on all leaf objects

```
(F G).(B L).Dragable←1
```

Create 8 vanilla spaces in every leaf

```
(2ρ<2ρ<⊖AV[17+ι8])(⊖NL 9).(⊖NL 9).(⊖NS'')2ρ<2ρ<8ρ<''
```

Compare this with expression

```
(⊖NL 9).(⊖NL 9).((⊖AV[17+ι8])⊖NS''8ρ<θ)
```

#### 11.2.3.3.2 Experiment with dyadic $\ominus NL$ in place of *plus* (+) in expressions like

```
(1 2)3 4(W(X Y)Z).+1 2(3 4)
```

using expressions like

```
'A'(W(X Y)Z).⊖NL 2 3
```

```
'Aa'(W(X Y)Z).⊖NL''2 3
```

in which the  $\ominus NL$  arguments have appropriate type, shape and rank.

## § 11.3 Arrays of Programs

### §§ 11.3.1 Interpreting $\dots(\dots)\dots(\dots).f_1$

We can now interpret an arbitrary dot-syntax expression. Having identified the dot-syntax sequence sub-expression, starting from the right we consider the right-most token or parenthesised expression in  $\dots(\dots)$ . This parenthesised name or expression preceded by a dot, is to be evaluated in the space preceding the dot. So the token or parenthesised expression in  $\dots(\dots)\dots$  must evaluate to a scalar ref or an array of refs. If there is a dot to the left of this then the token or parenthesised expression in  $\dots(\dots).\dots$  must evaluate to a ref array, and so on until the left-most token or parenthesised expression is encountered. Only now can anything be evaluated. First the left-most term is evaluated in the current covering space to a space ref or array of space refs. The next term is evaluated inside each of these spaces to give a set of sub-spaces, and so on until the right-most term is revisited in the backward pass analysis. The right-most term can finally be evaluated because the space or spaces in which it is to be evaluated are now known. The result of this final term in the dot sequence may evaluate to a numeric, character, ref or mixed array if it is an array expression, or a function if it is a function expression, or any type of named object if it is a single token.

This is rather similar to the VB analysis of the line **ActiveSheet.Range'A1:A2'.Rows.Count** which is read left to right and where **Range'A1:A2'** returns an unnamed object reference. The APL equivalent, however, requires parentheses round the *Range* expression from **Rules 4 & 5**, as in

```
ActiveSheet.(Range'A1:A2').Rows.Count
```

or

```
ActiveSheet.(Range'A1:A2').Cells.(Item 1).Value2
```

or

```
Documents.(Open'C:\MyWord.doc').Activate
```

This procedural proscription in the analysis of a dotted sequence, together with the rules for expansion of terms, is sufficient to obtain the (derived) result of the dot sequence. If the result is a function then the rules for distribution of arguments must be applied in order to obtain the final (array) results of the entire (array) expression.

11.3.1.1 Create a *Calendar* within a *Group* on a *Form*.

```
'F' 'F.G' 'F.G.C'[]WC''Form' 'Group' 'Calendar'
```

Write a niladic function *goo* in *F*-space which returns a ref to the *Group* space, then trace the expression

```
F.goo.C.DateToIDN 3↑[]TS
```

Do the same for *F* and *C* and check the order of execution of *foo.goo.coo* .

11.3.1.2 Create namespaces *a b c* and *d* within *A B C* and *D* within *A B C* and *D*. Use **varChar** to zoom in and view the structure of derived spaces such as

```
⊃⊃(A B C).(A B).(a b c d)
```

or

```
(3 1ρA B C).(2 2ρA B).(a b c (c,c,d))
```

Notice that this has a 3 by 1 outer shape, then a 2 by 2 inner shape, then a 4 vector structure, the last element of which is doubly enclosed.

If, in particular, the final term in a dot-syntax expansion resolves to a monadic function then a function of that name (if it has a name) is assumed to exist in every leaf namespace found to the left of the final dot.

11.3.1.3 Interpret, or otherwise explain, the following lines:

```

(ñ.+.).×          A wrong
#.ñ .+. .×
ñ.(+.×)          A right
#.ñ .+.×
3 4 ñ.(f₂.g₂) 5 6  A where f₂←+ and g₂←× in ñ-space
39
3 4 (ñ.f₂).g₂ 5 6  A should error
3 4 ñ.f₂.g₂ 5 6   A should error

```

### §§ 11.3.2 Arrays of .. Arrays of defined Functions

Motivation: Physics deals with arrays of functions. The position of a particle is a 3-vector. Generally the position is a function of time – a 3-vector of functions  $\mathbf{r}(t) = (x(t), y(t), z(t))$ . Even if the particle position is constant in one frame of reference, it is not necessarily constant in a different frame of reference. Since the laws of physics have to hold in all (inertial) frames of reference, the fundamental equations, eg  $\mathbf{F} = m \mathbf{a}$  (force equals mass times acceleration), generally have to be expressed in terms of arrays of functions.

Dyalog APL does not have notation to represent arrays of functions directly. However, there are ways in which arrays of functions can be represented. For example, functions can be represented as data via `□OR`.

Consider the function expressions

```

R←ϕ∘ϕ  A rotation 90° anticlock
H←ϵ     A reflection

```

then

```

□OR''RH'
ϕ∘ϕ  ϵ

```

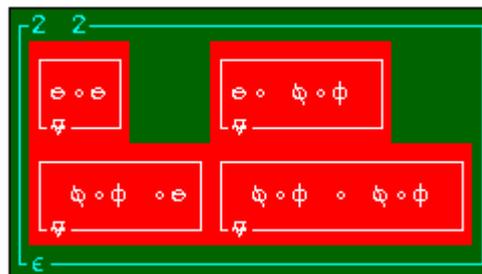
These functions may be combined into a 2 by 2 matrix  $M$  of (`□ORs` of) functions by snippet

```

M←2 2 ρc' '
:For r c :In 1 ρM
  ⍎'M',(⌘r),(⌘c), '←', ('HR'[r]), '∘', 'HR'[c]
  M[r;c]←c□OR'M',(⌘r),(⌘c)
:End

```

which, in `varChar`, looks like:



This technique may be applied to canonical functions to produce arrays that can be indexed and individual functions `FIXed` and executed with appropriate arguments.

Alternatively, an array of functions may be implemented more directly using an array of namespaces each of which contains **a different function of the same name**.

For example, if the position vector as a function of time  $t$  was given by  $\mathbf{r}(t) = (t^{-1}, 2t, 3t^2)$  then this could be implemented as

```
S←⊂NS''3ρc' '
S[1].r←1○÷
S[2].r←2○×
S[3].r←3○×○(2○(*~))
```

then the positions at the first 5 time points are given by the vector function  $S.r$  acting on each time:

```
↑S.r''0,14
0          0  0
1          2  3
0.5        4 12
0.33333333333333 6 27
0.25       8 48
```

11.3.2.1 Make a matrix function of a scalar angle  $\omega$ ,  $Rot\omega$ , representing the rotation matrix in 2 dimensions:

$$\begin{pmatrix} \cos \omega & -\sin \omega \\ \sin \omega & \cos \omega \end{pmatrix}$$

Use it to rotate the 2 element vector (2,3) clockwise through  $\pi/2$  radians.

If Dyalog APL is to be a language suitable for scientific programming then it needs complex numbers. These can be modelled but would be better built into the interpreter as is done in APL2, SharpAPL and J.

Many other mathematical features could be built into the language, such as a monadic determinant function of a matrix, for which Iverson has suggested notation  $\rho.\times\omega$ , or the exponential function of a matrix defined by the power series expansion

$$((\rho\omega)\rho(1+\Rightarrow\rho\omega)\uparrow 1)+\Rightarrow+/(+\times/\prime(\iota\alpha)\rho''c\omega)\div!\iota\alpha$$

by analogy with the usual scalar definition

$$1+\Rightarrow+/(+\times/\prime(\iota\alpha)\rho''c\omega)\div!\iota\alpha$$

where  $\alpha$  is the number of terms in the power series and  $\omega$  is the square matrix to be exponentiated.

Aside: Really big steps in mathematics are from scalar arithmetic to vector algebra and from vector algebra to tensor calculus. Linear vector spaces play a most fundamental role in mathematics, and in APL. If APL could efficiently and neatly handle arrays of functions then this would be a really big step along the road of executable maths. For example, scientists habitually deal on paper with the determinant of matrices of functions, such as the Jacobian determinant, and even with the exponential of matrices of functions. With the new possibility of space-arrays of functions, some of these higher mathematical constructs start to become expressible in raw APL.

### §§ 11.3.3 Arrays of .. Arrays of defined Operators

The same technique as we used to model arrays of functions can be employed to model arrays of operators. (In 3D vector analysis, the gradient and curl operators are **vector operators**. See New Foundations in *Vector Vol.20 No.1* for some more discussion of operators in APL.)

Consider the simple derivative operator that returns the gradient function  $df/dx$  of a given function  $f(x)$ .  $d/dx$  could be approximated in APL as operator  $\Delta$  where

$$f_1 \Delta \leftarrow ((f_1 \ x+\square CT) - f_1 \ x) \div \square CT$$

or better as

$$f_1 \Delta \leftarrow ((f_1 \ x+1E^{-6}) - f_1 \ x-1E^{-6}) \div 2 \times 1E^{-6}$$

With this operator we can find the gradient function  $g(x)=dy/dx$  of , for example, the function  $y=x^2$  with

$$g_1 \leftarrow 2 \circ (*~) \Delta$$

and apply this function, which should be  $g(x)=2x$ , at the points  $x=1..9$  to get

```
g_1 19
2 4 6.000000001 8 10 12 14 15.99999999 17.99999999
```

This operator ( $\Delta$ ) may be placed in 3 different spaces, with just a small adjustment in each, to yield a model of a gradient vector operator that takes the partial derivatives in the x, y and z directions.

```

D←⊖NS**3ρ<' '
⊖CS #.D[1]
▽ R←(f Δ)W;dW
[1] dW←3↑⊖'1E-6' ⋈ (dx 0 0)
[2] R←((f W+dW)-f W-dW)÷2×⊖'1E-6' ⋈ (∂f/∂x)
▽
⊖CS #.D[2]
▽ R←(f Δ)W;dW
[1] dW←3↑-2↑⊖'1E-6' ⋈ (0 dy 0)
[2] R←((f W+dW)-f W-dW)÷2×⊖'1E-6' ⋈ (∂f/∂y)
▽
⊖CS #.D[3]
▽ R←(f Δ)W;dW
[1] dW←3↑⊖'1E-6' ⋈ (0 0 dz)
[2] R←((f W+dW)-f W-dW)÷2×⊖'1E-6' ⋈ (∂f/∂z)
▽
⊖CS #

```

11.3.3.1 Test this vector operator on a scalar function of a vector ( $x^{-1}+2y+3z^2$ ), ie  $f_n$ ,

```

▽ R←f_n W
[1] R←(÷W[1])+(2×W[2])+3×W[3]*2
▽

```

Show that the gradient function ( $f_n D.\Delta$ ) at 3 vector points (x, y, z) gives the expected answer.

```

↑f_n D.\Delta ** (1 2 3)(2 3 4)(4 5 6)
-1 2.000000002 18
-0.2499999994 2.000000002 24
-0.06249999984 1.999999995 36.00000001

```

This simple approximation agrees well with the analytic solution ( $-x^{-2}, 2, 6z$ ) at (x, y, z).

11.3.3.2 Please ask for the next module on **Dynamic Functions** – it's shorter and easier ☺!

Comment: If an operator can be dotted with an operator one might expect that  $+.×$  should be the equivalent of  $\#.+ \#.\#.$ , or  $(\#.+)(\#.\#.)$ , but these latter expressions, unsurprisingly, cause a **SYNTAX ERROR** because  $\#.$  is hard to interpret even if operators can take operator operands. Actually, no primitive operators may be space-qualified, but derived functions such as  $\#.(+.)$ ,  $\#.(o.)$  and even  $\#.(/)$  can be. This is not a limiting factor as primitive operators ( $. o / \backslash$  etc) are identical in every space and so it should never matter from which space they were called. User defined operators, on the other hand, **may** be space-qualified. (Perhaps a bold version ( $\bullet$ ) of the relatively new and subtle addition  $\square AV[94+\square IO]$ , ( $\cdot$ ), should have been chosen instead of ( $\cdot$ ) for dot syntax.)

## Module12: Dynamic Programs

Dynamic Programming (of functions and operators) is an exciting alternative method of program specification to canonical function definition. Dynamic Programs have advantages and also some disadvantages with respect to the usual (canonical) form of programming. Advantages include; clarity for short algorithms, dynamic creation of small localised programs for in-line application, and *more direct control over the power of pure APL notation*. Disadvantages include decreased semantic density, missing features (such as line labels, branching ( $\rightarrow$ ), control structures,  $\square$ PATH,  $\square$ CS and  $\square$ MONITOR), partially implemented features (such as  $\square$ STACK,  $\square$ STATE,  $\square$ REFS and  $\square$ AT) and limited and less intuitive tracing facilities. Some of the gaps narrow with each new version of Dyalog. An early (1985) amusing example of “direct definition” may be found in An APL Dialogue in *Vector Vol.1 No.3*.

Limitations encountered in tracing code, particularly in programs that have been defined inside canonical programs, or the inability to use DFns directly as callbacks or the difficulty of calling non-result returning functions without causing a *VALUE ERROR*, particularly with GUI/OLE programming where one has no control over the shyness of the supplied methods, are unfortunate. The latter can be circumvented using *execute* with a dummy result, as in `sink←⊕'foo rarg∘0'` when *foo* itself returns no result.

### § 12.1 Direct Definition

In Dyalog APL it is possible to directly define an ambivalent function using function specification or *direct function assignment*. Here a name is given to the result of a *function expression* via the assignment arrow ( $\leftarrow$ ) with a function name (eg *f*) on its left and a function (eg  $+$ ) on its right. This syntax implies that assignment be classed as a dualistic niladic operator (where the right operand function may be ambivalent), if assignment were to be formally classified.

```
f←+      ⍝ Session statement (1)
```

A monadic call to *f* will apply the *prefix* function *identity* or conjugate ( $+$ ) and return the Rarg.

```
f 5 ↵ 5
```

A dyadic call to *f* will add the left to the right argument via *infix* function *plus* ( $+$ ).

```
3 f 5 ↵ 8
```

Reference to left and right arguments may be totally elided in statement `f←+` because the definition ( $+$ ) is ambivalent and unambiguous in its argument(s).

Some more complex function expressions can be expressed by means of operators. For example,

```
f←+∘⊞∘↑ ⍝ Used monadically on a vector of vectors
```

However, if one wished to express an algorithm involving left and right arguments in **arbitrary ways**, then the limitation of an assigned function expression to the form

*{left arg} (function expression) right arg*

is too restrictive.

This restriction *could* be alleviated by invoking symbols  $\alpha$  and  $\omega$  to represent implicit left and right arguments to an assigned function. This interpretation of  $\alpha$  and  $\omega$  originates from the models of **direct definition** employed by I.P.Sharp in 2<sup>nd</sup> generation SharpAPL and from earlier APL publications.

Then the ambivalent definition in statement (1) above *could* be accomplished by **two** specifications.

```
f←+ω      ⍝ Overwrites any monadic fn definition
```

```
f←α+ω     ⍝ Overwrites any dyadic fn definition
```

Subsequently, the dyadic form of  $f$  above **could** be replaced by some new dyadic function, say 'under'  
 $f \leftarrow w \div \alpha$     *a Divide rarg by larg (cf larg over rarg)*  
 And such a function **could** be called without having to give it an explicit name.

```
4 (w ÷ α) 3 ↪ 0.75
```

Bear in mind that any name given to an ambivalent function has to be sufficiently general a term as to be suitable for both the monadic and the dyadic context. (The APL primitives actually change their names in the different contexts: eg  $\vdash 5 \equiv +5$  is read as "it is necessarily true that five matches *identity* five" whereas  $\vdash 8 \equiv 3+5$  is read as ".. eight matches three *plus* five".) This highlights the value of pure symbols which should become available with Unicode.

The functions dyadic *catRow* and monadic *justRow* **could** then be defined by direct function definition:

```
catRow ← { (⊢α), ⊢w }    a Catenate Rows  

justRow ← { (-+/' ' ' ° = ° φ'' w) φ'' w }    a Right Justify Rows
```

### §§ 12.1.1 Programming DFns

Programming DFns (dynamic functions) is very like this, except that the essential function definition must be surrounded by braces ( $\{ \}$ ). Thus dynamic functions **can** be defined by:

```
catRow ← { (⊢α), ⊢w }    a Catenate Rows  

justRow ← { (-+/' ' ' ° = ° φ'' w) φ'' w }    a Right Justify Rows
```

12.1.1.1 Define a square root DFn, *sqr*t, such that

```
sqr t 14 ↪ 1 1.414213562 1.732050808 2
```

Consider the *rank* idiom - the **shape of the shape** of an array ( $\rho\rho Arr$ ).  $rr \leftarrow \rho\rho$  gives a **SYNTAX ERROR** because the left-most rho ( $\rho$ ) cannot take a function Rarg – the right-most rho ( $\rho$ ). However the token string  $3 \circ$  is consistent with a right-most rho ( $\rho$ ) as in  $3 \circ \rho$  because jot is an operator.

We need to construct a genuine function so that function assignment can capture the derived rank idiom.

```
rr ← ρ ρ    a Assignment of a function expression
```

But function *rr* is ambivalent and involves a **reshape of shape** algorithm, so  $\vdash 1 \ 1 \equiv 2 \ rr \ , 3$ . By calling  $\rho \circ \rho$  'the rank idiom' it is clear that the dyadic form has been completely overlooked.

The (monadic) rank idiom may be captured in the dynamic function

```
rr ← { ρ ρ w }    a Assignment of a monadic dynamic function
```

In this case there is, as yet, **no dyadic form**. We could define a dyadic form

```
rr ← { α ρ ρ w }    a Assignment of a dyadic dynamic function
```

But this overwrites the previous definition, which means there is now **no monadic form**. We clearly need a mechanism for assigning an ambivalent function.

First note the following features of dynamic function definition.

0. Let  $w$  represent Rarg and  $\alpha$  Larg.
1. Any number of diamondized expressions (segments) may be included within the braces  $\{ \dots \}$ .
2. The first expression (from left to right) that explicitly returns a result will terminate the function at that point and return that result.

3. All variable names created inside expressions in segments on the way to the final result-bearing segment are automatically shadowed prior to assignment.
4. A default left arg  $\alpha$  may be provided simply by assigning  $\alpha$  to a suitable default value. This assignment takes effect only if  $\vdash(, 0) \equiv \square NC ' \alpha '$ , ie if no left argument has been supplied.

Assignment of a default left arg by  $\alpha \leftarrow \dots$  provides a way to define an **ambivalent** function as long as a default  $\alpha$  can be found which will also furnish the appropriate monadic form. The dyadic function has to have a natural monadic case such that the dyadic case with some specific Larg leads to the monadic case. This is possible for a few primitive functions. For example *divide* and *reciprocal* are such that

$3 \{ \alpha \leftarrow 1 \diamond \alpha \div w \} 4 \vdash 0.75$  and  $\{ \alpha \leftarrow 1 \diamond \alpha \div w \} 4 \vdash 0.25$ , and also *power* and *exponential*

$3 \{ \alpha \leftarrow * 1 \diamond \alpha * w \} 4 \vdash 81$  and  $\{ \alpha \leftarrow * 1 \diamond \alpha * w \} 4 \vdash 54.59815003$ , and *log* and *ln*

$3 \{ \alpha \leftarrow * 1 \diamond \alpha \otimes w \} 4 \vdash 1.261859507$  and  $\{ \alpha \leftarrow * 1 \diamond \alpha \otimes w \} 4 \vdash 51.386294361$ , and ..

$\ominus \{ \alpha \leftarrow \phi i r r w \diamond \alpha \otimes w \} 4 \vdash 4$  and  $\{ \alpha \leftarrow \phi i r r w \diamond \alpha \otimes w \} 4 \vdash 4$ , and *minus* and *negate*

$3 \{ \alpha \leftarrow 0 \diamond \alpha - w \} 4 \vdash -1$  and  $\{ \alpha \leftarrow 0 \diamond \alpha - w \} 4 \vdash -4$ , and somewhat

$3 \{ \alpha \leftarrow 0 \diamond \alpha + w \} 4 \vdash 7$  and  $\{ \alpha \leftarrow 0 \diamond \alpha + w \} 4 \vdash 4$ , but monadic *identity* actually applies to non-numeric data too and therefore  $\{ \alpha \leftarrow 0 \diamond \alpha + w \} ARR$  could give a **DOMAIN ERROR**.

The beautiful design of the APL 1 primitive functions is an excellent model for the construction of user-defined functions. Primitive functions apply to arguments of various types and various ranks in meaningfully related ways, like much basic arithmetic notation applies unchanged in the complex domain. Thus in Sharp APL, and now in Dyalog APL version 11, *and* ( $\wedge$ ) and *or* ( $\vee$ ) have been generalised to *lcm* and *gcd* because the Boolean cases follow as a natural consequence of the more general definitions of *lowest common multiple* and *greatest common divisor* (or highest common factor). Furthermore, the monadic and dyadic definitions of primitive functions are usually closely related in meaning, as in the classic case in arithmetic of *negate* and *minus* (-).

The above 4-point scheme for defining dynamic functions is not yet general enough even to model ambivalent primitive functions unless we explicitly use *execute* ( $\pm$ ) in a construct such as

`cross ← { ± > ( b , ~ b ← ( , 0 ) ≡ □ NC ' α ' ) / ' + w ' ' α + w ' }`

Even if we add the following 5<sup>th</sup> point, this limitation is still present.

5. DFns may be nested within DFns in the same way as canonical functions may be nested. eg `unwrap ← { ( w ≠ □ AV [ 3 + □ IO ] ) { α \ α / w } w } ⌘ to replace <LF> with blanks`

Without the ability to jump over diamondized segments, many algorithms become difficult to program. Nevertheless we already have a useful new form of function definition that yields some new idioms.

$\{ w \}$	⌘ Function ( <b>dex</b> ) which returns the right argument
$\{ \alpha \}$	⌘ Function ( <b>lev</b> ) which returns the left argument
$\{ \}$	⌘ Function ( <b>sink</b> ) which does not return any result

Another useful function idiom for converting a niladic form to an ambivalent form is simply `{ niladic }`.



It is not possible to enter a multi-line DFn in the APL session (although **Shift+Enter** as opposed to **Enter** could be defined as *continue* ( $\uparrow$ ) as opposed to *enter* ( $\mathcal{I}$ )). However, you may enter a multi-line DFn in the editor as a stand-alone DFn, or as part of a larger canonical function.

For example, you could define a function to determine the mean value of a numeric vector in the session

```
mean←{(+/ω)÷ρω}           A arithmetic mean
```

or as a 1 line function within a canonical function

```
∇ Variation
[1] Nos←,[]                A input numbers
[2] mean←{sum←+/ω∓num←ρω∓sum÷num} A arithmetic mean
[3] Nos-mean Nos          A difference from average
∇
```

or as a multi-line function within a canonical function

```
∇ Variation
[1] Nos←,[]                A input numbers
[2] mean←{sum←+/ω          A total
[3]      num←ρω           A number of numbers
[4]      sum÷num}         A arithmetic mean
[5] Nos-mean Nos        A difference from average
∇
```

or as a stand-alone **multi-line dynamic function**

```
∇ mean←{sum←+/ω          A total
[1]      num←ρω           A number of numbers
[2]      sum÷num}         A arithmetic mean
∇
```

Note that the final comment will be lost unless it is placed inside the outermost brace.

12.1.2.1 Trace each of the above functions, using some arbitrary set of numbers for input. Check for global variables left in the workspace.

As well as completely empty lines or lines consisting entirely of diamonds or comments, it is also possible to have lines containing nothing but a single left brace {, or a left brace followed by a right brace }{, or a single right brace }. The following function has a valid header line and a valid closing line:

```
∇ compress←{             A remove multiple blanks
[1]      (~' '⊆ω)/ω
[2]      }
∇
```

shown in  $\square VR$  form. Alternatively, the function below is shown in  $\square CR$  form:

```
to←{[]IO←0                A Sequence α .. ω
      from step←1 -1×-\2↑α,α×ω-α    A step default is +/- 1.
      from+step×ι1+0⌈(ω-from)÷step+step=0 A α thru ω inclusive.
}
```

12.1.2.2 Trace the line

```
Eigen ?10 10ρ1000
```

where DFn  $\nabla Eigen \nabla$  is to be found in the distributed workspace  $..\backslash WS \backslash MATH.DWS$ . Compare this function with canonical function  $\nabla EV \nabla$  in §§ 9.3.3

### §§ 12.1.3 Guards and Error Guards

Imagine that dyadic execute ( $\oplus$ ) was defined to take a Boolean Larg ( $BSc$ ) and a character string Rarg ( $CVec$ ) whereby the character string was executed if the Boolean were 1, ie  $\oplus \leftarrow \{ \alpha \leftarrow 1 \diamond \oplus \alpha / \omega \}$ , then this is something like a *guard* ( $BSc : Expr$ ) in dynamic programs. A guard, signified by a single colon (:), is neither a primitive function nor an operator but a new ungrammatical symbol, only available within a dynamic program, with the following meaning:

$BSc : \dots \quad :If BSc \diamond \dots \diamond :End$

An expression (returning  $BSc$ ) to the left of the colon (:) does not need to be surrounded by parentheses and an expression to the right is not surrounded by quotes, as would be the case with the *execute* model.

A DFn may then be written as a series of segments each with an opening guard that determines whether or not the rest of the segment is executed. The first segment to be executed may then return the final result. For example, by analogy with the (atypical) circle function ( $\circ$ ), we could call functions by number:

```

▽ fn←{
[1]      α=1:+ω  A identity
[2]      α=2:-ω  A negate
[3]      α=3:×ω  A signum
[4]      α=4:÷ω  A reciprocal
[5]      α=5:*ω  A e to power
[6]      α=6:⊗ω  A natural log
[7]      }
▽
5 fn 2 fn 3 ⊣ *-3 ⊣ 0.04978706837

```

12.1.3.1 Write a single line DFn which discloses (once) an array if it is scalar and enclosed.

Hint: .. rank zero and depth of magnitude greater than one.

Imagine  $\square TRAP$  had been defined dyadically with the error numbers on the left and the execute cutback expression on its right: this is something like an *error guard* ( $NVec : Expr$ ) in dynamic programs.

A error guard, signified by a double colon (: :), is neither a primitive function nor an operator but a new grammatical **pair of symbols** (going in an unfortunate **J** direction), only available within a dynamic program and with the following meaning:

$NVec :: \dots \quad :Trap NVec \diamond \dots \diamond :End$

The expression (returning  $NVec$ ) to the left of the double colon (: :) does not need to be surrounded by parentheses (making :: impossible to interpret even as a dualistic niladic operator) and the expression to the right is not surrounded by quotes, as would be the case with the  $\square TRAP$  model.

7. Use *guard* (:) to replace *branch* ( $\rightarrow$ ) or *:If*, and *error guard* (: :) to replace  $\square TRAP$  or *:Trap*

The expression to the left of an error guard evaluates to a vector of error numbers. The expression on the right of the error guard is evaluated in the event that one of these errors is generated by subsequent lines (or segments of a line). For example the following function will return  $\square DM$  in the event of any error in the second segment.

```

cover←{0::↑□DM⊙□ω}
cover ?3 3ρ3

```

**DOMAIN ERROR**

```
cover[] cover←{0::↑□DM ◇ □ω}
                ^
```

Note that the trap is unset when executing the expression immediately to the right of an error guard, making trap loops less likely. As with `□TRAP` it is possible to have a hierarchy of traps set, or a series of traps performing different functions. The following example attempts to tie a file, and depending on the error, performs a different alternative, each alternative still being covered by the traps above.

```
open←{
  0::0
  22::ω □FCREATE 0
  24 25::ω □FSTIE 0
  ω □FTIE 0
}
```

In this case, the last line is the first to be evaluated. If a **FILE NAME ERROR** (22) occurs then an attempt is made to create the file. If any error occurs at this point then the function returns 0.

## § 12.2 Extended direct Definition

### §§ 12.2.1 Programming DOps

In canonical form, programming operators is very much like programming functions. Only the header line is slightly different with parentheses round the effective derived function. Likewise, programming DOps is similar to writing DFns but there is no header line to distinguish the two sub-classes. One clue as to the program class when examining a dynamic program is given by the colour of the braces. It is possible to select in [Options][Colours][Syntax][Element] either D-Op (dyadic) or D-Op (monadic) – what we call dualistic and monistic to distinguish from functional form (see APL Linguistics in *Vector Vol.2 No.2 p118*). Pairs of braces can take any of three different colours, one for DFn, one for monistic DOp and one for dualistic DOp.

How does the interpreter know what class a program is? The left operand in a DOp is represented by the double-symbol  $\alpha\alpha$  and the right operand in a dualistic DOp is represented by the double-symbol  $\omega\omega$ . If the double-symbol  $\omega\omega$  exists within the braces (not counting its presence in sub-braces) then the program within the braces must be a dualistic operator as only a dualistic operator has a right operand. If there is an  $\alpha\alpha$  but no  $\omega\omega$  then the program must be a monistic operator, and if there is no  $\alpha\alpha$  then the program is class 3 (a function), and the braces are coloured accordingly.

- Use  $\alpha\alpha$  to represent the left operand and  $\omega\omega$  to represent the right operand of a dynamic operator.

So we could define the primitive monistic *commute* operator ( $\overset{\sim}{\sim}$ ) to be

```
comm←{α←ω ◇ ω αα α}
```

Firstly, if there is no left argument to the derived function then it is taken to be the same as the right argument. (Try primitive *commute*,  $+ \overset{\sim}{\sim} 4 \downarrow 8$ .) Then the function left operand ( $\alpha\alpha$ ) is passed the arguments  $\alpha$  and  $\omega$  in the reverse order -  $\alpha$  becomes the right argument and  $\omega$  the left, exactly as required by the definition of commutation.

```
4 -comm 3 ↓ 4- $\overset{\sim}{\sim}$ 3 ↓ 3-4 ↓ -1
10 *comm 3 ↓ 10* $\overset{\sim}{\sim}$ 3 ↓ 3*10 ↓ 59049
```

Or we could cover the **J** grammatical concept of *hook* with the dualistic operator *hook*:

```
hook←{α←ω◇α αα ωω ω}
4 *hook- 3 ↓ 4*○-3 ↓ 4*-3 ↓ 0.015625
```

DOPs can be multi-line, and they can have guards, just like functions.

```
pow←{
    ⍝ Explicit function power.
    α=0:1
    †{ω}∘αα/(ια),cω
}
```

The derivative operator from elementary calculus takes a single monadic function and returns a monadic derived function, the gradient function. So the operator is monistic and the derived function is monadic. The derivative of a function,  $f(x)$  is  $f'(x)$  where  $f'(x)=df/dx \approx (f(x+dx)-f(x))/dx$ , or to a better approximation  $f'(x)=df/dx \approx (f(x+dx)-f(x-dx))/2dx$ . This is clearly what we have on line [2] below.  $\alpha\alpha$  represents the function operand  $f(x)$  and  $\omega$  represents the function argument ( $x$ ). The operator  $\Delta$  then models the derivative operator  $d/dx$ .

```
∇ Δ←{ ⍝ derivative operator
[1]      dω←⊠'1E-6'
[2]      ((αα ω+dω)-αα ω-dω)÷2×dω
[3]      }
∇
```

The derivative of  $3x^4$  with respect to  $x$  is the function  $12x^3$  for all  $x$ , for example for  $x=3, 4$  and  $5$ .

```
{3×ω*4}Δ 3 4 5 ↪ {4×3×ω*3}3 4 5 ↪ 324 768 1500
```

Symbolically, one could write

```
{a×ω*n} Δ ↪ {a×n×ω*n-1}
```

You can see some other examples of dynamic operators in the `..\WS\DFNS.DWS` workspace. You can download the latest version of this useful workspace, as well as an article `DFNS.PDF` by John Scholes, from [Download Zone] of [www.dyalog.com](http://www.dyalog.com). You can also find examples in the *Language Reference* and in the versions 7.3 or 8.1 new release Help files, downloadable from [Document Download Zone].

One particularly useful operator is `memo` which remembers the result of a function as applied to any particular argument. If called again identically through `memo` then the result is not recalculated, but just returned directly from memory. Functions without side-effects are suitable for memoization. Another example of an operator created by Phil Last, who like John Scholes is a prolific author of fabulous ‘D’ programs, is `else` which, depending on Boolean  $\alpha$ , applies the left operand or the right operand to the derived function argument  $\omega$ .

```
else←{
    ⍝ Condition f else g ...
    α:αα ω ⍝ True: apply left operand.
    ωω ω ⍝ False: apply right operand.
}
```

Join the Dyalog dynamic functions mailbox group [dfns@dyalog.com](mailto:dfns@dyalog.com) for live examples and discussions of issues relating to general dynamic programming, led by the main protagonists.

## §§ 12.2.2 Idioms and Utilities

The entire Finnish APL Idioms list, with over 500 entries and maintained by Veli-Matti Jantunen, has been rewritten in terms of dynamic programs. Every canonical idiom has a dynamic counterpart.

12.2.2.1 Give the following idioms meaningful names. Ask yourself if the word you have chosen reads well in the context of its use. Add some more of your favourite phrases...

```
{ω[⊠AV⊣ω;]}
{ω/ιρω}
{(+/ω)÷ρω}
```

```
{↑(-ιρω)↑**ω}
{□AV[(□AVιω)-48×ω∈□A]}
```

Beware of illegibility and consequent unintelligibility like

```
{m|r←φ3ρφωω,ι(aa←αα)/m←0
α←(aa←{ω}οaa)/m←1
l r←-1+r[|{ω+r×0>ω}(m|r[r←3ρ(ρρω),ρρα)[Ψm×ι3]
↑aa↑(c[l↑ιρρα]α),[-0.1-ι1]c[r↑ιρρω]ω}
```

or inscrutable (until §12.3) one-liners like

```
{' * ' √ . ≠ (ρω) ↑ □ ← ' * + ' / ~ ω { (α + . = ω) , α { + / > [ / + / ** (c ∪ α) ∘ . = ** (α ≠ ω) ∘ / ** α ω } ω } □ : ∇ ω }
```

### §§ 12.2.3 Object.Object. .. Object.Operator Rationale

DFns may be space qualified either by name, or without a name. For example, given function

```
plus←{α+ω}
```

then

```
3 #.plus 4 ↪ 7
```

and

```
3 #.{α+ω} 4 ↪ 7
```

In this respect DFns are just like user-defined functions, object methods or primitive functions. All of the rules stated in Module 11 apply to DFns as they do to other functions.

The same rules also apply to space-qualified DOps. If we had a *natural log* DFn (*ln*) in #

```
ln←{⊗ω}
```

and a derivative operator Δ in #.A.B, then whilst in any other space (see property □SE.CurSpace) we could find the derivative of ln(x) at any points x, say at 6 and 7:

```
#.ln #.A.B.Δ 6 7 ↪ ÷6 7 ↪ 0.1666666667 0.1428571429
```

This all seems quite natural and useful.

```
h←f ñD.ô
```

⊘ Operator(s) ô in space(s) ñ<sub>D</sub> with operand f

```
h←f ñD.ô g
```

⊘ h is space-array of derived functions...

The problem from the point of view of rational APL grammar is that any attempt to argue that the *dot* of dot syntax should be considered to be an operator is now confronted with the situation where *an operator has an operator operand*. This introduces entirely new APL grammar whose implications have been explored in New Foundations in *Vector Vol.20 No.1*. Either you can accept the pragmatic rationale for *Object.Operator* syntax given above or you may seek a deeper justification elsewhere. (Note that in advanced mathematics, the first derivative operator (d/dx) applied to the first derivative operator (d/dx) gives the second derivative operator (d<sup>2</sup>/dx<sup>2</sup>) so there is certainly a precedent in pure mathematics.)

## § 12.3 Recursion

### §§ 12.3.1 Recursive Functions

Most problems that can be solved with iteration can also be solved with recursion. One advantage of recursion is that the program often looks more like the original mathematical formula.

It has always been possible to write recursive functions in APL by referring to the function itself within its own definition. Thus niladic `foo` defined by `⊎fx'foo' 'foo'` is infinitely recursive, as is monadic `foo` defined by `foo←{foo ⊎←w}`. The essential novelty in recursive DFns is the possibility of writing *unnamed* recursive functions. This is implemented simply by using the symbol `del` (`∇`) (*function self*) inside a DFn to refer to the entire DFn itself. So the useless monadic infinitely recursive DFn `foo` above may be replaced by the equally useless `{∇ w}` which may or may not be assigned an arbitrary name.

Many examples of useful recursive DFns are to be found in the supplied DFNS.DWS workspace. Here are a few examples from that workspace.

*Power*, as in  $x^y$  where  $y$  is a positive integer, is just repeated multiplication;  $x \times x \times \dots \times x$ ,  $y$  times. This can clearly be written with a looping solution, or in APL without a loop:

```
x/4ρ3 ↪ 81
```

Alternatively it may be written as a recursive DFn:

```
pow←{w=0:1 ∘ α×α ∇ w-1}
3 pow 4 ↪ 3*4 ↪ 81
```

*Factorial* is a classic case of recursion where factorial of an integer,  $x$ , may be written as  $x(x-1)(x-2)..1$  which translates directly into DFn

```
{w=0:1 ∘ w×∇ w-1} ↪ {!w}
```

with the added value of 1 for factorial zero which enables the function to end the recursion. An alternative program for factorial may be written

```
{α←1 ∘ w=0:α ∘ (α×w)∇ w-1} ↪ {!w}
```

This 'tail-recursive' form turns out to be faster because the segment containing the *function self* returns the result of self immediately as the result of the entire function whereas the first algorithm multiplies the result of *function self* by  $w$  before returning a result as the result of the entire function, making certain internal interpreter optimisations impossible. To be tail-recursive, the answer ultimately returned by the top-level call to the function must be identical to the value returned by the very bottom level call. The ultimate answer, 81, is not the same as the deepest level return value which was 1, so power is not a tail-recursive function but the second (faster) form of factorial is tail-recursive.

There are many examples of beautiful mathematical functions with an elegant recursive definition. The greatest common divisor may be programmed as

```
{w=0:α ∘ w ∇ w|α} ↪ {α∇w}
```

See also algorithms for prime factors or the ancient algorithm for identifying prime numbers first espoused by Eratosthenes of Cyrene who lived around 275-195 BC.

One well known recursive algorithm is that for obtaining the determinant of a matrix. The essence of the method is visible in the following multi-line DFn:

```
det←{⊎IO ⊎ML←0
  1{
    0 0≡ρw:α
    (α×>w)∇ 1 1↓w-w[;0]∘.×w[0;]÷>w
  }w
}
```

A Determinant of matrix w.  
 A initial accumulator.  
 A null matrix: finished.  
 A accumulator ∇ sub-matrix.

Notice how the system variables are automatically localised. Notice also how subtraction and multiplication are at the core of the algorithm, prompting Iverson to propose that a dualistic monadic *dot* operator be introduced such that `{-.×w}` be the *determinant* of a matrix argument. He further calls `{+.×w}` the *permanent* function.

Nested arrays offer much scope for recursive functions. It was not a coincidence that the *each* operator was introduced at the same time as nested arrays. For example, a recursive definition of *enlist*, which may be expressed simply as  $\{\square ML \leftarrow 1 \diamond \in \omega\}$ , is given in function *enlist* in the DFNS workspace.

```

enlist
File Edit View
[0] enlist←(□ML←0           A List α-leaves of nested array.
[1]   α←0                   A default: list 0-leaves.
[2]   α≥ 1+1=ω:,ω         A all shallow leaves: finished.
[3]   1↓↑,/(←←ω),α ∇",ω  A otherwise: concatenate sublists.
[4]   }
Function      Last saved by: Dyadic:13 October 2002 12:14      Pos: 0,1
  
```

12.3.1.1 Study the function *refs* below (also to be found in the DFNS workspace).

```

refs←{
  α←θ ◊ (ρ,α)†α{          A Vector of sub-space refs for ω.
    1∈ω=α:α              A default exclusion list.
    ω.(†∇◊±~/φ(←α,ω),†□NL 9) A already been here: quit.
  }ω                      A recursively traverse sub-spaces.
                          A for given starting ref.
}
  
```

Load distributed workspace WDESIGN.DWS and trace *refs #* with the tracer in [Options][Configure][Trace/Edit][Classic Dyalog mode]. Examples of use include:

```

(refs #).□wx
(refs #).□NL 2
(refs □SE).(ρ◊□CR**3†↑□NL 3)
  
```

Aside: The functions *Legendre*, *Hermite* and *Laguerre* in the distributed MATH.DWS workspace represent the sets of (function) solutions to three commonly applicable differential equations. See, for example, <http://www.efunda.com/math/Laguerre/index.cfm>. These (infinite) sets of orthogonal functions are the eigenfunctions of the corresponding differential operator. A 'recurrence relation' relates each function in a set to neighbouring functions. Thus these three functions may be replaced by recursive definitions.

### §§ 12.3.2 Recursive Operators

There are two distinct kinds of self-reference for recursive DOpS. The symbol  $\nabla$  may be used to refer to the current derived function - the operator bound to its operand(s). When the operands are functions, this is the most frequently used form of self-reference. However, if the operands are arrays, we often need a recursive reference to the operator itself and there we must use the double symbol  $\nabla\nabla$ .

An example of the *first type* of recursion within a DOp is given by the *while* operator. As long as the right operand function  $\omega\omega$  acting on the argument  $\omega$  returns 1, apply the derived function again to the result of the left operand function  $\alpha\alpha$  applied to  $\omega$ , otherwise return  $\omega$ .

```

while←{
  ωω ω:∇ αα ω           A Conditional function power.
  ω                      A While ωω ω: apply αα αα ·· ω.
  ω                      A Otherwise: finished.}
  
```

A fascinating example of the *second type* of recursive operator is given in function *kt* in  $\backslash$ DFNS.DWS.

The most general second type of operator recursion involves a situation whereby the function operands of an operator change at each level of recursion. A simple example is given by

```

comp←{
  α=0:αα ω
  (α-1)αα◊αα ∇∇ ω}
  
```

A potentially very useful example of operator recursion is given by the *function determinant* operator. Imagine you had a 2 by 2 matrix of functions (whose APL representation is as yet undefined),

$$M(x) = \begin{pmatrix} p(x) & q(x) \\ r(x) & s(x) \end{pmatrix}$$

then the function determinant is defined mathematically as the function resulting from

$$\det(M(x)) = p(x)s(x) - q(x)r(x)$$

where multiplication and subtraction are now operators like

```
times←{(αα w)×(ww w)}
```

```
minus←{(αα w)-(ww w)}
```

and the essential recursive APL operator would contain a line something like

```
(αα times ⊃ww)∇∇ 1 1↓ww minus ww[;0]∘.times ww[0;] divide ⊃ww
```

if *ww* was allowed to be a matrix of functions... For a larger size square matrix of functions, the recursive determinant operator would take different function args at each level.

12.3.2.1 The determinant of the function matrix of problem 11.3.2.1 is clearly 1. Consider how you might express this in executable notation.

### §§ 12.3.3 Biological Beauties

Much of the beauty of nature rests on self-similarity - the fact that patterns may be repeated at different size scales. All sorts of natural objects from crystals and sea shells to fern leaves and onions may be modelled by recursive functions. (See Stephen Wolfram's *New Kind of Science* for an extensive and monumental computational analysis of self-similarity.)

One of the first discoveries in this vast new subject was made by Gaston Julia in 1918 and developed and visualised via computer by Benoit Mandelbrot around 1975. They found infinite depth in simple iterative algorithms. We can capture in APL the Mandelbrot set using his algorithm,  $Z=Z^2+C$ . We can picture the set of points on the complex plane whose modulus never exceeds 2 under this iteration. These points are connected and produce a *line* on the plane whose dimension may be considered as not 1 but *fractional*.

The essential algorithm is in the second, third and last lines of the recursive DFn, *square*, defined below. The second line calculates the square of a complex number and adds the position in the complex plane under consideration. The third line determines whether the modulus is greater than 2 (in which case it will diverge and is therefore outside the set). The last line applies the function recursively.

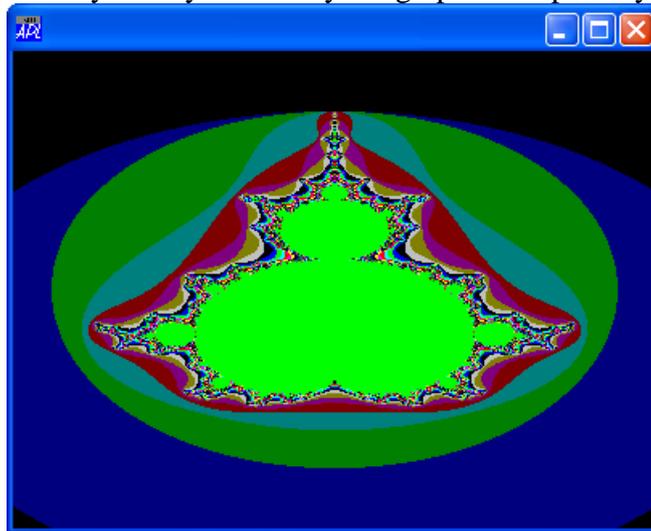
```
Mandelbrot;r;c;v;ADDR;BITS;Cu2;Zu2;x;y;⊞IO; cmap
⊞IO←1
Xmin←-2.5 ⋄ Xmax←1.5           A set X coord limits
Ymin←-1.5 ⋄ Ymax←1.5           A set Y coord limits
r c←300 400                     A number of rows and cols
v←r×c                           A number of pixels
BITS←vρ0                         A initial colour black
x←Xmin+((Xmax-Xmin)÷r-1)×0,⊞r-1  A X range
y←Ymin+((Ymax-Ymin)÷c-1)×0,⊞c-1  A Y range
Cu2←↑,x∘.,y                     A r×c points (2 coords each) on complex plane
Zu2←v 2ρ0                       A zero initial value of Z at each point
ADDR←⊞v                          A address in bits vector
cmap←⊞2 2 2⊞0,⊞7
cmap←(127×cmap)⊞(255×cmap)
cmap[8;]←192
'FRM'⊞WC'Form'('Size'(r c))('Coord' 'Pixel')('Picture' 'BMP' 2)('OnTop' 1)
'BMP'⊞WC'Bitmap'('Bits'(r cρ0))('CMap' cmap)
1 ⊞NQ'FRM' 'Flush'
square←{Zu2 Cu2 ADDR BITS←w
      Zu2←Cu2+((Zu2[;1]*2)-Zu2[;2]*2),[1.5]2××/Zu2           A Z←C+Z*2
```

```

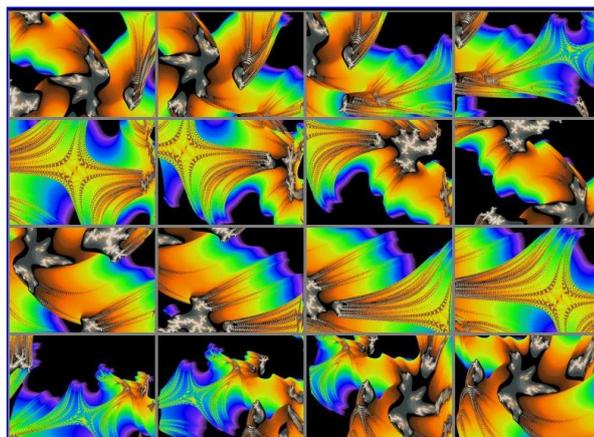
[]←+/-OUTu←-2<0.5*~(Zu2[;1]*2)+(Zu2[;2]*2)      A 2<|Z
^/OUTu:                                             A all outside, QUIT
ADDR←OUTu/ADDR                                     A remove outside addresses
Zu2←OUTu+Zu2                                       A remove outside values
Cu2←OUTu+Cu2                                       A remove outside points
0=ρADDR:                                           A none to update, QUIT
BITS[ADDR]←[]+1+[ /BITS                            A increment effective counter
cbits←(r c)ρ256⊃cmap[1+15|r cρBITS;]              A recalculate colour from depth
_←_⊃''BMP''[]WS''CBits''cbits⊃0'                  A set new colours
∇ Zu2 Cu2 ADDR BITS                                A recurr with subset
}
square Zu2 Cu2 ADDR BITS                           A go

```

The result is a picture, whose beauty is only limited by the graphical capability of the output medium.

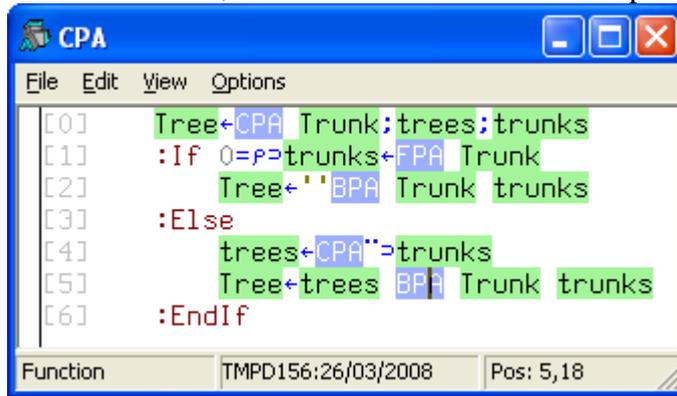


This simple algorithm and the astonishing pictures that it can generate can be applied to the quaternionic (or the octonionic) domain with an identical mathematical algorithm. Prizes have been awarded to some fantastic 3D projections and 2D sections of quaternionic fractals. Many examples can be viewed on the Internet, eg at <http://www.lactamme.polytechnique.fr/Mosaic/images/JU.g2.0.16.D/display.html>.



Such is the power and beauty of recursion by computer. From an examination of a number of recursive models we can extract a fundamental form which is at the heart of many of them. This recursive operator might be named *CPA* from *Critical Path Analysis* wherein two distinct functions emerge from network analysis. The first is *FPA*, *Forward Path Analysis* whereby the network is analysed in a forward (time) direction. Then there is the *BPA*, *Backward Path Analysis*, stage wherein the network is analysed in the opposite temporal order. These two analyses together build a comprehensive description of the network.

Take a straightforward case of a canonical recursive **function** *CPA*, such as that below, that first executes a forward pass *FPA* until a leaf is encountered, and then executes a backward pass *BPA*.



```

[01] Tree←CPA Trunk; trees; trunks
[02] :If 0=ρtrunks←FPA Trunk
[03] Tree←''BPA Trunk trunks
[04] :Else
[05] trees←CPA trunks
[06] Tree←trees BPA Trunk trunks
[07] :EndIf
  
```

This may be rewritten, in a slightly simplified form, as a dynamic recursive operator,

```

CPA←{ ⍺ Tree←(FPA CPA BPA) Trunk
      0=ρtrunks←αα ω:''ωω ω
      trees←(αα ∇∇ ωω)''trunks
      trees ωω ω
    }
  
```

It reapplies the same operands *αα* and *ωω* at every level and so may be replaced with the simpler operator

```

CPA2←{ ⍺ Tree←(FPA CPA BPA) Trunk
        0=ρtrunks←αα ω:''ωω ω
        trees←∇''trunks
        trees ωω ω
      }
  
```

This operator can form the basis of the analysis of many nested structures in APL. *FPA* (or *αα*) is a function that digs down one level into a structure, and *BPA* (or *ωω*) is a function that builds the final result, going backwards one level at a time. We might even propose it be a new *primitive* APL operator ( $\exists$ ).

So, for example, we could take  $\square WN$  as the *FPA* function which digs down into a GUI structure one level and use the simple construction  $\{(\leftarrow\alpha), \leftarrow\omega\}$  for the *BPA*, backward pass synthesis. Thus

```

ρ''''''(⊞WN CPA{⊞α}, ⊞ω)'⊞se'
          9 8 9 2 6 11 9 16 0 7
  
```

12.3.3.1 Consider the following *FPA* candidates and attempt to run an example:

```

{⊞CMD'Dir ',ω} ⍺ even better using ∇NtDirX∇ in NtUtils WS
{>'ω} ⍺ to examine a nested array
⊞REFS ⍺ to examin a function calling tree
{ω.⊞NL 9} ⍺ to examine namespace structure.
  
```

By means of this simple-looking operator many diverse subjects may be investigated and pictured; especially beautifully using OpenGL briefly discussed in Module 9. Valid realms of application include project plans, road systems, real trees, lungs and other natural and idealized fractals of all varieties.

One of the many great things about APL is that it is often imagination and not the programming language itself that is the limiting factor. “What we can conceive we can achieve!”

12.3.3.2 Please ask for the next module on **multi-threading** 😊.

## Module13: APL Threads

Windows divides its workload into tasks or *processes*. Each process is allocated virtual address space and given control of some resources. A *thread* is the smallest kernel-level object of execution. When a process is created, a primary thread is generated along with it. This thread is then scheduled to run on a processor. After the primary thread has started, it can create other threads that share its address space and system resources but have independent contexts. Threads, like processes, can time-slice a processor's throughput leading to the illusion of parallel processing on single-processor machines, which are, usually, most of the time looping idly.

Dyalog APL runs in a *C thread*. When APL starts up it generates an *APL thread*, called the *base thread* or root thread, which can create other APL threads, each with their own execution stack and state indicator. Thus Dyalog supports parallel processing of APL code via multi-threading whereby more than one APL expression can apparently run concurrently. This allows background calculations to run at the same time as interactive tasks, which can greatly improve system responsiveness from a user's point of view.

### § 13.1 Spawning a new Thread

#### §§ 13.1.1 The Spawn Operator, &

An APL thread is initiated by an asynchronous call on a monadic or dyadic function using the new monistic primitive operator *spawn* (&).

$\{TID\} \leftarrow \{\alpha\} f \& w$	$\alpha$ Runs function $f$ in a new thread with ID $TID$
--	--

The (shy) result of the derived function  $f \&$  is the identity of the thread in which  $f$  is being run. When the (ambivalent) function  $f$  terminates, its result (if any) is, by default, returned in the session.

```
t ← 3 × & 4
```

```
1 2
```

```
t ↪ 1
```

The result returned by the derived function  $\times \&$  is **not** the result of the multiplication, but is the unique thread number of the newly created thread – in this case  $t=1$ . We denote this behaviour by

```
3 × & 4 ↪ 2
```

```
1 2
```

The thread number is now 2, the next available positive integer.

An analogous situation arises in the case of executing a diamondized statement whereby the result of an expression is not necessarily that which is displayed in session.

```
r ← ⍎ '33 ⋄ 44 ⋄ 55' ↪ 55
```

```
3 3
```

```
4 4
```

Compare with

```
r ← ⍎ & '33 ⋄ 44 ⋄ 55' ↪ 3
```

```
3 3
```

```
4 4
```

```
5 5
```

which is run in thread number 3, or

```
+ ◦ ⍎ ⍎ 5 ↪ 4
```

```
5.078
```

which is run in thread number 4. Thread ID's are allocated sequentially from 0, the base thread ID, to  $\bar{1}+2*31 \hookrightarrow 2147483647$ , at which point, the sequence 'wraps around' and numbers are allocated from 1 again, avoiding any still in use. The counter may be reset to 0 by `)RESET`.

Functions that take a significant length of time to return their result may be run in the background if their results are not immediately required.

```
#.⊥&'S←OLEServers'
```

Niladic functions can be accommodated by way of execute, or with a monadic DFn:

```
⊥&'Niladic'      ⌘ because Niladic& is syntactically incorrect
{Niladic}&0      ⌘ argument 0 is discarded by monadic dfn
```

13.1.1.1 What would happen if you run function `{+∇&w}` on any argument?

A thread can spawn any number of new sub-threads. This implies a hierarchy of parent and child threads whose ancestral root is ultimately base thread number 0. Children of a terminated parent thread are adopted by the grandparent.

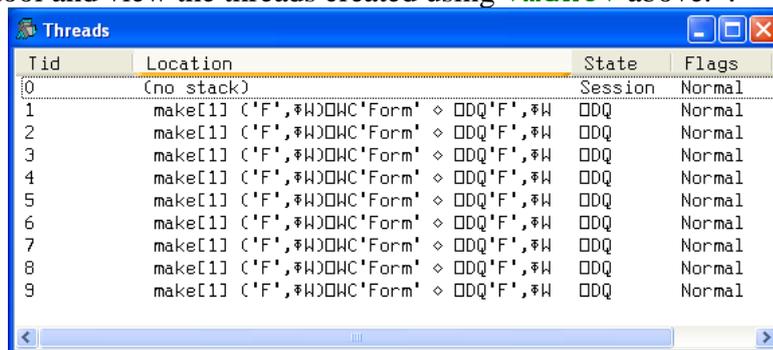
Many parallel threads can be initiated by using each `(∇)` in conjunction with spawn `(&)` because `f&∇` is equivalent to `(f&∇)`. Compare this with `f∇∇&` which is equivalent to `(f∇)∇&` which launches only one new thread.

13.1.1.2 Use the function `∇make∇` to initiate a number of `∇DQ`'ed *Forms* in parallel.

```
∇ make W
[1] ('F',∇W)∇WC'Form' ∇ ∇DQ'F',∇W
∇
```

In order to monitor and debug applications involving many threads a new *threads tool* has been introduced in Dyalog version 10.1 (see Dyalog APL Version 10.1 Release Notes). The new tool may be opened from the session menu by [Threads][Show Threads...] or from the pop-up menu [Threads...].

13.1.1.3 Open the threads tool and view the threads created using `∇make∇` above.



Tid	Location	State	Flags
0	(no stack)	Session	Normal
1	make[1] ('F',∇W)∇WC'Form' ∇ ∇DQ'F',∇W	∇DQ	Normal
2	make[1] ('F',∇W)∇WC'Form' ∇ ∇DQ'F',∇W	∇DQ	Normal
3	make[1] ('F',∇W)∇WC'Form' ∇ ∇DQ'F',∇W	∇DQ	Normal
4	make[1] ('F',∇W)∇WC'Form' ∇ ∇DQ'F',∇W	∇DQ	Normal
5	make[1] ('F',∇W)∇WC'Form' ∇ ∇DQ'F',∇W	∇DQ	Normal
6	make[1] ('F',∇W)∇WC'Form' ∇ ∇DQ'F',∇W	∇DQ	Normal
7	make[1] ('F',∇W)∇WC'Form' ∇ ∇DQ'F',∇W	∇DQ	Normal
8	make[1] ('F',∇W)∇WC'Form' ∇ ∇DQ'F',∇W	∇DQ	Normal
9	make[1] ('F',∇W)∇WC'Form' ∇ ∇DQ'F',∇W	∇DQ	Normal

Delete each Form and observe the corresponding thread disappear.

13.1.1.4 Examine the function `∇thrd∇` which recurs until the SI stack is 4 levels deep.

```
∇ thrd w ⌘ start with thrd 0 after )RESET
[1] :If (∇1+ρ∇SI)<3
[2]   thrd&w,1 ⌘ previous,
[3]   thrd&w,2
[4] :End
[5] ∇DL ?10 ⌘ threads are adopted when parent disappears
∇
```

Run the expression

```
t hrd 0
```

and watch the threads disappear in the thread tool.

§§ 13.1.2 Thread Identity from `⊠TID` and `⊠TNAME`

Each thread has a positive integer ID. The ID of the current thread may be found by means of a system command,

```
)TID ⌘ Reports identity of current thread
```

or by way of a system function inside an APL program,

```
TID←⊠TID ⌘ Current thread number
```

The result `TID` is a simple positive integer scalar, of dataType ZSc. The base thread, which is always present, has `⊠TID=0`. This identity assumes that the base thread is the current thread. Otherwise

```
)RESET
⊠&'⊠TID' ↪ 1
1
```

```
⊠&'⊠TID' ↪ 2
2
```

Each thread can also be given an arbitrary name, of dataType CVec, using the new system variable,

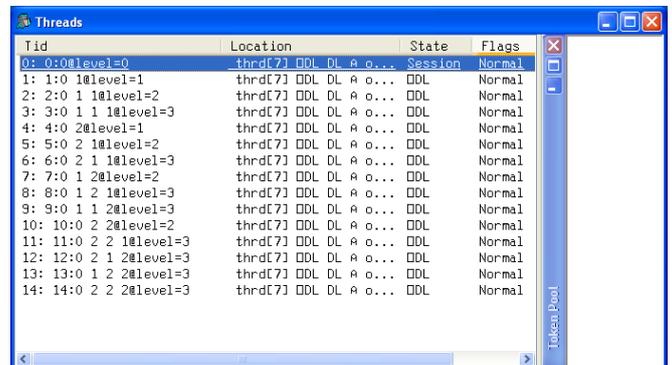
```
⊠TNAME ⌘ The name of the current thread
```

Initially, in a new thread `⊠TNAME=''`.

13.1.2.1 Modify the preceding function `∇ t hrd ∇` as below and create a global variable `DL` with a suitable delay value, say 10 seconds

```
∇ t hrd w ⌘ start with "thrd 0" after )RESET
[1]   ⊠←⊠TNAME←(⊖⊠TID),':',(⊖w),'@level=',⊖1+ρ⊠SI
[2]   :If (⊖1+ρ⊠SI)<3
[3]     thrd&w,1 ⌘ previous,
[4]     thrd&w,2
[5]   :End
[6]   ⊠DL DL
∇
```

Open the Threads tools, reset the SI stack and trace `t hrd 0` to line [5]. Right click on one of the threads in the Threads tool and select [Auto Refresh] and [Switch to]. See new trace window with new current thread in the caption. A star appears against this thread in the Threads tool indicating that it is suspended. Use `)TID` to verify that this is now the current thread. View the `)SI` stack. Reset and save the workspace.



When more than one thread is running, the `)SI` stack is a branching tree originating from the root (base) thread. If a thread sustains an untrapped error then execution of the thread is suspended and all other threads are paused. The session is attached to the suspended thread making it possible to examine local variables and trace through the code. Error messages are prefixed with thread numbers. More information

on debugging threads can be found in the Dyalog version 10.1 [Release Notes](#). In particular, the session supports a number of new facilities for examining thread states.

Threads are flagged in the Threads tool as either normal or paused. A *paused thread* is one that has temporarily been removed from the list of threads that are being scheduled by the thread scheduler. A paused thread is effectively frozen. Runaway threads may be paused with the [Pause All] item in the Thread Tool pop up menu.

It is possible to switch suspension to a different thread, but not to a pendent thread, using the system command `)TID` with a thread ID parameter.

```
)TID TID ⌘ Switch to suspension of thread number TID
```

This suspends a running thread and opens a new trace window on the thread, making it the current thread.

### §§ 13.1.3 Thread Numbers with `⊞TNUMS` and `⊞TCNUMS`

`⊞TNUMS` returns all the thread numbers corresponding to initialised threads.

```
TIDS←⊞TNUMS ⌘ The numbers of all threads
```

The result of the niladic system function is a positive integer vector, data type ZVec.  $\neq 0 \in \text{⊞TNUMS}$

Each thread may have child threads. The resulting hierarchy may be analysed using the system function `⊞TCNUMS` that reports only the child threads of the argument threads.

```
ChildTIDS←⊞TCNUMS ParentTIDS ⌘ The numbers of all child threads of given parents
```

`ParentTIDS` is a simple array of thread numbers (data type ZArr), and `ChildTIDS` is a simple vector of thread numbers (data type ZVec), or zilde if there are none.

13.1.3.1 In the function `t h r d` above, set `DL` to `60` and trace into (**Ctrl+Enter**) `t h r d 0`. Hit **Enter** until you reach line [5]. This will initiate 14 new threads and keep them alive for a minute. Explore the results of `⊞TNUMS` and `⊞TCNUMS`.

It is possible to terminate threads, and optionally (and by default) their dependents, under program control with system function `⊞TKILL`:

```
{Terminated}←{Descendents}⊞TKILL TIDS ⌘ Terminate threads/families in TIDS
```

The Rarg is a simple array of thread IDs (ZArr), Larg is a Boolean determining the fate of descendents (default is 1; terminate entire progeny). The result is a simple vector of actual terminations (ZVec).

$\neq 0 \in \text{⊞TKILL 0}$  ⌘ the base thread is always present.

If an intermediate thread is terminated then that thread's parent adopts the orphaned children.

13.1.3.2 Add new line [2] to the function `∇ t h r d ∇` and replace the delay with an infinite loop.

```
∇ t h r d w
[1]  ⊞←⊞TNAME←(⌘⊞TID),':',(⌘w),'@level=',⌘-1+⊞SI
[2]  ⌘←'0',⌘{⊞ML←1 ⋄ ∈w}{-1+⊞SI)⌘←'⊞tcnums''
[3]  :If (-1+⊞SI)<3
[4]    t h r d&w,1
[5]    t h r d&w,2
[6]  :End
[7]  I←0 ⌘ I is local to thread
[8]  :While 1 ⌘ otherwise thread disappears->adoption
[9]    :If {(w÷1000000)≠[w÷1000000]}I ⌘ update showTree every 1E6
```

```
[10]      updateTree w []TNAME I
[11]      :End
[12]      I←I+1
[13]      :Endv
```

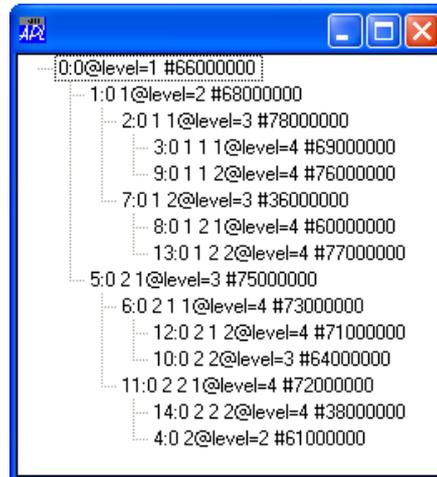
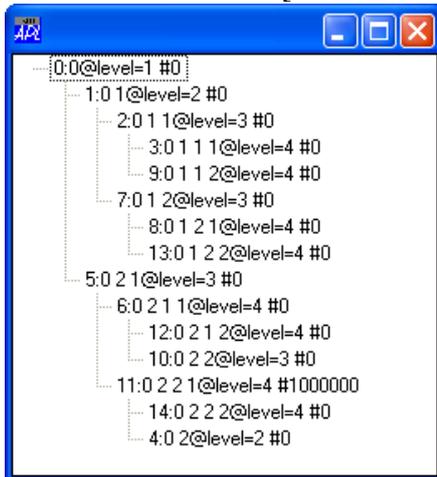
where the function `∇updateTree∇` is, using version 10 new header line syntax,

```
∇ updateTree(W Name I);IDS;Ind
[1]  IDS←↑⊓3 3 3 3↑0 9,(11+ι6),20+ι6  ⌘ list possible IDs
[2]  Ind←IDSι←4↑w                      ⌘ look for current ID
[3]  IDS←F.TV.Items
[4]  IDS[Ind]←←Name,' #',⌘I           ⌘ build item label
[5]  F.TV.Items←IDS
[6]  {F.TV.Expanding w}''ι15         ⌘ NB can't do 1 []NQ... in thread
[7]  ∇
```

Run the niladic function `∇showTree∇`. This sets up a `TreeView` of the coming thread hierarchy.

```
∇ showTree
[1]  'F'[]WC'Form'('Posn' 70 75)('Size' 25 20)
[2]  'F.TV'[]WC'TreeView'('Size' 100 100)
[3]  F.TV.Items←15⌘, '-'
[4]  F.TV.Depth←0 1 2 3 3 2 3 3 1 2 3 3 2 3 3
[5]  F.TV.HasButtons←0
[6]  {F.TV.Expanding w}''ι15∇
```

Open the Threads tool and check the [Auto Refresh] item. Run `thrd 0` after a `)RESET`



Tid	Location	State	Flags	Treq
0: 0:0@level=1	thrd2[12] :If ⏟(w+...	Session	Normal	
1: 1:0 1@level=2	thrd2[12] :If ⏟(w+...	Defi...	Normal	
2: 2:0 1 1@level=3	thrd2[12] :If ⏟(w+...	Defi...	Normal	
3: 3:0 1 1 1@level=4	thrd2[12] :If ⏟(w+...	Defi...	Normal	
4: 4:0 2@level=2	thrd2[13] :End	Defi...	Normal	
5: 5:0 2 1@level=3	thrd2[12] :If ⏟(w+...	Defi...	Normal	
6: 6:0 2 1 1@level=4	thrd2[12] :If ⏟(w+...	Defi...	Normal	
7: 7:0 1 2@level=3	thrd2[12] :If ⏟(w+...	Defi...	Normal	
8: 8:0 1 2 1@level=4	thrd2[12] :If ⏟(w+...	Defi...	Normal	
9: 9:0 1 1 2@level=4	thrd2[12] :If ⏟(w+...	Defi...	Normal	
10: 10:0 2 2@level=3	thrd2[12] :If ⏟(w+...	Defi...	Normal	
11: 11:0 2 2 1@level=4	thrd2[13] :End	Defi...	Normal	
12: 12:0 2 1 2@level=4	thrd2[12] :If ⏟(w+...	Defi...	Normal	
13: 13:0 1 2 2@level=4	thrd2[12] :If ⏟(w+...	Defi...	Normal	
14: 14:0 2 2 2@level=4	thrd2[12] :If ⏟(w+...	Defi...	Normal	

Watch the tree update every million counts. Notice the update order is not predictable. Hit **Ctrl+Break** in the session. Notice the suspended thread is no longer updated. Break again. View the `)SI` stack. Experiment with [Restart All], [Pause All], [Resume All], `TKILL`, [Strong Interrupt] in the System Tray and [Action][Interrupt] in the Session.

## § 13.2 MultiThread Interactions

### §§ 13.2.1 Thread Synchronisation with `□TSYNC`

Often it is necessary to wait for the result of a thread before another program can run. This situation is managed using `□TSYNC`, which takes an argument of a simple array of thread numbers. `□TSYNC` waits until the thread initiating function has finished and all the results, if any, have been produced. It then returns an array of the same outer shape as the argument, each thread result (if there is one) being an enclosed element of the array in the corresponding position.

```
{ArrArr}←□TSYNC ZArr ⌘ Wait for and return results of all thread numbers in ZArr
```

If a thread is subject to an active `□TSYNC`, the thread result appears as the result of `□TSYNC` rather than in the session.

If one thread is waiting for another to finish and that one is waiting for another .. in a cyclic dependency then a trappable `DEADLOCK` error (number 1008) is generated. This error is also generated if you attempt to wait for the base thread to finish by `□TSYNC 0`.

### §§ 13.2.2 Holding Tokens with `:Hold`

When many threads wish to access the same resource then some method for synchronising and controlling access is needed. Access is controlled by *tokens*, arbitrary character vectors identifying entry into critical sections of code.

```
:Hold VecCVec ⋄ ... ⋄ :Else ⋄ ... ⋄ :EndHold ⌘ Attempt to acquire tokens
```

The control structure initiated by `:Hold` blocks entry into the next segment of code until all the tokens in the vector of character vectors (`VecCVec`) have been acquired. If no other `:Hold` has acquired a token then it may be acquired by the current thread. The token is released on exit from the `:Hold` structure.

If a `:Else` clause has been included then execution proceeds into the `:Else` clause if all the tokens in `VecCVec` are not available. Each token may only be held once in the workspace. Trailing blanks are ignored. Holds may be nested in a cumulative fashion, which gives a further danger of `DEADLOCK`.

Thus the function below always results in a `DEADLOCK`. (Note DFns do not support controls structures.)

```

▽ foo
[1]   :Hold 'ι∇|' '7896' '#.#.#' ''
[2]       :Hold 'ι∇|'
[3]       ⌘ execution can never get here
[4]       :End
[5]   :End▽

```

```

foo
DEADLOCK
foo[2] :Hold 'ι∇|'
^

```

A list of all tokens that have been acquired or requested by a `:Hold` control structure can be displayed by the system command `)HOLDS`.

```
)HOLDS ⌘ Reports all tokens acquired or requested by :Holds
```

This command displays all the tokens that have been acquired or requested, one per line. The token is followed by a colon and then the number of the (one and only) thread that has acquired the token followed by all the threads which are currently requesting it.

For example, given the *DEADLOCK* in force above,

```

⊖⊖'foo' ⊣ 1
⊖⊖'foo' ⊣ 2
)HOLDS
:      0      1      2
#.#.#:  0      1      2
7896:   0      1      2
ι∇|:    0      1      2
    
```

### §§ 13.2.3 Pooling Tokens with `⊖TPUT` and `⊖TGET`

Dyalog version 10.1 contains an alternative method for synchronising threads. A pool of tokens is maintained from which tokens may be acquired, when available, and into which tokens may be deposited.

In this context, a *token* has a different meaning from that in section 13.2.2. Tokens are no longer character strings. They are represented by a non-zero integer scalar '*type*', and may optionally have an arbitrary array '*value*'.

The pool may contain up to  $2 * 31 \hookrightarrow 2147483648$  tokens. They are identified by their type and are managed in a FIFO (first in first out) fashion and therefore do not have to be unique.

You can put a token, identified by its type and the sequential order in which it was deposited, into the pool of tokens.

`{TIDs}←{Values}⊖TPUT Types`    *⊖* Puts token *Types* in pool, freeing threads *TIDs*

*Types* (dataType ZVec) is a vector of token types. *Values* (dataType VecArr) is an optional vector of values associated with each corresponding token. The default value is the type itself. The result, if any, of `⊖TPUT` is a vector of thread numbers that have been unblocked by the introduction of the new token(s) into the pool.

Let us put two tokens both of type 29 into the pool:

```

⊖≡⊖TPUT 29 ⊣ 1
⊖≡⊖TPUT 29 ⊣ 1
    
```

The niladic system function `⊖TPOOL` returns the type of every token in the pool.

`Types←⊖TPOOL`    *⊖* Returns *Types* of tokens in the pool

*Types* is a simple integer scalar or vector of token types, or zilde if empty. Thus

```

⊖TPOOL ⊣ 29 29
    
```

The ambivalent system function, `⊖TGET`, retrieves tokens from the pool and returns their values. Negative tokens may be retrieved any number of times. Positive tokens are removed from the pool when they are retrieved.

`{Values}←{TimeOut}⊖TGET Types`    *⊖* Gets token *Types* out of FIFO pool, when available

*Types* is a simple integer scalar or vector that specifies one or more tokens. *TimeOut* is a maximum time in seconds to wait for a response (dataType NSc). *Values* is an arbitrary array value in the case of a single token, or a vector of array values in the case of more than one token being retrieved. `⊖TGET` returns only when the tokens are available (or in the event of a timeout in which case zilde is returned).

13.2.3.1 Try to get token type 29 when the pool is empty.

```
⊞TGET& 29 ⍝ try to get token type 29
```

Note the appearance of a new thread in the Threads tool. Now place a token, type 29, in the pool.

```
⊞TPUT 29
```

Try again and this time give the token a value.

```
+∘⊞TGET& 29
```

```
⊞A ⊞TPUT 29
```

```
ABCDEFGHIJKLMNPOQRSTUVWXYZ
```

The output came from the `⊞TGET` and there are now no tokens in the pool. If we had put a negative type in the pool then it could be retrieved any number of times as a positive type without being removed from the pool. But getting a negative type removes it from the pool.

All tokens can be removed from the pool by `⊞TGET⊞TPOOL`. Note that `⊞TGET 0` can only be stopped by a [Strong Interrupt] (from the System Tray icon).

13.2.3.2 Experiment with `⊞A⊞TPUT^-29`, `+∘⊞TGET&29` and `+∘⊞TGET&^-29`. Also experiment with a request for two such tokens with, for example, `+∘⊞TGET&-29 29` and `⊞A⊞AI⊞TPUT^^-29`.

Outstanding token requests from calls to `⊞TGET` in various threads can be found from the result of system function `⊞TREQ`.

```
Types←⊞TREQ TIDs
```

```
⍝ Current token requests for all thread identities TIDs
```

`⊞TREQ` takes a vector (or scalar) Rarg of thread IDs, and returns a vector of all the requested token *Types* (ZVec or zilde) in all threads in TIDs (positive ZVec).

13.2.3.3) `RESET` to clear the pool. Put 26 tokens into the pool, with values 'A', 'B', .. 'Z'. Note the contents of the Token Pool, which can be docked in the Threads tool. In thread 1, get all the values of 8 5 12 12 15 in a single `⊞TGET` request. Use `⊞TREQ` to examine the outstanding token requests, or look in the Treq column of the Threads tool. Put another token 12 with value 'L' into the pool.

## § 13.3 General Thread Programming

### §§ 13.3.1 Thread Switching

If you execute more than one Dyalog APL thread, a maximum of only one thread is actually running at any instant; the others are paused. Each APL thread has its own State Indicator, or SI stack. When APL switches from one thread to another, it saves the current stack (with all its local variables and function calls), restores the new one, and then continues processing.

Execution may switch from one thread to another only at certain critical points in the code. Generally, execution may not switch mid-line. But the interpreter may switch to a different APL thread

**at the end of every line of code.**

Therefore, one very useful way to ensure that two primitive expressions are executed sequentially without interference from other threads is to place the two expressions on the same line, separated by a **diamond separator** (`⋄`).

Global names set on one line of a function might not have the same value on the next line if other threads access them.

Local names follow the same name scope rules within threads as they do without threads except that each thread stack should be viewed as an independent program stack as far as visibility of local names is

concerned. Local names created at a point in the code after the thread has been spawned are visible to the code in that thread thereafter, but are not visible to threads spawned at an earlier stage. (See, for example, the unlocalised variable *I* in the function `▽thrd[7]▽` above.)

If an intermediate thread is terminated, the grandparent adopts the child threads. This can cause names to change scope and so, generally, all variables should be localised inside functions, especially in multi-thread environments.

There are other times at which execution may switch from one thread to another. These are points at which the interpreter is waiting in an idle state for input:

```
□DQ □DL □FHOLD □ED □SR □ □ :Hold.
```

Other times when the interpreter might execute code in other threads are: while waiting for input from the `□SE` session, while waiting for a `□NA` function call to finish, while waiting for an **AP function** to finish or while waiting for an **OLE function** to terminate. Thread interjection can be controlled with `:Hold`.

### §§ 13.3.2 External Threads with `□NA`

Normally a `□NA` call runs in the same C-thread as APL itself. In order to make the call run entirely in the background it must be run in a separate C-thread. This can be done by placing an `&` after the name of the function in the `□NA` definition and when calling the external function, so defined, through the spawn operator.

Consider the four different ways of running the external function, `Sleep`.

1. Calling `Sleep` in the normal way, for, say, 10 seconds (=10,000 milliseconds), causes the APL session to completely cease responding for the duration:

```
□NA'kernel32|Sleep I4' ◇ Sleep 10000 ⌘ EVERYTHING FREEZES
```

2. Calling `Sleep` in a separate APL thread the normal way through the spawn operator, for, say, 10 seconds, also causes the APL session to stop responding for the duration and does not continue processing code until the 10 seconds has elapsed:

```
□NA'kernel32|Sleep I4' ◇ Sleep&10000 ⌘ EVERYTHING FREEZES
```

3. Calling `Sleep`, having defined it in the `□NA` with a trailing `&`, causes the APL session to be partially active but again all further processing of APL code is frozen until the 10 seconds has elapsed:

```
□NA'kernel32|Sleep& I4' ◇ Sleep 10000 ⌘ SOME MENUS ACTIVE
```

4. Calling `Sleep` through the spawn operator, having defined it in `□NA` with a trailing `&`, causes the APL session to actively respond and further processing of APL code is enabled immediately:

```
□NA'kernel32|Sleep& I4' ◇ Sleep&10000 ⌘ ALL SESSION ACTIVE
```

It is not possible to thread the `□DQ` function directly.

```
□DQ&'MSG'□WC'MsgBox'
```

**DOMAIN ERROR**

This is because `□DQ` can only be run on objects in the same thread. It can, however, be run under cover.

```
□FX'msg w' '□DQ''MSG''□WC''MsgBox''  
msg&1
```

**13.3.2.1** Repeat this with a `Form` and note that APL is free to continue processing in the parent thread after the `Form` has been `□DQ`'ed in a separate thread.

In the special case of a *MsgBox* object, and of other modal dialogue boxes such as a *FileBox*, all processing in other APL threads is suspended until the message box has been dismissed.

13.3.2.2 Define a message box external function  $\nabla mbx1 \nabla$  in the normal manner, and a second function  $\nabla mbx2 \nabla$  that will run in a separate C thread.

```
'mbx1' □NA'I user32|MessageBoxA I <0T <0T I'
'mbx2' □NA'I user32|MessageBoxA& I <0T <0T I'
```

Call the functions in the four different ways; for example,

```
mbx1 0 'call unthreaded' 'NA unthreaded' 1
```

and note the different responses. In particular note that this enables the special case of a message box to run in a separate C-thread and free other APL-threads to continue processing.

One significant advantage of multi-threaded DLL calls that are run in separate C threads is the fact that they, unlike APL threads, can take advantage of multiple processors, if the operating system allows it.

Once a C-thread has been started it is maintained in the APL-thread for subsequent use in that thread and is discarded only when the APL thread finishes. □NA calls that are to be run concurrently in separate APL-threads should be 'thread-safe'. Note that standard Windows API functions *are* thread safe.

□NA calls that interact with Dyalog GUI objects should generally be run in the same C-thread and therefore should not be multi-threaded.

### §§ 13.3.3 Threading callback Functions

All GUI objects are owned by the thread that created them. The Root object (#) and the Session objects (□SE) are *owned* by the Base thread (0). If a thread is terminated then any objects that it owns become owned by the parent thread.

All the events generated by an object are reported to the thread that owns the object and cannot be detected by any other threads. The only exception to this rule relates to events associated with *TCP Socket* objects. Because of the danger of losing TCP events, which should be processed immediately, events from *TCP Sockets* can be seen in **every** thread.

There is a special syntax associated with threading callback functions of GUI objects. Ampersand (&) is simply appended to the name of the callback function when it is assigned to the object's *Event* property, in an analogous fashion to the mechanism used in threading □NA calls. This syntax applies equally to niladic callback functions.

13.3.3.1 Consider a *Form* with a niladic callback  $\nabla draw \nabla$  that draws a *Poly* line on each *MouseMove* event. The *Form* may be created by

```
w←'Form'('Coord' 'Pixel')('BCol' 0 0 0)
w,←('Size' 200 200)('Event' 'MouseMove' 'draw')
'F' □WC w
```

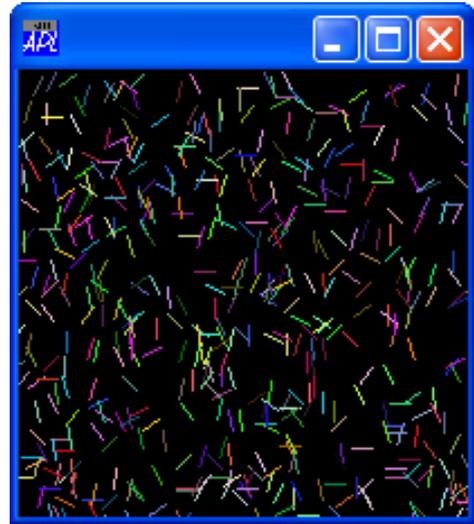
where the callback function is defined as

```
∇ draw;y1;x1;y2;x2      ⍝ line, length 10, random angle
[1]   □DL 0.1          ⍝ waste some time
[2]   y1 x1←?2ρ200     ⍝ random line origin
[3]   x2←(?21)-11     ⍝ -10 ≤ x ≤ 10
[4]   y2←((10*2)-x2*2)*0.5 ⍝ since r2=x2+y2 for a circle
[5]   y2 x2←+y1 x1    ⍝ add random origin
[6]   'F.' □WC'Poly'('Points'(2 2ρy1 x1 y2 x2))('FCol'(?3ρ255))∇
```

Using the mouse, drag your *sprite* over the *Form* and, at the same time, type in the session at the cursor position. Note the response. Now define the *Form* with

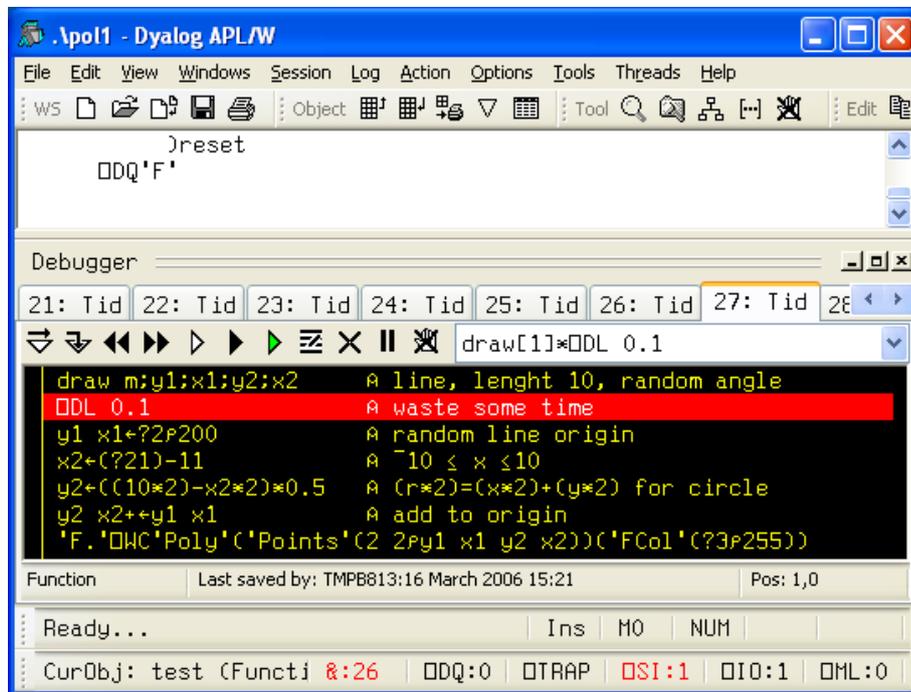
```
'F'⊂WS'Event' 'MouseMove' 'draw&'
```

Apply the same procedure again, perhaps with the Threads tool open. Notice the difference in both the drawing rate and the responsiveness of the Session.



Callback functions may be processed by the default Session dequeue mechanism as invoked above. Alternatively, callback functions may be processed explicitly using `⊂DQ`. `⊂DQ'.'` will process events from any active visible object owned by the current thread or created by callbacks from these objects.

13.3.3.2 Give `vdraw` a Rarg. This can be useful for identifying the owner of an event when tracing. Trace `⊂DQ'F'` via **Ctrl+Enter**. Move the sprite cautiously into the *Form*. Trace through some of the resulting threads in the Debugger.



A thread may use `⊂NQ` to post an event to an object owned by another thread. Any valid Larg except 1 (process immediately) may be used with `⊂NQ&`.

13.3.3.3 Drag the Session (or any other window) over the *Form F* to clear the contents (because we used unnamed *Poly* objects which are not refreshed). Enqueue the following events (from a separate thread) and note the effects.

```
□NQ&'F' 'MouseMove' (?200)(?200) 0 0
0□NQ&'F' 'MouseMove' (?200)(?200) 0 0
2□NQ&'F' 'MouseMove' (?200)(?200) 0 0
1□NQ&'F' 'MouseMove' (?200)(?200) 0 0
```

*DOMAIN ERROR*

```
F.MouseMove&( ?2ρ200),0 0
```

Identify the fundamental difference between the last two expressions.

For further information on multi-threading in Dyalog APL, see the *Language Reference*, Relnotes.hlp for versions 8.2 and 10.1 and [www.dyalog.com](http://www.dyalog.com) [Products][Version 10.1].

13.3.3.4 Please ask for the next module on **TCP/IP Sockets** 😊.

## Module14: TCP/IP Sockets

### § 14.1 The *TCP*Socket Object

#### §§ 14.1.1 IP Addresses and Ports

An *Internet Protocol* (IP) address uniquely identifies a specific network card on a specific computer. Associated with an IP address are one or more ports. Communication between computers requires specification of the IP address and port number of both ends of the connection.

14.1.1.1 To discover the IP address that has been assigned to your computer network card, enter

```
#.TCPGetHostID
```

(Remember [Options][Object Syntax][Expose Root Properties] should be checked.)

A (32-bit) IP address is written as 4 numbers between 0 and  $2^8 - 1 \rightarrow 255$ , separated by dots. For example, 216.239.39.99 is one of the IP addresses of Google.com.

IP addresses beginning 10. or 172. or 192. are internally assigned within an *intranet* and are not reachable from outside your local area network. 127.0.0.1 is the standard IP address used for a loopback network connection. If you try to connect to 127.0.0.1, you are immediately looped back to your own machine.

A (16-bit) port number can be anything between 1 and  $2^{16} - 1 \rightarrow 65535$ . Internet traffic generally uses port number 80. FTP (*File Transfer Protocol*) conventionally uses port 21. SMTP (*Simple Mail Transfer Protocol*) conventionally uses port 25.

#### §§ 14.1.2 *SocketType* and *Style* Properties

TCP/IP stands for *Transmission Control Protocol/Internet Protocol*. It is a fundamental part of the standard protocol for communications on the Internet.

Dyalog APL provides an object called a *TCP*Socket that enables access to TCP/IP communications by way of the API functions in C:\WINDOWS\system32\winsock.dll.

There are two types of TCP/IP connections, both of which are supported by the *TCP*Socket object. A *UDP* (User Datagram Protocol) socket is like a postal service. A single small package is sent to an address. Packages may arrive in any order, and sometimes might not arrive at all!

The second more common type of connection is a *stream* socket, which is like a telephone service. Someone initiates a call. Once a connection is established, both parties have equal status and either party can terminate the call at any stage. Packets are received in the order in which they are transmitted. They are guaranteed to arrive - with automatic error correction.

The equivalent of waiting for a telephone call is creating a *listening* socket. The minimum information required to create a listening socket is the local port number that is to be used for communications, and the name of the new APL object.

```
'S0'⊂WC'TCP'Socket' ('LocalPort' 123)
```

The default socket type is stream, the alternative being UDP,

```
S0.SocketType ⊂ Stream
```

(Remember to make sure that ⊂S0.⊂WX.)

The current state of the socket is 'Listening'

```
S0.CurrentState ⊂ Listening
```

The default *Style* specifies that character data will be transmitted. This is the standard *Style* for Internet traffic.

```
S0.Style ↪ Char
```

Alternative *Styles* are *Raw* and *APL*. 'Raw' communication is via integer vectors between  $-128$  and  $255$  (negative numbers, such as  $-50$ , are added to  $256$  and sent as, in this example,  $256-50=206$ , making the range effectively  $0$  to  $255$ ). *Style* 'APL' communicates via arbitrary APL arrays. The latter is only suitable for communication between two APL workspaces.

### §§ 14.1.3 Workspace to Workspace Communications

In the first instance, we are going to create a listening socket in a workspace on our computer by running the following function, where function `▽show▽` is just `⊞FX'show Msg' 'Msg'` (not `show←{w}`).

```
▽ listen;w
[1] w←c'Type' 'TCPSocket'
[2] w,←c'LocalPort' 123
[3] w,←c'Style' 'APL'
[4] w,←c'Event' 'All' 'show'
[5] 'S0'⊞WC w
▽
```

The *Create Event* displays immediately, via `▽show▽`, the message

```
S0 Create 1
```

The *TCPSocket* object has been assigned a socket number

```
S0.SocketNumber ↪ 696
```

which is the Windows handle of the socket. The current state is 'Listening'. The intended final state is 'Server'

```
S0.TargetState ↪ Server
```

as opposed to possible states 'Client' or 'Closed' for a stream socket. Setting the target state to 'Closed' is the approved method of closing a socket because then APL waits until all data has been sent before issuing a *TCPClose Event*.

```
S0.TargetState←'Closed'
S0 TCPClose
```

Next we start a new instance of Dyalog APL and use the function below to create a *TCPSocket* that will connect to a listening socket on port 123. Again, function `▽show▽` is just `⊞FX'show Msg' 'Msg'`.

```
▽ connect;w
[1] w←c'Type' 'TCPSocket'
[2] w,←c'RemoteAddr' '127.0.0.1'
[3] w,←c'RemotePort' 123
[4] w,←c'Style' 'APL'
[5] w,←c'Event' 'All' 'show'
[6] 'S1'⊞WC w
▽
```

The minimum information required to create a connecting socket, apart from its arbitrary APL name, is the remote address and the remote port number, which must be set to the number of the listening socket's local port. When *RemoteAddr* and *RemotePort* properties match the IP address and port number of any listening socket, the client and server sockets connect. In this simple case we use our own IP address (found from `#.TCPGetHostID`) or, equivalently, the standard loopback address `127.0.0.1`.

14.1.3.1 Run `∇listen∇` in one workspace and `∇connect∇` in another. After the initial *Create* event in the listening workspace, you should get a *Create* event in the connecting workspace, and simultaneously, a *TCPAccept* event in the listening workspace. Check that the *TCPAccept* event is immediately followed by a *TCPConnect* event in the connecting workspaces followed by a *TCPReady* event in both workspaces.

The sockets, both of *Style* APL, are now connected through port number 123 (on the listening side). As with a telephone, once the connection has been established, the communication is peer-to-peer and neither party monopolises the connection communication resources.

Either party can send arbitrary APL arrays, including arrays that contain `⊞OR`'s of namespaces, to the other in a single atomic operation using the *TCPSend* method. For example, the 'connecting' workspace can send matrix `(3 3⍴19)` to the 'listening' workspace by

```
2 ⊞NQ'S1' 'TCPSend' (3 3⍴19)
```

which is reported in the 'listening' workspace through the *TCPRecv Event* as

```
S0 TCPRecv 1 2 3 127.0.0.1 1568 19313036
           4 5 6
           7 8 9
```

Or the 'listening' workspace can send vector `⊞A` to the 'connecting' workspace by

```
2 ⊞NQ'S0' 'TCPSend' ⊞A
```

which is reported in the 'connecting' workspace through the *TCPRecv* event as

```
S1 TCPRecv ABCDEFGHIJKLMNOPQRSTUVWXYZ 127.0.0.1 123 1927140
```

Erasing the socket object on either side (eg *S1*) closes the connection and removes the partner object (*S0*).

14.1.3.2 Change the last line in `∇listen∇` and `∇connect∇` to

```
'S...'⊞WC w ⋄ ⊞DQ'S...'
```

By running `{listen}&1` in one workspace and `{connect}&1` in the other, show that TCP messages are received independently of which thread owns the socket. This is intentional as otherwise, if messages are not processed immediately, there is a chance that they might get lost.

If you try to run `∇listen∇` and `∇connect∇` in the same workspace but in different threads then you will get error number 10061 reported by the *TCPError* event. The interpretation of such error codes is to be found in many places on the Internet, eg [http://www.sockets.com/err\\_lst1.htm](http://www.sockets.com/err_lst1.htm). In this case the connection has been refused because there is a conflict over multiple port allocations for a single process.

14.1.3.3 Write a callback on the *TCPRecv* event in both workspaces that will execute a linear APL expression being sent by the other party.

Hint: Include a line like `⊖(3>Msg)~⊞AV[6 9 10]`

The distributed workspace REXEC.DWS, described in the *Interface Guide*, furnishes a more sophisticated example of remote execution by a web server.

## § 14.2 A simple character Socket

### §§ 14.2.1 Connecting to a server Socket with *TCPConnect*

14.2.1.1 Create a simple character listening socket in the first workspace, which we shall call the *server*.

```
'S0'⊖WC'TCP Socket'('LocalPort' 123)
```

In a separate workspace, create another character socket, the *client*, which will connect to the server socket

```
w←'TCP Socket'('RemoteAddr' '127.0.0.1')('RemotePort' 123)
```

```
'S1'⊖WC w ⋄ S1.onTCPConnect←'show'
```

```
#.S1 TCPConnect
```

Note the immediate response of the *TCPConnect* event as soon as the two sockets connect. When a connection succeeds, the *CurrentState* of the client *TCP Socket* object changes from 'Open' to 'Connected' and it generates a *TCPConnect* event.

As you can verify, it is important that the *TCPConnect* event is active as soon as the socket is created. If the *Event* is set on the line after the *⊖WC* then the response from the server can easily be lost through sloth of the client or zeal of the server.

### §§ 14.2.2 Sending to the server Socket using *TCPSend*

We may use the connect event on the client to immediately send a request to the server socket. In order to see the request we set a callback on the server *TCPRecv* event.

```
S0.onTCPRecv←'show'
```

On the *TCPConnect* event in the client we put the callback function

```
∇ conn Msg
```

```
[1] 2 ⊖NQ(=Msg)'TCP Send' 'file1'
```

```
∇
```

This will send the character string 'file1' to the server as soon as the connection is achieved. So the server begins with

```
'S0'⊖WC'TCP Socket'('LocalPort' 123)
```

```
S0.onTCPRecv←'show'
```

And some time later the client runs

```
w←'TCP Socket'('RemoteAddr' '127.0.0.1')('RemotePort' 123)
```

```
'S1'⊖WC w ⋄ S1.onTCPConnect←'conn'
```

As soon as *S1* is created, the server gets the message below, where 1323 is the remote port number.

```
#.S0 TCPRecv file1 127.0.0.1 1323
```

### §§ 14.2.3 Receiving from the server Socket with *TCPRecv*

Having received this message from a client, the server can respond to it by, for example, sending back to the client the contents of file1.

First we create some native files containing 'valuable' information for the client:

```
'file1'⊖NCREATE 1 ⋄ 'this is file 1...'⊖NAPPEND 1 ⋄ ⊖NUNTIE 1
```

```
'file2'⊖NCREATE 1 ⋄ 'THIS IS FILE 2...'⊖NAPPEND 1 ⋄ ⊖NUNTIE 1
```

Then we replace the *TCPRecv* callback on the server with function

```

▽ recv Msg;t;d
[1]   :If 'file1'≡3>Msg      a did they ask for file1?
[2]       t←'file1'□NTIE 0  a tie file1
[3]   :Else⊃If 'file2'≡3>Msg a did they ask for file2? etc...
[4]       t←'file2'□NTIE 0  a tie file2
[5]   :End
[6]   d←□NREAD t 82(□NSIZE ^1)0 ◊ □NUNTIE t a read the file
[7]   2 □NQ(>Msg)'TCPSEND'd  a send the contents
▽
    
```

One further consideration before this will work is: ‘What will the client do with the requested information sent from the server?’ We need at least a basic callback on the client *TCPRecv* event.

So now the server runs:

```

'S0'□WC'TCPSocket'('LocalPort' 123)
S0.onTCPRecv←'recv'
    
```

And the client runs:

```

w←'TCPocket'('RemoteAddr' '127.0.0.1')('RemotePort' 123)
'S1'□WC w ◊ S1.onTCPConnect←'conn' ◊ S1.onTCPRecv←'show'
    
```

On running this last line, the immediate response from the receive event is:

```

#.S1 TCPRecv this is file 1... 127.0.0.1 123
    
```

Note that the receive event must be set in the `□WC`, or at least on the same line as the `□WC`, in order to ensure that the incoming message is not lost.

<sup>14.2.3.1</sup> Using the appropriate remote address, the remote *TCPGetHostID*, repeat the above example using two separate computers rather than two workspaces on the same computer.

## § 14.3 Some Complications

### §§ 14.3.1 HTTP and HTML

The above trivial example of a character *TCPocket* connection affords a very simple model of how the Internet is generally used at the socket level. Remote servers wait in a listening state for a call. A browser connects to some server and immediately sends a request for a file – the home page say. The server receives the request and sends the file contents, which the browser then presents on the screen. The server then closes the connection.

Details concerning exactly how the incoming information is formatted on the client screen are solved by way of HTML plain ASCII text. Server files do not contain arbitrary text strings, but rather text specified in *HyperText Markup Language*. This embodies a method of incorporating format information into plain text. Browsers such as Microsoft Internet Explorer, Netscape Navigator and Mozilla Firefox all recognise this format and present text thus expressed in a uniform, precise and detailed fashion for all surfers to see.

HTML is constantly developing and the language is described in many places, for example in <http://www.w3.org/TR/xhtml1/>. Each page presented on a browser may be viewed in its (approximate) original HTML form by selecting menu item [View][Source] in the browser.

Not only does an Internet browser expect HTML text, all the information sent between browser (client) and server must conform to a special transmission protocol, as the title TCP/IP - Transmission Control Protocol / Internet Protocol – would suggest. Requests and responses between browsers and servers are wrapped in

*Hypertext Transfer Protocol* (HTTP). The authoritative guide to HTTP is in the Request For Comments RFC2616, found, for example, at <http://www.w3.org/Protocols/rfc2616/rfc2616.txt>.

### §§ 14.3.2 Buffering received Data

Data sent using *TCPSend* may be automatically split into packets and sent sequentially. Similarly, data received from a server may come in packets. It is therefore necessary to implement a mechanism in *TCPRecv* that will reconstitute an entire message. The simplest way to do this is simply to catenate incoming data in the *TCPRecv* callback until a *TCPClose* event is encountered (because the server closed the connection) at which point the incoming data is assumed to be complete and is saved appropriately.

More specifically, HTTP defines the sequence *carriage return, linefeed* (CRLF) to be the end-of-line marker. So command lines end with CRLF (ie `⎕AV[4 3]`). Further, at the end of the message HTTP requires an empty line followed by CRLF. Therefore the receive callback looks for CRLF CRLF at which point it knows that the end of the entire message has been reached.

### §§ 14.3.3 Servicing multiple Connections

14.3.3.1 In your server workspace, write a function `▽listen▽` which sends a reply as soon as a message is received from a client.

```
▽ listen;w
[1] w←,c'Type' 'TCPsocket'
[2] w,←c'LocalPort' 123
[3] w,←c'Event' 'TCPRecv' 'recvByServer'
[4] 'S0'⎕WC w
▽
```

where the callback on the receive event is

```
▽ recvByServer Msg
[1] ⎕←3>Msg
[2] 2 ⎕NQ(=Msg)'TCPsend' 'Server says, "Goodbye Client!'"
[3] (≠>Msg).TargetState←'Closed'
▽
```

In the client workspace, write a function `▽connect▽` which creates a connection socket and immediately sends a message.

```
▽ connect;w
[1] w←,c'Type' 'TCPsocket'
[2] w,←c'RemotePort' 123
[3] w,←c'RemoteAddr' '127.0.0.1'
[4] w,←c'Event' 'TCPConnect' 'conn'
[5] w,←c'Event' 'TCPRecv' 'recvByClient'
[6] 'S1'⎕WC w
▽
```

where the connect callback is

```
▽ conn Msg
[1] 2 ⎕NQ(=Msg)'TCPsend' 'Client says, "Hello Server!'"
▽
```

and the receive callback is

```
▽ recvByClient Msg
[1] ⎕←3>Msg ⌘ display message from server
▽
```

Run `▽listen▽` and `▽connect▽`. Note that the messages did get through, but that at the end of the brief conversation all sockets have disappeared. Therefore no further communication is possible until the server creates a new listening socket.

On sending its reply to a request from a client, a server generally sets its *TargetState* to `'Closed'`. This closes the server socket and terminates the connection. At this point the connection is closed and further requests from the client must open a new connection. This means that the server should have a new listening socket available immediately in order to be ready for the next request.

In order to deal with this situation, *TCP Sockets* have a special mechanism to create a new listening socket as soon as a connection has been made to the current listening socket.

The *TCPConnect* event triggers in the client when the server attempts to connect to the client. The *TCPAccept* event triggers in the server when the client actually connects to the server. This event includes the socket handle in the callback message, and at this point the server has the opportunity to *clone* the listening socket. In the *TCPAccept* callback it is possible to create a new socket which is a clone of the current socket by setting the *SocketNumber* property of the new socket to the socket number reported in the callback. The only other properties that may be set here are the *Event* and *Data* properties. This new socket then becomes a new listening socket, thus maintaining the server's ability to serve.

14.3.3.2 Add a new line in the `▽listen▽` function:

```
w,←c'Event' 'TCPAccept' 'acc'
```

where the `▽acc▽` callback is

```
▽ acc Msg;w
[1] w←,c'Type' 'TCPSocket'
[2] w,←c'SocketNumber' (3>Msg)
[3] w,←c'Event' (⊡>Msg).Event
[4] ⌈w,←c'Data' (⊡>Msg).Data
[5] ('S',⌈1+⊡1↓>Msg)⌈WC w
▽
```

Run `▽listen▽` and `▽connect▽` again. Each time a client connects, the callback on *TCPAccept* clones the original listening socket with a sequence of new *TCP Socket* objects using the name *S1, S2,...* The server workspace always has a listening socket available and therefore `▽connect▽` may be run again and again...

14.3.3.3 Why doesn't `connect◇connect` work? Fix it by changing the line that creates the client socket:

```
[6] ('S',⌈COUNT←COUNT+1)⌈WC w
```

where *COUNT* is a suitably initialised global variable.

Details regarding the *TCP Socket* object and its properties and methods are to be found in the *Object Reference* and in the Dyalog 7.3 or 8.1 release notes. Detailed examples of its use can be found in the *Interface Guide*, [www.dyalog.com](http://www.dyalog.com) [Products][Dyalog for Windows][TCP/IP Support], APL97.RTF available from [www.dyalog.com](http://www.dyalog.com) and the example workspaces in `..\samples\tcpip\`.

14.3.3.4 Please ask for the next module on **Web Servers**.

## Module15: APL Web Servers

### § 15.1 Making a simple Server

It is possible to use *TCP Sockets* on your computer to host a web server that will be accessible by everyone on your network. A good example, trivialised below, is to be found in the supplied workspace `..\Samples\tcpip\www.dws`, namespace `#.SERVER`, and is described in detail in the *Interface Guide*. A more robust example of a web server is to be found in the distributed workspace `..\aplservice\server.dws` and associated files.

#### §§ 15.1.1 Creating a listening Socket

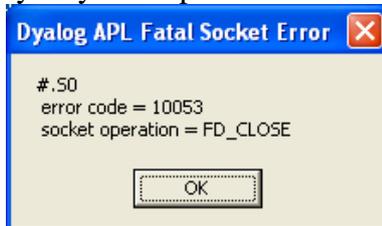
15.1.1.1 In a clear WS, create a socket on *LocalPort 80*, the default for simple web servers:

```
'S0' □WC'TCP Socket' ('LocalPort' 80)
```

Check that it's *CurrentState* is *'Listening'*. Now open your Internet Explorer and type `http://127.0.0.1` in the address bar and hit **Enter**. Check that the *CurrentState* is now *'Connected'*. (You might have to use a different experimental port if 80 has been barred.)

Had we created a socket on a different port, say 2020, typing `http://127.0.0.1:2020` into the address bar we would have established a connection to this port, but **:80** is the assumed default. (Different ports relate to different services. The IP address-plus-port combination uniquely identifies a web site.)

The socket connection is not much use as it stands. IE and our server remain connected indefinitely but the connection does nothing useful. If you go to the IE address bar and hit **Enter** again then an error box pops up in APL. It tells us that we did not get the communication protocol right. This is not surprising as our server socket did not acknowledge in any way the explorer who made the link.



The *CurrentState* of *S0* is now *'Closed'* and the socket soon disappears.

Placing a callback function with result 0 on the *TCPError* event, or setting the *Event* action code to `-1`, will stop these pop-up error boxes from appearing.

15.1.1.2 In a clear WS, create a socket on *LocalPort 80*, and set all *Events* to `▽show▽`.

```
□FX'show Msg' 'Msg'
```

```
'S0' □WC'TCP Socket' ('LocalPort' 80)('Event' 'All' 'show')
```

The *Create Event* callback (*show*) should respond immediately with the message

```
S0 Create 1
```

Now repeat the above experiment by entering `http://127.0.0.1` into your Internet Explorer.

Inside IE, the browser creates a socket using as the remote address the IP address specified in its address bar. It then gets a connect event and immediately sends an HTTP request to our server.

The subsequent events from the APL host server's point of view occur as follows. First the *TCPAccept* event triggers and reports the socket number (handle) of the original socket we created (eg 272).

```
S0 TCPAccept 272
```

The original `S0` has actually been replaced by a new socket `S0` that has taken on the responsibility of connecting to the client browser that requested the connection. The original listening socket, now nameless, continues to exist, at least until any callback on the `TCPAccept` event has finished processing.

The `TCPAccept` event is followed by a `TCPReady` event and then by a `TCPRecv` event. This last event, the `TCPRecv` event, triggers on receipt of some data from IE. The event message contains the received data concatenated with the IP address and port number of the network source of the data (the address and port of the client – our IE browser socket). The actual data received from IE, which prints out in the session as a consequence of the `▽show▽` callback, looks something like:

```
GET / HTTP/1.1
Accept: */*
Accept-Language: en-gb
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Host: 127.0.0.1
Connection: Keep-Alive
```

This message is composed in HTTP protocol and says something like "Hello Host, please send me your home page. I understand all sorts of stuff if you talk `HTTP/1.1` language. I'll keep the line open until I hear from you. Over." (Over is CRLF CRLF.)

Details of the precise meaning of the message may be extracted from the official World-Wide Web specification of the HTTP/1.1 protocol - in <http://www.w3.org/Protocols/rfc2616/rfc2616.txt>.

The first and most important line is made up of a *request type*, a *request document URI* (Uniform Resource Identifier) and a *protocol*, all separated by a single space:

```
GET / HTTP/1.1
```

The network location of the URI is also transmitted in a Host header field.

The above is a GET request, for the domain root index file (/ followed by a space or just a space) in protocol HTTP/1.1 . RFC2616 says,

"The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process."

All we need to understand so far is that GET may be followed (after a space) by the name of a file on the server whose hypertext content is to be returned (inside a valid HTTP message) to the client, and if no file is specified, but only a /, then this implies that the file to be returned is the site home page.

Setting the `TargetState` of `S0` to `'Closed'` will close the connection gracefully and erase the socket.

### §§ 15.1.2 Cloning a listening Socket on `TCPAccept`

One important job an APL server has to do on the `TCPAccept` event is to clone the listening socket so that someone else can establish a connection (even while the first is still connected). This is done by creating a socket in the `TCPAccept` callback with the same `SocketNumber` as the original listening socket.

**15.1.2.1** In a clear WS, create a socket on `LocalPort 80` (or `LocalPortName http`), with a callback on the `TCPAccept` event and a global variable `COUNT`, set to zero:

```
'S0' □ WC 'TCP Socket' ('LocalPort' 80) ('Event' 'TCPAccept' 'acc')
```

in which `▽acc▽` is

```

▽ acc Msg;w      ⍝ PERFORM WHEN ACCEPT CONNECTION TO CLIENT
[1]  COUNT←←1      ⍝ increment COUNT on each accept.
[2]  w←,←'Type' 'TCPSocket'      ⍝ create a socket,
[3]  w,←←'SocketNumber'(3>Msg)    ⍝ with the handle of the original,
[4]  w,←←'Event'((>Msg)⊖WG'Event')⍝ with all the events as before.
[5]  ('S',⊖COUNT)⊖WC w          ⍝ create with next name.

```

After connecting from IE, check  $(S0 S1).CurrentState$   $\leftarrow$  'Connected' 'Listening'. Set the *TargetState* of  $S0$  to 'Closed' and repeat the request from IE. After connecting from IE, check that  $(S1 S2).CurrentState$   $\leftarrow$  'Connected' 'Listening'. Note the encouraging message in the IE status bar saying "Opening page http://127.0.0.1/..."

### §§ 15.1.3 Sending an HTML File on *TCPRecv*

The simple general pattern of events in a working server is this:

1. Wait for a request to connect from a web browser
2. Connect and clone listener in preparation for another request
3. Send the information requested by the browser, eg the home page
4. Close the connected socket
5. REPEAT AS REQUIRED

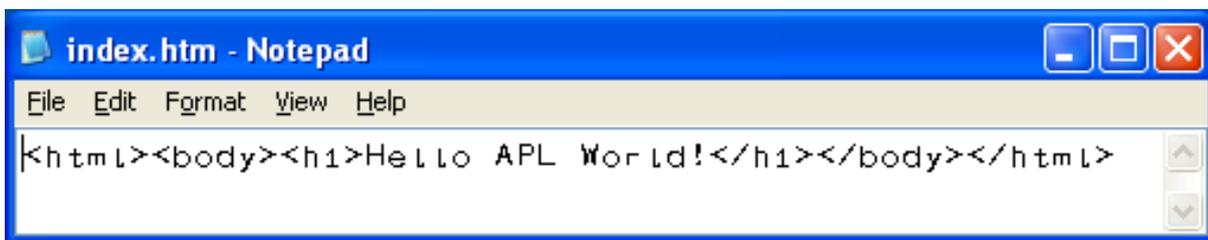
A *TCPRecv* callback implements items 3 and 4. In the simplest scenario, all the receive function has to do is read the web site home page, send it back to the server and close the connection.

```

▽ rec Msg;t;d      ⍝ PERFORM WHEN RECEIVE MESSAGE FROM CLIENT
[1]  t←'C:\homepage\index.htm'⊖NTIE 0
[2]  d←⊖NREAD t 82,2⊖NSIZE t      ⍝ read
[3]  ⊖NUNTIE t
[4]  2 ⊖NQ(>Msg)'TCPSEND'd        ⍝ send
[5]  (>Msg)⊖WS'TargetState' 'Closed' ⍝ close

```

A trivial home page in the file C:\homepage\index.htm, written in HTML so that IE can present it nicely, could be just one simple line:



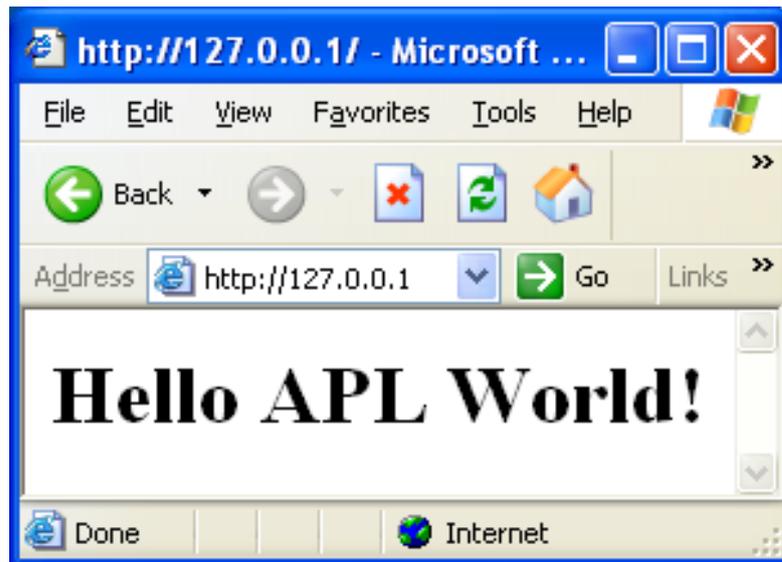
15.1.3.1 Erase all sockets and restart IE. Create a new socket with *TCPAccept* and *TCPRecv* callbacks:

```

COUNT←0
'S0'⊖WC'TCPSocket'('LocalPort' 80)⍝('LocalAddrName' 'LocalHost')
'S0'⊖WS('Event'('TCPAccept' 'acc')('TCPRecv' 'rec'))

```

With line [1] of  $\nabla rec \nabla$  pointing to a suitable root file, such as index.htm above, use IE to navigate to your web site. IE should then display the intended page.



Note that in order to run/trace this repeatedly it might be necessary to close IE between connections as IE remembers (caches), and doesn't deem it necessary to request again, a static page before displaying it.

## § 15.2 Making a realistic Server

Of course a real web page would be expected to have more on it than just 'Hello...'. Not only that, a real web designer would aspire to having more than one page on his web site. This and much more is possible using hypertext HTML. See a beginner's guide at <http://www.put.com/HTMLPrimer.html>.

The HTML in the main web site file (traditionally called index.htm - the one that is loaded if nothing special is requested), can tell a client browser about any other HTML files and pictures to be found on the web site host that are required in order to construct the complete home page. The browser opens another connection to the web site and requests that file next. If that file points to others which are required to complete the page being displayed then the browser will open another connection and ask for that file, and so on .. until the entire page has been built on the screen.

Also embedded in HTML may be links to other files on the web site that define (through a number of sub-files) other complete pages. The links can be defined such that one click from a user can send the browser off to request all the required files and then construct the page even while more pieces are being delivered from the web site host server. The links can equally refer to pages (or, by default, the main page) of any other web site on the network by including the IP address of the site in question inside the HTML.

Thus, for example, the server in the distributed workspace `..\server\tcpip\www.dws` refers to a homepage to be built from `..\samples\tcpip\homepage\index.htm`. This homepage file, shown below, includes links to other files such as that in:

```
<frame src="home.htm" name="overview">
```

The HTML in home.htm includes calls to other files that call yet more files - all required in order to build the home page.

```

index.htm - Notepad
File Edit Format View Help
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<meta name="GENERATOR" content="Microsoft FrontPage 2.0">
<title>Dyalog APL</title>
<meta name="KEYWORDS"
content="APL, Dyalog APL, Programming Languages, Array Processing, Array, Arrays">
<meta name="
description" content="The Dyalog APL Home Page">
</head>
<frameset rows="40,*">
  <frame src="navigate.htm" name="navigate" marginwidth="1"
marginheight="1" scrolling="no" noresize>
  <frame src="home.htm" name="overview">
</frameset>
<body>
<p><!--webbot bot="PurpleText"
preview="The frameset on this page can be edited with the FrontPage Frames Wizard; use the Open or Open With option from the
FrontPage Explorer's edit menu. This page must be saved to a web before you can edit it with the Frames Wizard. Browsers that don't
support frames will display the contents of this page, without these instructions. Use the Frames Wizard to specify an alternate
page for browsers without frames."
s-viewable=" " --> </p>
<p>This web page uses frames, but your browser doesn't
support them.</p>
</body>
</frameset>
</html>

```

Calls to pages other than the site home page are to be found in navigate.htm. When the area in the specified rectangle is clicked, the browser will take the user to the URL (*Uniform Resource Locator*) specified by the HREF (*Hypertext Reference*) value.

```
<AREA SHAPE="RECT" COORDS="338, 6, 392, 27" HREF="support.htm">
```

The requested resource in HREF may be anywhere on the network:

```
<AREA SHAPE="RECT" COORDS="460, 6, 537, 27" HREF="http://195.212.12.1:8081/frserve.htm">
```

Developing web sites in raw HTML can be an onerous task. However, numerous WYSIWYG applications have been developed to write HTML for you. This makes it much easier to create complex web sites with many pages, each containing text, graphics and other controls. Microsoft FrontPage, Macromedia Dreamweaver or even Microsoft Word 9.0, are some of the many applications that you may use to design and alter WYSIWIG pages of your web site.

### §§ 15.2.1 Threading multiple Connections

A web server is usually intended to be able to host a service to a number of clients simultaneously. So far we have enabled this by ensuring that a new listening socket is always present. However, a new client cannot connect to a new socket or expect a reply until processing for previous clients has finished. Creating each new listening socket in a separate APL thread can dissipate this potential queue and thus make the service more responsive to multiple simultaneous connections.

15.2.1.1 Change `▽ acc Msg;w` as below so that new listeners are cloned in a separate thread. Use the unique thread ID to name the socket and process socket events through `□DQ`.

```

▽ acc Msg;w      ⍝ PERFORM WHEN ACCEPT CONNECTION TO CLIENT
[1]   :If 9=□NC>Msg ⍝ if socket exists,
[2]   clone&Msg ⍝ clone in a separate thread.
[3]   :End
▽

```

where

```

▽ clone Msg;w    ⍝ CLONE THE OLD LISTENING SOCKET
[1]   w←,'Type' 'TCPSocket'      ⍝ create socket,
[2]   w,←'SocketNumber'(3>Msg)   ⍝ with the handle of the original,
[3]   w,←'Event'((>Msg)□WG'Event') ⍝ and with all previous events.
[4]   □DQ('S',⊘□TID)□WC w       ⍝ create with unique name.
[5]   ⍝ a line after a line with a □DQ is helpful for tracing
▽

```

Check that your web server still works with this enhancement in place.

A number of other essential features and useful suggestions are to be found in the supplied `SERVER` workspace. For example, you are advised to include a callback, even if empty, on the `TCPClose` event to ensure that a socket does not close prematurely. Also the `TCPErrror` event callback returns `0` to stop popup error boxes appearing during the transaction process. The callback simply erases any socket causing an error. The `TCPErrror` event message contains the error details and therefore more error handling could be done in this callback. To avoid potential problems in development, all sockets should be expunged before initiating the service.

In preparation for the information expected in the `TCPRecv` callback, a variable called `BUFFER` is initialised to `⎕AV[4 3]` (CRLF) in the connected `TCPSocket` namespace. CRLF is the recognised HTTP command string separator. `BUFFER` is going to be filled with the HTTP commands from the client browser that is connected to this socket.

## §§ 15.2.2 Communicating through HTTP

If large amounts of data are being sent between stream sockets in a single transaction then the data is automatically broken into manageable packets and sent one at a time. The HTTP identifier for the end of a complete set of packets is CRLF CRLF – two empty lines. The possibility of receiving incomplete message packets is thus incorporated into a robust server by adding lines to the `TCPRecv` callback which add packets to a buffer and look for the end marker CRLF CRLF.

```
(⊜>Msg).BUFFER,←3>Msg
:If ⎕AV[4 3 4 3]≠~4↑(⊜>Msg).BUFFER ⌘ if we have not got everything,
:Return ⌘ stop and wait for more.
:EndIf
```

In the simple examples above, a small amount of data was sent so no `:Return` was necessary, but this is not guaranteed in a stream socket, although the order of receipt of packets is guaranteed to be the same as the order of transmission.

When a browser first establishes a connection with our server, the HTTP data shown in section §§15.1.1 is received, terminated with `⎕AV[4 3 4 3]`. Our APL server could simply send the requested file or it could proceed with a *challenge* to the browser saying that our site has restricted access and requires a user ID and password to be supplied before entry will be granted. This our server may do by sending the client an HTTP request for authorization looking something like:

```
HTTP/1.1 401 Authorization Required
Date: Fri, 24 Mar 2006 13:12:55 GMT
Server: Dyalog APL
WWW-Authenticate: Basic realm="User Area 100"
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1
1c0
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML>
<HEAD>
<TITLE>401 Authorization Required</TITLE>
</HEAD>
<BODY>
<H1>Authorization Required</H1>
<P>This server could not verify that you are authorized to access the document requested .. </P>
<HR>
<ADDRESS>Dyalog APL 10.0 Server</ADDRESS>
</BODY>
</HTML>
0
```

This command tells the browser to prompt the user for an ID and password. The arbitrary words "User Area 100" chosen by the server site programmer are displayed on the password dialog box to indicate the *realm* to which access is being offered. An HTML error message is embedded (with a 'parity check' 1c0). This message is to be displayed by the client in the event that the supplied credentials are insufficient.

Note that in order to specify which version of the HTML standard they conform to, all HTML documents should start with a *document type declaration* (informally, a "DOCTYPE"), which makes reference to a *document type definition* (DTD). Using the line  
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">  
is adequate for our purpose, and indeed it may be omitted entirely.

When an ID and password have been entered, the two strings, separated by a colon, are encoded and included in the form of an Authorization field in all further messages sent by the browser to the server. On this subject, RFC2616 says,

"A user agent that wishes to authenticate itself with a server-- usually, but not necessarily, after receiving a 401 response--does so by including an Authorization request-header field with the request. The Authorization field value consists of credentials containing the authentication information of the user agent for the realm of the resource being requested."

The server has an opportunity to validate credentials at the head of each GET request before processing the specific contents of the request URI and sending a response to the client browser. The credentials are sent in a base-64 encoded string in an Authorization field such as

Authorization: Basic QWxhZGRpbjpvGVuIHNlc2FtZQ==

The distributed PATCH workspace contains functions for encoding and decoding credentials. For example (noting the changed translation vector in `⎕NXLATE`):

```
enco 'Aladdin:open sesame' ↪ 'QWxhZGRpbjpvGVuIHNlc2FtZQ=='
```

and

```
deco 'QWxhZGRpbjpvGVuIHNlc2FtZQ==' ↪ 'Aladdin:open sesame'
```

where

```
▽ strg←deco code;raw;alph
[1] alph←⎕A,(26↑17↓⎕AV),⎕D,'+/'
[2] raw←→{(⎕DR w)11 ⎕DR w},⊆(6ρ2)τ(alph⊎code~'=')-⎕IO
[3] strg←82 ⎕DR(-8|ρraw)↑raw
▽
```

This gives a *Basic* level of security. There is also a higher *Digest* level of security as described in HTTP Authentication: Basic and Digest Access Authentication in <http://www.ietf.org/rfc/rfc2617.txt>.

### §§ 15.2.3 Running APL Functions on a Server

So a conversation between IE and a server could proceed as

```
IE Client: GET / HTTP/1.1...
APL Server: HTTP/1.1 200 OK...
IE Client: GET /images/tennis.gif HTTP/1.1...
IE Client: GET /banners/scoresheet.gif HTTP/1.1...
APL Server: HTTP/1.1 200 OK...
APL Server: HTTP/1.1 200 OK...
```

But negotiations between clients and servers are not restricted to requesting and supplying static pages. Often significant background processing is desired. The GET request type can fulfil both roles.

If the GET request URI does **not** contain a question mark then the URI is assumed to be an HTML file name whose content is to be returned. If, however, the URI **does** contain a question mark then a browser recognises this as a *query URI* that can perform operations with significant side effects.

Such a request URI may be embedded in an HTML page in a hyperlink such as

```
<a href="driller.RUN? title="drill down">Sage Driller</a>
```

This creates an element that becomes a hyperlink (with an optional 'hover box' title). In the event that a user clicks on "Sage Driller", the browser will send

```
GET driller.RUN? ...
```

On receipt, the server in `..\aplservice\server.dws` interprets this as a request to run a function `▽RUN▽` in a namespace called *driller*. The function must be defined dyadically and is automatically given a left argument of the name of the socket involved. The right argument consists of parameters following the `?`, and separated by `&` if there is more than one parameter in the argument. A number of parameters may be needed for a particular function, or none at all as in the case of *driller.RUN*. The other primary requirement of the APL function is that any result is in the form of an HTML string. The browser will display this HTML automatically on receipt.

Note that this syntax is not APL-specific. For example

```
<a id=2a class=q href="http://groups.google.co.uk/grphp?hl=en&tab=wg&ie=UTF-8">Groups</a>
```

is to be found on the Google front page.

By adding a new namespace with a top-level function adhering to the above syntax, new applications may be added to the server workspace in a very straightforward and elegant fashion. See, for example, functions *rain.Climate*, *rain.Fourier* or *CODEVIEW.FUNCTION*.

The POST request type may be used as an alternative way of initiating a function call in the server.

"The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line."

The POST request type is suitable for calling a function from an HTML form. For example, the `loan.htm` file contains the line

```
<form action="loan.RUN" method="POST" ...
```

In this case *loan.RUN* is recognised as a function call when a POST request is received from the client as a result of the user clicking on the form.

The `..\aplservice\server.dws` workspace supports both GET and POST requests. More examples of client-server negotiations may be found in `..\ws\FTP.dws` and `..\ws\PATCH.dws`.

A very useful tool for intercepting TCP interactions between a computer and the outside world may be downloaded free from <http://www.westbrooksoftware.com/tsdownload.shtml>.

## § 15.3 Internet Practicalities

### §§ 15.3.1 Domain Name Servers

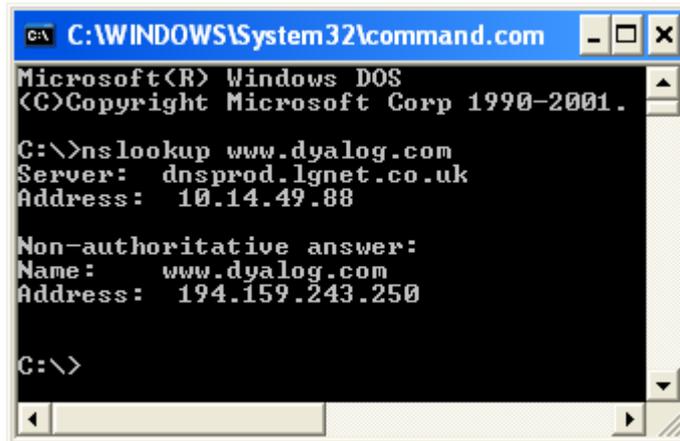
Web sites are better known by their names than by their IP addresses. For example, [www.dyalog.com](http://www.dyalog.com) is more memorable than 194.159.243.250. But an IP address in *LocalAddr* or *RemoteAddr* is essentially equivalent to a domain name in *LocalAddrName* or *RemoteAddrName*.

Either the address or the address name may be used in the specification of a *TCPSocket* property. A name is converted into the equivalent IP address by a *Domain Name Server* (DNS) which is always accessible

from an ISP via the Winsock API. A *TCP Socket* object has a *TCPGotAddr* event. This event is triggered when an address name is resolved into an IP address. For example

```
'S0' WC 'TCP Socket' ('RemoteAddrName' 'www.dyalog.com') ↓
    ('RemotePort' 80) ('Event' 'TCPGotAddr' 'show') ⌘
S0 TCPGotAddr
    S0.RemoteAddr
194.159.243.250
```

In Windows XP the DNS may be invoked by the nslookup utility which can be applied to any domain name to extract the underlying IP address.



Port numbers are also often referred to by the service names.

15.3.1.1 Create a socket with *LocalPortName* http and show the event message from *TCPGotPort*. Check the *LocalPort* property when the port name has been resolved.

### §§ 15.3.2 Firewalls and proxy Servers

If you are inside the walls of a business, the chances are that you have to go through a firewall every time you communicate with the outside world via your personal computer. A firewall is a proxy server that filters and controls the traffic between the company intranet, a trusted zone, and the outside Internet, which is not trustworthy and is teeming with parasites. A proxy server is both a client and a server. It acts as a server for all requests to the outside world from the intranet, and as a client to all Internet servers. It therefore offers a protective barrier between relative order and relative anarchy.

In this case, a proxy server responds to a request, not with HTTP/1.1 401 Authorization Required..., but with HTTP/1.1 407 Proxy Authentication Required... With a simple request for an Internet page

```
GET http://www.google.co.uk/ HTTP/1.0
Accept: */*
Accept-Language: en-gb
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Host: www.google.co.uk
Proxy-Connection: Keep-Alive
```

the response of a proxy server may be something like:

```
HTTP/1.1 407 Proxy Authentication Required ( The ISA Server requires authorization to fulfill the
request. Access to the Web Proxy service is denied. )
Via:1.1 OURPRXY
Proxy-Authenticate: Basic realm="Privileged User Access Area"
Pragma: no-cache
Cache-Control: no-cache
Content-Type: text/html
Content-Length: 2370
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML dir=ltr><HEAD><TITLE>The page cannot be displayed</TITLE>
...
```

to which a suitable reply, containing an encoded Proxy-Authorization field, could be

```
GET http://www.google.co.uk/ HTTP/1.0
Accept: */*
Accept-Language: en-gb
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Host: www.google.co.uk
Proxy-Connection: Keep-Alive
Proxy-Authorization: Basic QWxhZGRpbjpvcmVudHJlc2FtZQ==
```

and the proxy server's response, having checked the credentials might be

```
HTTP/1.1 200 OK
Via: 1.0 OURPRXY
Date: Thu, 30 Mar 2006 07:59:35 GMT
Content-Type: text/html
Cache-Control: private
Server: XYZ/2.2
...
```

The command Proxy-Authorization: Basic QWxhZGRpbjpvcmVudHJlc2FtZQ== must be included in all further requests to the proxy server from the client.

### §§ 15.3.3 An ISP running Dyalog.DLL

In order to have a web server that runs Dyalog APL code and that is accessible through the Internet, it is clearly necessary to have Dyalog APL running on a computer which is connected to the Internet.

Your Internet Service Provider may be willing, at a price, to run Dyalog APL but this gives you less control than you would wish, at least during the development phase. Rather than ask your ISP to host your site, why not host it yourself from an old computer in the 'demilitarised' shed outside?

[15.3.3.1](#) Please ask for the next module on **Web Clients**.

## Module16: APL Web Clients

### § 16.1 Getting to the outside World

Once you know the IP address of an Internet site, possibly acquired via the nslookup utility, you can check your connection with the ping utility. This is a very useful computer network testing tool described in <http://en.wikipedia.org/wiki/ping>. You may ‘ping’ another computer on your intranet, including your firewall proxy server, or, if the firewall does not get in the way, you can ping any computer on the Internet. For example,

```
C:\> ping 212.58.224.131
```

will check your connection to [www.bbc.co.uk](http://www.bbc.co.uk). Another useful network tool, tracert, may be employed to determine the route taken by a packet from your computer to another address, including your router, your modem, or the BBC web site:

```
C:\> tracert 212.58.224.131
```

#### §§ 16.1.1 Direct Connection through your Internet Service Provider

You might not want to host a web server. All you might want to do is get information directly from the Internet, or browse the Internet through a Dyalog APL *TCPSocket* object circumventing IE.

If you are connected directly to the Internet, by broadband or narrowband, wireless or wired, then you can set the *RemoteAddrName* of a socket to any domain name on the Internet, *RemotePort* 80, and you obtain an immediate connection.

**16.1.1.1** If you have a direct connection, try connecting a socket with *TargetState* ‘Open’ to *RemoteAddrName* ‘ww.bbc.co.uk’ or to any other domain on the Internet.

#### §§ 16.1.2 Proxy Servers and Firewalls

If you are connected to the Internet through a proxy server, then the only remote address that you may use is the address of the proxy itself. The proxy server then passes your request on to the outside world – the Internet. To discover the IP address of your proxy you can use nslookup on the proxy server name or make use of *RemoteAddrName* and *TCPGotAddr*. The name of the proxy is normally readily available – for example it is sometimes used for the *realm* in the login dialog box.

**16.1.2.1** If you connect through a proxy, try to connect a socket to it.

### § 16.2 Asking the Web

#### §§ 16.2.1 Connecting and sending the Question

**16.2.1.1** Write a function that returns a CRLF separated list of commands, ending with CRLF CRLF, eg

```
GET http://www.google.co.uk HTTP/1.1  
Host: www.google.co.uk
```

including a suitably encoded Proxy-Authorization command, if necessary,

```
Proxy-Authorization: Basic QWxhZGRpbjpvGvUHNlc2FtZQ==
```

**16.2.1.2** Create a socket on port 80 with *RemoteAddrName* ‘www.google.co.uk’, or the name of your proxy server if you have a firewall. Send the above commands when the *TCPConnect* event triggers.

Hint: See workspace [www.dws](http://www.dws), namespace #.BROWSER for guidance.

## §§ 16.2.2 Receiving and interpreting the Answer

16.2.2.1 Use `google.co.uk` in IE to search for `dragons+castles` and notice that the address bar now contains the string `http://www.google.co.uk/search?hl=en&q=dragons%2Bcastles&meta=`  
Put this string into a direct socket request via:

```
GET http://www.google.co.uk/search?hl=en&q=dragons%2Bcastles&meta= HTTP/1.1
Host: www.google.co.uk
```

Capture the return message from `TCPRecv` and check that the response begins with `HTTP/1.1 200 OK`.

16.2.2.2 Copy the HTML text from the received message above and, using Notepad, paste it into a new file called `hoo.htm`. Open IE and browse to that file using [File][Open]. Compare the display with the original Google results page. Note that pictures are sent in raw sockets and therefore must be received in sockets of `Style 'raw'`.

16.2.2.3 Dig out the Last Trade price for the Boeing Company from Yahoo using command line

```
http://finance.yahoo.com/q?s=BA&d=t
```

by way of a Dyalog APL stream socket.

16.2.2.4 Get some historical prices from Yahoo using command line

```
http://chart.yahoo.com/t?a=01&b=01&c=1900&d=05&e=18&f=2001&g=d&s=jdsu&y=0&z=jdsu&q=q
```

Use the function `▽htmlx▽` in the `DFNS.DWS` workspace, or any other method, to extract the numbers from the HTML table.

16.2.2.5 Send the following HTTP request, with appropriate `Proxy-Authorization:` or `Authorization:` header field, if necessary

```
GET http://finance.yahoo.com/bonds/market_summary HTTP/1.0
Host: www.yahoo.com
Proxy-Authorization: Basic QWxhZGRpbjpvGVuIHNIc2FtZQ==
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Accept: */*
Accept-Language: en-gb
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/3.0Gold (Win95; I)
```

and check that the response looks something like:



16.2.2.6 Please ask for the next module on [Dyalog.Net](http://Dyalog.Net) 😊.

## Module17: Dyalog.Net

### § 17.1 Revealing the .NET Framework

*Vade cum nostris*, but be prepared to meet some deeper shades of gray ☹.

#### §§ 17.1.1 Getting Microsoft .NET

The Microsoft .NET Framework is the new low-level platform on which all Windows applications, and, with the appearance of Windows Vista, even operating systems, are supposed to be built. Once upon a time Windows was built starting from DOS, the *Disk Operating System* software backbone of 80's PCs. DOS itself was probably (see <http://museum.sysun.com/museum/cpmhist.html>) modified CP/M. CP/M was originally developed around 1975 for Intel 8080 chips and Zilog Z80 chips on Intel's 8080 emulator under DEC's TOPS-10 operating system.

Dyadic Systems and Zilog Inc. developed Dyalog APL as a joint venture around 1982 ☺.

By the beginning of the 90's, the Microsoft Windows *Application Programmers Interface* (API) had appeared, based essentially on C functions in *Dynamic Link Libraries* (.DLL files). This was intended to replace DOS completely as the new basis for application development. Windows and its API slowly evolved away from DOS. Around the mid 90's, VB/VBA and OLE/COM were introduced as the new foundations (language *cum* interface) upon which all user level applications were supposed to be built. These dreams are still being realised in the big wide IT world of today.

But now the intension is to replace all of these platforms with a new platform, called Microsoft .NET, which is built 'over' the Windows API and is based essentially on libraries of C# functions. Other languages, such as Dyalog APL, VB.NET, C++, Jscript, COBOL, FORTRAN, Python, RPG, Pascal, SmallTalk, Perl, Oberon and Eiffel, can also contribute libraries as equal partners of C# because of the **common language specification** at the entrance to Microsoft .NET Framework functionality. The .NET base class libraries, or *assemblies*, comprising over 30 .DLL files, contain a huge array of functions embedded in *classes*, grouped within *namespaces* by area of application. These functions can all be called through a highly object-oriented approach by a growing a number of programming languages, including Dyalog APL.

Documentation is found at <http://msdn2.microsoft.com/netframework/aa569294>.

The Dyalog APL interface to .NET (Dyalog.Net) has been available from Dyalog version 9.5 onwards. To run .NET a computer requires Windows 2000 or Windows XP Professional or Vista together with the Microsoft .NET Framework (version 1 + SP1 or version 2). Both are freely installable from Microsoft downloads at <http://www.microsoft.com/downloads/Search.aspx?displaylang=en> via **dotnetfx.exe**. The .NET platform is an integral part of Vista and of subsequent operating systems from Microsoft.

You can check your .NET framework installation level from [Control Panel][Add or Remove Programs], or from registry entry HKEY\_LOCAL\_MACHINE\Software\Microsoft\NET Framework Setup\NDP\v1.1.4322\SP value, or by looking in directory ..\WINDOWS\Microsoft.NET\...

#### §§ 17.1.2 Assemblies (à), Namespaces (ñ) and Classes (¢)

When Microsoft .NET is installed on your computer, you will find a subdirectory called something like ..\Microsoft.NET\Framework\v1.1.4322\ in your Windows directory. This directory contains about 30 DLLs that together (or in stand-alone subsets) form the *substance* of the .NET Framework and contain most of the Microsoft-supplied functionality available to .NET programmers.

## ===== ASSEMBLIES (à) in the .NET Framework =====

<ul style="list-style-type: none"> <li>mscorlib.dll</li> <li>System.dll</li> <li>System.Configuration.Install.dll</li> <li>System.Data.dll</li> <li>System.Data.OracleClient.dll</li> <li>System.Design.dll</li> <li>System.DirectoryServices.dll</li> <li>System.Drawing.dll</li> <li>System.Drawing.Design.dll</li> <li>System.EnterpriseServices.dll</li> <li>System.EnterpriseServices.Thunk.dll</li> <li>System.Management.dll</li> <li>System.EnterpriseServices.Thunk.dll</li> <li>System.Management.dll</li> <li>System.Messaging.dll</li> <li>System.Runtime.Remoting.dll</li> </ul>	<ul style="list-style-type: none"> <li>System.Runtime.Serialization.Formatters.Soap.dll</li> <li>System.Security.dll</li> <li>System.ServiceProcess.dll</li> <li>System.Web.dll</li> <li>System.Web.Mobile.dll</li> <li>System.Web.RegularExpressions.dll</li> <li>System.Web.Services.dll</li> <li>System.Windows.Forms.dll</li> <li>System.XML.dll</li> <li>cscompmgd.dll</li> <li>ISymWrapper.dll</li> <li>Microsoft.Jscript.dll</li> <li>Microsoft.VisualBasic.dll</li> <li>Microsoft.Vsa.dll</li> </ul>
---	--

All word phrases here written in green may be used, in one way or another (in character string arguments, as methods/functions, as properties/variables, ...) in [Dyalog.Net](#) code. There are over 14,000 new unique dot-qualified strings available for inclusion in your programs in [Dyalog.Net](#).

It's like going from the Roman alphabet to Chinese characters, or from  $\square AV$  to Unicode, or from {the set of all letters} to {the set of all words}, or from the safe set of integers,  $\mathbb{Z}$ , all-be-they of infinite number aleph null ( $\aleph_0$ ), to the wild real numbers,  $\mathbb{R}$ , or the beautiful complex numbers,  $\mathbb{C}$ , both of number aleph one ( $\aleph_1$ )!

The .NET framework is highly object-oriented. An *instance* of an object is generally created from a *class* which holds the object creation code. A class represents a species of object - like the Dandelion (*Taraxacum officinale*) represents all the dandelions in your garden. Essentially, each **.NET assembly** (a logical .DLL) contains a number of **.NET namespaces** that each contains many **.NET classes** (or object blueprints). Classes contain *members* – these members include **methods, properties, fields** and **events**.

Here is a list of almost all the .NET namespaces from almost all the DLLs in the Microsoft .NET Framework (version 1.1). Their content comprises the .NET base class library.

## ===== NAMESPACES (ñ) in .NET Framework =====

<ul style="list-style-type: none"> <li>System</li> <li>System.CodeDom</li> <li>System.CodeDom.Compiler</li> <li><b>System.Collections</b></li> <li>System.Collections.Specialized</li> <li>System.ComponentModel</li> <li>System.ComponentModel.Design</li> <li>System.ComponentModel.Design.Serialization</li> <li>System.Configuration</li> <li>System.Configuration.Assemblies</li> <li>System.Configuration.Install</li> <li>System.Data</li> <li>System.Data.Common</li> <li>System.Data.Odbc</li> <li>System.Data.OleDb</li> <li>System.Data.OracleClient</li> <li>System.Data.SqlClient</li> <li>System.Data.SqlServerCE</li> <li>System.Data.SqlTypes</li> <li>System.Diagnostics</li> <li>System.Diagnostics.SymbolStore</li> <li>System.DirectoryServices</li> <li>System.Drawing</li> <li>System.Drawing.Design</li> <li>System.Drawing.Drawing2D</li> <li>System.Drawing.Imaging</li> </ul>	<ul style="list-style-type: none"> <li>System.Drawing.Printing</li> <li>System.Drawing.Text</li> <li>System.EnterpriseServices</li> <li>System.EnterpriseServices.CompensatingResourceManager</li> <li>System.EnterpriseServices.Internal</li> <li>System.Globalization</li> <li>System.IO</li> <li>System.IO.IsolatedStorage</li> <li>System.Management</li> <li>System.Management.Instrumentation</li> <li>System.Messaging</li> <li>System.Net</li> <li>System.Net.Sockets</li> <li>System.Reflection</li> <li>System.Reflection.Emit</li> <li>System.Resources</li> <li>System.Runtime.CompilerServices</li> <li>System.Runtime.InteropServices</li> <li>System.Runtime.InteropServices.CustomMarshalers</li> <li>System.Runtime.InteropServices.Expando</li> <li>System.Runtime.Remoting</li> <li>System.Runtime.Remoting.Activation</li> <li>System.Runtime.Remoting.Channels</li> <li>System.Runtime.Remoting.Channels.Http</li> <li>System.Runtime.Remoting.Channels.Tcp</li> <li>System.Runtime.Remoting.Contexts</li> </ul>
---	---

```
System.Runtime.Remoting.Lifetime
System.Runtime.Remoting.Messaging
System.Runtime.Remoting.Metadata
System.Runtime.Remoting.Metadata.W3cXsd2001
System.Runtime.Remoting.MetadataServices
System.Runtime.Remoting.Proxies
System.Runtime.Remoting.Services
System.Runtime.Serialization
System.Runtime.Serialization.Formatters
System.Runtime.Serialization.Formatters.Binary
System.Runtime.Serialization.Formatters.Soap
System.Security
System.Security.Cryptography
System.Security.Cryptography.X509Certificates
System.Security.Cryptography.Xml
System.Security.Permissions
System.Security.Policy
System.Security.Principal
System.ServiceProcess
System.Text
System.Text.RegularExpressions
System.Threading
System.Timers
System.Web
System.Web.Caching
System.Web.Configuration
System.Web.Hosting
System.Web.Mail
System.Web.Mobile
System.Web.Security
System.Web.Services
System.Web.Services.Configuration
System.Web.Services.Description
System.Web.Services.Discovery
System.Web.Services.Protocols
System.Web.SessionState
System.Web.UI
System.Web.UI.Design
System.Web.UI.Design.WebControls
System.Web.UI.HtmlControls
System.Web.UI.MobileControls
System.Web.UI.MobileControls.Adapters
System.Web.UI.WebControls
System.Windows.Forms
System.Windows.Forms.Design
System.Xml
System.Xml.Schema
System.Xml.Serialization
System.Xml.XPath
System.Xml.Xsl
Microsoft.CSharp
Microsoft.JScript
Microsoft.VisualBasic
Microsoft.Vsa
Microsoft.Win32
```

Each namespace contains a number of classes that can instantiate objects. Altogether there are over 700 classes in Microsoft .NET. One library of namespaces, *mscorlib.dll*, contains the *core* classes from which many other common classes inherit behaviour and characteristics. The .NET namespaces found in *mscorlib.dll* are:

### ===== NAMESPACES (ñ) in *mscorlib.dll* Assembly =====

```
System
System.Collections
System.Configuration.Assemblies
System.Diagnostics
System.Diagnostics.SymbolStore
System.Globalization
System.IO
System.IO.IsolatedStorage
System.Reflection
System.Reflection.Emit
System.Resources
System.Runtime.CompilerServices
System.Runtime.InteropServices
System.Runtime.InteropServices.Expando
System.Runtime.Remoting
System.Runtime.Remoting.Activation
System.Runtime.Remoting.Channels
System.Runtime.Remoting.Contexts
System.Runtime.Remoting.Lifetime
System.Runtime.Remoting.Messaging
System.Runtime.Remoting.Metadata
System.Runtime.Remoting.Metadata.W3cXsd2001
System.Runtime.Remoting.Proxies
System.Runtime.Remoting.Services
System.Runtime.Serialization
System.Runtime.Serialization.Formatters
System.Runtime.Serialization.Formatters.Binary
System.Security
System.Security.Cryptography
System.Security.Cryptography.X509Certificates
System.Security.Permissions
System.Security.Policy
System.Security.Principal
System.Text
System.Threading
Microsoft.Win32
```

In the entire .NET Framework base class library there are over 30 assemblies, altogether containing over 100 namespaces. These 100 or so namespaces together contain over 700 classes. These 700 or so base classes (and their instantiated objects) are all immediately available to [Dyalog.Net](#) programmers.

The namespaces are organised hierarchically. This avoids name clashes and helps to approximately categorize the **functional** (or **objective**) nature of the contents.

The *System.Collections* namespace in *mscorlib.dll*, for example, contains about 25 classes.

==== CLASSES (ϕ) in *System.Collections* Namespace =====

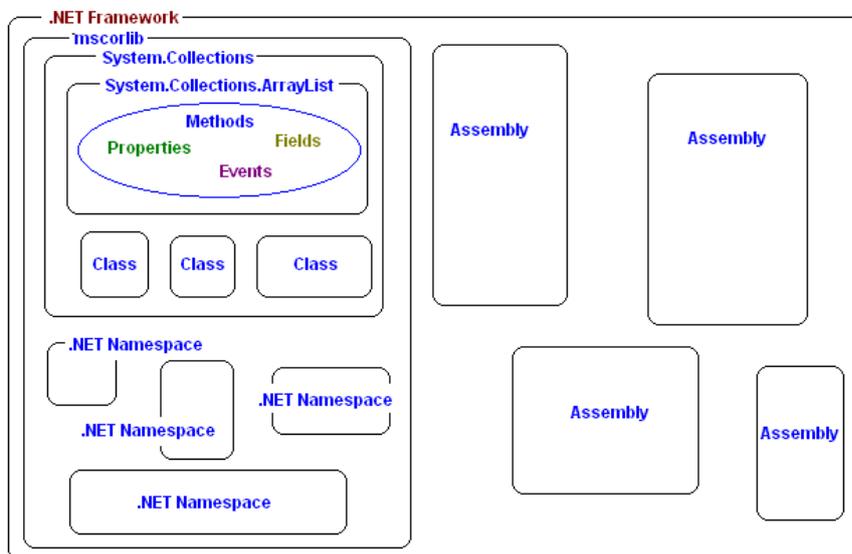
**System.Collections.ArrayList**

- System.Collections.BitArray
- System.Collections.CaseInsensitiveComparer
- System.Collections.CaseInsensitiveHashCodeProvider
- System.Collections.CollectionBase
- System.Collections.Comparer
- System.Collections.DictionaryBase
- System.Collections.DictionaryEntry
- System.Collections.Hashtable
- System.Collections.ICollection
- System.Collections.IComparer
- System.Collections.IDictionary
- System.Collections.IDictionaryEnumerator

- System.Collections.IEnumerable
- System.Collections.IEnumerator
- System.Collections.IHashCodeProvider
- System.Collections.IKeyComparer
- System.Collections.IKeyedCollection
- System.Collections.IList
- System.Collections.KeyComparer
- System.Collections.Queue
- System.Collections.ReadOnlyCollectionBase
- System.Collections.SortedList
- System.Collections.Stack

Classes are used to create objects, which have properties and methods. In the interests of application efficiency, some information about a class is not carried around with the class itself but is kept separately in its *MetaData* which is stored in the assembly's corresponding type library, or .TLB file. Typically, a class itself has no *GetMethods* method. Instead, you have to use the *GetType* method to instantiate a *reflection* of the original object in order to list the methods. The reflected object may be of 'data Type' *System.RuntimeType*. This object inherits the niladic *ToString* method that reports the data Type of the original object as, for example, *System.Collections.ArrayList* in the case of an instance of *ArrayList*. It also has *GetMethods*, *GetProperties* and *GetFields* methods that describe the members of the original instance of the *System.Collections.ArrayList* class. These methods access the *MetaData* which contains member names, data Types and method calling information.

Classes contain Methods, Properties, Fields and Events.



If we create an instance of the *System.Object* class then the default display form of the instance also happens to be *System.Object*, and the *dataType* of the instance is also called *System.Object*. However, these three names are logically distinct. The name of the class need not be identical to the *dataType* description of the instance, which need not be identical to the object display form. Thus, for example, the *GetType* method of an instance of the *ArrayList* class returns an instance of an object of the *System.Type* class whose *dataType* is reported as *System.RuntimeType* and whose display form is the full class name of the original instance, viz *System.Collections.ArrayList*. The *ToString* method is inherited by most objects. Its result is of type *System.String* and is returned to APL as a simple character vector. This is the default display form of an object and often spells out the

dataType (or simply the type) of the object. In Dyalog version 11, the display form of an object may be set to any arbitrary character array via the new System Function `⎕DF`.

17.1.2.1 What assembly do you think the *System.ServiceProcess* namespace is probably in? What namespace contains the *System.ServiceProcess.ServiceControllerPermissionEntry* class? (Version 11 has extended `⎕NL` to facilitate this latter question.)

Hint: Use Google or consult <http://msdn.microsoft.com/library/>.

### §§ 17.1.3 Using `⎕USING`

Occasionally namespaces have members that are spread across multiple assemblies. In particular, the *System* namespace is spread over *mscorlib.dll* and *system.dll*. A .NET namespace is a logical design-time naming convenience, used mainly to organize classes in a single hierarchical structure. From the viewpoint of the runtime, there are no actual namespaces. Nevertheless, treating an assembly as a namespace receptacle is convenient. But it is therefore not easy programmatically to get a list of namespaces from an assembly name and so the namespace name together with its assembly origin must be specified before it can be utilized.

Dyalog APL contains a new system variable called `⎕USING`. It is a bit like `⎕PATH`, which redirects APL to the location of some program that is in another APL namespace. `⎕USING` (closely analogous to the using directive in C# or the import directive in VB.NET) redirects APL to the location of some class that is in some particular .NET namespace that is in some particular assembly.

`⎕USING←'ñ1,à1' 'ñ2,à2' ...`      *a* Use .NET namespace *ñ<sub>i</sub>* from assembly *à<sub>i</sub>*

In a new clear workspace `⊢⎕USING≡0ρc''`. APL namespaces and programs inherit their local value of `⎕USING` from the parent space, as does `⎕PATH`. Unlike `⎕PATH`, the System Variable `⎕USING` cannot be saved in the .DSE Session file.

`⎕USING` is a vector of character vectors (APL 'dataType' `VecCharVec`) each character vector of which contains two parts separated by a comma. The first part specifies the case-sensitive name of a .NET namespace, and the second part specifies the name of a DLL file, thus

`⎕USING←,c'NetNamespace,C:\..\Assembly.dll'`

The primary assembly in the .NET framework, containing the most commonly used namespaces, is *mscorlib.dll*. The namespace in this assembly with the most commonly used classes is named the *System* namespace. Its content is exposed to APL by setting

`⎕USING←,c'System,mscorlib.dll'`

This particular namespace, and only this space, may be exposed by simply typing `⎕USING←'System'`, or even just `⎕USING←''` as long as you thereafter prefix everything with "*System.*". We shall often use the explicit verbose form for clarity, and because all other namespaces have to be treated this way.

This establishes the basic starting point for [Dyalog.Net](#). If, for example, classes in the namespace *System.Windows.Forms* are also to be invoked in [Dyalog.Net](#) code then the entry

`⎕USING,←c'System.Windows.Forms,System.Windows.Forms.dll'`

must be added to the local list of namespace paths.

17.1.3.1 Set `⎕USING` in such a way that all the classes in .NET namespace *System* and .NET namespace *System.Windows.Forms* may be used directly from Dyalog APL.



```
1287 35 4 1053 872 186
3↑>AppDomain.CurrentDomain.GetAssemblies.GetExportedTypes
System.Object System.ICloneable System.Collections.IEnumerable
```

```
VRV←AppDomain.CurrentDomain.GetAssemblies.GetExportedTypes.Module
```

Each of these objects is of data type *System.Type* and has a *Module* property that returns an object of data type *System.Reflection.Module* whose default display form describes the assembly in question.

```
3↑>AppDomain.CurrentDomain.GetAssemblies.GetExportedTypes.Module
CommonLanguageRuntimeLibrary CommonLanguageRuntimeLibrary CommonLanguageRuntimeLibrary
⊂CS⇒AppDomain.CurrentDomain.GetAssemblies.GetExportedTypes.Module
)NS
#. [System.AppDomain].[System.Reflection.Assembly].[System.RuntimeType].[System.Reflection.Module]
```

17.2.1.1 Objects of data type *System.Type* have some properties that return simple string or Boolean values. Investigate the properties *IsClass*, *IsPublic*, *Name*, *Namespace* and *FullName*. Write an expression that returns a vector of objects of data type *System.RuntimeType* which describe all the public classes in the first assembly in the current domain.

17.2.1.2 Check that the *System.Math* class is available for use when `⊂⊂USING≡''`.  
Hint: Look at the 'Name Category' (*⊂NC*) of *c'Math'*.

## §§ 17.2.2 Examining Methods

In Microsoft .NET there are no functions or variables outside classes. Some methods in some .NET classes, such as those considered in §§17.2.1, may be used without explicitly creating instances. They have some public methods and properties accessible directly from the class namespace. *System.Math* is such a class. It is a container for some simple mathematical functions that may be called directly from the class. A few basic mathematical methods and fields are immediately available from the *Math* class, assuming that the *System* namespace is on the `⊂USING` path.

```
Math.((Sin 0.5)(Asin 0.5)) ⊂ 1 ^1∘0.5 ⊂ 0.4794.. 0.5235...
Math.((Log 0.5)(Exp 0.5)) ⊂ (⊗0.5)(*0.5) ⊂ ^0.6931.. 1.648...
Math.Abs''^5+19 ⊂ |^5+19 ⊂ 4 3 2 1 0 1 2 3 4
Math.(E PI) ⊂ (*1)(∘1) ⊂ 2.7182.. 3.1415...
```

Remember that `..` and `...` were defined as single symbols.

17.2.2.1 Load the MetaData for the *Math* class and try using some of the methods in the class.  
Hint: Right click MetaData in WS Explorer to load the .TLB file  
Make sure [View][Type Libraries] in WS Explorer is checked.

Most .NET classes are used by creating an instance of the class. In .NET, to examine programmatically the information associated with members of a class (methods, properties, etc.), you have to create an instance of the class and then use the *GetType* method to create an associated object of data type *System.Type* and examine the results of its *GetMethods* and *GetProperties* methods. Information about the original object-generating class is extracted from the assembly Type Library file (.TLB) and is reported via a *Type* object instantiated from the *Type* class.

In typical OO style, the .NET Framework deals in classes and objects and methods and properties... (Events may be used as methods via `⊂NQ ...` and from version 10.1 onwards it is possible to declare events on *NetType* objects created with Dyalog APL.)

```
RSC←⊂NEW ⊕ ...
```

Ⓐ Create (instantiate) an instance of the class

In Dyalog version 11 instances may be created from classes with the `NEW` system function. This monadic function takes a class as the first parameter of its argument, followed by a second parameter if required. For example, assuming the `System` namespace is visible,

```
DT←NEW DateTime (2006 7 4)
```

creates a new instance of the `DateTime` class specifically relating to American Independence Day. The `MethodList` and `PropList` properties of this instance return a list of its methods and properties. The list of methods (with further data type information) may be discovered from `MetaData` or, equivalently, from expressions such as

```
2↑DT.GetType.GetMethods @
System.DateTime Add(System.TimeSpan) System.DateTime AddDays(Double)
```

17.2.2 Find the minimum and maximum dates allowed for a `DateTime` object.

Returning to the `ArrayList` class, given that `USING` is set appropriately, eg

```
USING←'System,mscorlib.dll' ↵
      'System.Collections,System.Collections.dll' ℑ
```

an instance of the `ArrayList` class may be created by the statement

```
AL←NEW ArrayList
```

and the methods associated with this object may be obtained using its `MethodList` property.

Alternatively, a more detailed description of each method may be obtained from the `GetMethods` method of the reflected object. Unsurprisingly (by now) the `GetMethods` method returns a vector of `objects`. The display form of each object (data type `System.Reflection.RuntimeMethodInfo`) gives information about a method and its syntax.

```
2↑AL.GetType.GetMethods @
Int32 get_Capacity() Void set_Capacity(Int32)
```

Thus each class (object creation program inside its own space) contains a number of methods (functions). For example, the `System.Collections.ArrayList` class contains the following methods.

## == METHODS in `System.Collections.ArrayList` Class ==

```
System.Collections.ArrayList Adapter (System.Collections.IList)
      Void AddRange (System.Collections.ICollection)
      Int32 Add (System.Object)
      Int32 BinarySearch (Int32, Int32, System.Object, System.Collections.IComparer)
      Int32 BinarySearch (System.Object, System.Collections.IComparer)
      Int32 BinarySearch (System.Object)
      Void Clear ()
      System.Object Clone ()
      Boolean Contains (System.Object)
      Void CopyTo (Int32, System.Array, Int32, Int32)
      Void CopyTo (System.Array, Int32)
      Void CopyTo (System.Array)
      Boolean Equals (System.Object)
      System.Collections.ArrayList FixedSize (System.Collections.ArrayList)
      System.Collections.IList FixedSize (System.Collections.IList)
System.Collections.IEnumerator GetEnumerator (Int32, Int32)
System.Collections.IEnumerator GetEnumerator ()
      Int32 GetHashCode ()
      System.Collections.ArrayList GetRange (Int32, Int32)
      System.Type GetType ()
      Int32 IndexOf (System.Object, Int32, Int32)
      Int32 IndexOf (System.Object, Int32)
      Int32 IndexOf (System.Object)
      Void InsertRange (Int32, System.Collections.ICollection)
      Void Insert (Int32, System.Object)
      Int32 LastIndexOf (System.Object, Int32, Int32)
      Int32 LastIndexOf (System.Object, Int32)
      Int32 LastIndexOf (System.Object)
System.Collections.ArrayList ReadOnly (System.Collections.ArrayList)
      System.Collections.IList ReadOnly (System.Collections.IList)
```

	Void	RemoveAt	(Int32)
	Void	RemoveRange	(Int32, Int32)
	Void	Remove	(System.Object)
System.Collections.ArrayList	Repeat		(System.Object, Int32)
	Void	Reverse	(Int32, Int32)
	Void	Reverse	()
	Void	SetRange	(Int32, System.Collections.ICollection)
	Void	Sort	(Int32, Int32, System.Collections.IComparer)
	Void	Sort	(System.Collections.IComparer)
	Void	Sort	()
System.Collections.ArrayList	Synchronized		(System.Collections.ArrayList)
System.Collections.IList	Synchronized		(System.Collections.IList)
System.Array	ToArray		(System.Type)
System.Object[]	ToArray		()
System.String	ToString		()
	Void	TrimToSize	()
	Int32	get_Capacity	()
	Int32	get_Count	()
	Boolean	get_IsFixedSize	()
	Boolean	get_IsReadOnly	()
	Boolean	get_IsSynchronized	()
System.Object	get_Item		(Int32)
System.Object	get_SyncRoot		()
	Void	set_Capacity	(Int32)
	Void	set_Item	(Int32, System.Object)

Names of methods are repeated in this list when dataTypes of argument parameters vary. Each line specifies one way in which the method may be called. The situation often arises in APL, but behind the scenes. Consider, for example, the dyadic primitive functions  $\iota$  and  $\sim$ . Their arguments may be numeric or character. Under the covers, APL checks which and applies the required algorithm. Dyadic  $\rho$  can accommodate a right argument of many different dataTypes and, as of version 11, the arguments to  $\wedge$  and  $\vee$  may be Boolean or integer. The arguments to  $!$  may be integer or real, etc... Unlike APL (and VBScript ...), but like most mainstream low-level programming environments such as FORTRAN, VB and C#, .NET requires us to be more explicit about possible types of arguments to, and results of, methods. In fact it would be useful to include in APL documentation all the explicit calling options associated with each primitive function. However, to specify the precise structure of every intermediate array in an real APL application would be a thankless task.

$\square$  *NEW* is not the only way in which instances of classes may be created. Classes often have methods that return instances. For example, the *DateTime* class has a property (niladic function) called *Now*. *Now* returns an object of dataType *System.DateTime*.

17.2.2.3 Use the *IsLeapYear* method of an instance of the *DateTime* class created by the *Today* property to determine whether or not the year 3000 is a leap year.

### §§ 17.2.3 Examining Properties

Normally, classes instantiate objects whose methods and properties are then used and changed. Consider, *DateTime*  $\phi$ , in *System*  $\bar{n}$ , in *mscorlib.dll*  $\hat{a}$ . This class has a property (niladic function) called *Now*. *Now* returns an object of dataType *System.DateTime*. (This is the *dataType* we met in Module 0. In .NET there is a *Type* class whose purpose is to yield object dataType information.)

$DT \leftarrow DateTime.Now$

*DT* is an object with about 90 methods and 59 properties, as verified by

$\rho \text{ `` } DT.(MethodList PropList) \downarrow (,90)(,59)$

and whose dataType is discovered from the default display form (see  $\square$ DF) of the object returned by the (in this case niladic) *GetType* method (which is inherited ubiquitously from *System.Object*  $\phi$ ).

$\bar{\nabla} DT.GetType \downarrow 'System.DateTime'$

*GetType* returns an object of type *System.Type*. This object has an *Assembly* property whose display form contain the name of the assembly from which the current type has come.

```
8↑DT.GetType.Assembly ↪ 'mscorlib'
```

(Note that the *Type* class has a monadic *GetType* method that takes a *String* argument.)

The object returned by the *Assembly* property of the instance of the *Type* class representing *DT*,

```
Ass←DT.GetType.Assembly
```

itself has a *GetTypes* method that returns a vector of objects (*RVec*) representing all classes in the assembly.

```
ρAll←Ass.GetTypes ↪ 2373
```

Some of these classes are *Enumerations*. Their principal purpose is to supply alternate names for values of an underlying primitive instance. All the objects in vector *All* have a niladic method, *IsEnum*, which returns a Boolean value indicating whether the corresponding object in *All* is an Enumeration.

```
ρEnums←(All.IsEnum)/All ↪ 384
```

Taking the first such type, the *GetFields* method gets all the field names for the first Enum in the list.

```
ρEnums[1].GetFields θ ↪ 17
```

A field is a member of an object or class and represents a variable associated with the object or class. For example, if the 23<sup>rd</sup> enumeration is the *System.DayOfWeek* enumeration

```
↑↑Enums[23] ↪ 'System.DayOfWeek'
```

then the fourth field in the enumeration happens to be the *Tuesday* public static field.

```
↑4⇒Enums[23].GetFields θ ↪ 'System.DayOfWeek Tuesday'
```

17.2.3.1 Use an instance of *DateTime*  $\phi$  to find the day of the week today.

17.2.3.2 Verify that the object returned by *Assembly* (of data type *System.Reflection.Assembly*) has a *Location* property that returns a string containing the directory in which the assembly resides.

17.2.3.3 Find how many classes there are in *System.Web.dll*.

Hint: Create a new instance of, say, *System.Web.UI.Control* or *System.Web.Mail.MailMessage*.

As with object methods, the dataTypes associated with object properties make up an essential part of their specification. Therefore, in the pursuit of clarity, we talk about **dataType** – an adjective describing what type of data structure an object conforms to. You might think of it as a very sophisticated (albeit non-existent) version of monadic  $\square DR$ . For simple objects like the number 9, the dataType might be *System.Int16*. *System.Int16* is actually, in most situations, a *value type* - a light-weight class which is treated as a value rather than a full-blown class. On the other hand, the dataType of a more complicated object such as an instance of the *ArrayList* class is usually described in the same words as the namespace-qualified name of the class itself - *System.Collections.ArrayList*. *System.Collections.ArrayList* is called a *reference type* – a full-blown object passed around by references to it (shallow copies) rather than by making a genuine duplicates (deep copies).

### = PROPERTIES in *System.Collections.ArrayList* Class =

Int32	Capacity	
Int32	Count	
Boolean	IsFixedSize	
Boolean	IsReadOnly	
Boolean	IsSynchronized	
System.Object	SyncRoot	
System.Object	Item	[Int32]

Note that the *Item* property looks different from the others as it seems to take an argument! *Properties* are often like shared variables and are accessed through *get\_..* and *set\_..* control functions. *Fields* are more like

simple APL variables. *Methods* are like locked monadic or niladic functions – sometimes chameleon-like, *ie* either! *Events* are treated like methods in *Dyalog.Net* and cannot as yet be assigned callback functions.

The *GetProperties* method returns objects representing the properties of a *DateTime* object, with their dataTypes.

```

⌈3>DT.GetType.GetProperties ⍉ ↪ 'System.DayOfWeek DayOfWeek'
⍉DTP←{(⍉ρρω)1ρω}DT.GetType.GetProperties ⍉ ↪ 16 1

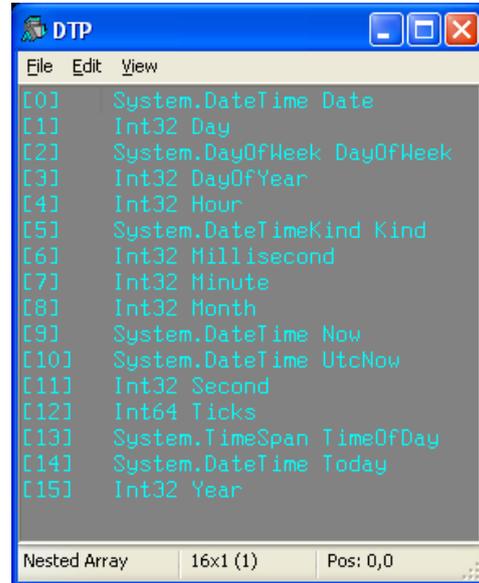
```

Thus the *DayOfWeek* property returns an object of type *System.DayOfWeek*, whose display form is the actual day of the week relating to the instance date, whereas *Day* contains a simple integer day number (Int32 => ZSc) relating to the instance date.

```

DT.DayOfWeek.ToString ⍉ ⌈ CVec
Tuesday
⌈DT.DayOfWeek ⌈ CVec
Tuesday
⌈FMT DT.DayOfWeek ⌈ CMat
Tuesday
DT.DayOfWeek ⌈ RSc
Tuesday

```



17.2.3.4 With a new instance of *DirectoryInfo*  $\phi$  from *System.IO*  $\bar{n}$ , call the *GetFiles* method with argument *'\*.\*'* to get (objects representing) all the files in a given DOS directory. Then read the *Name* and *CreationTime* properties of the vector of instances of data type *System.IO.FileInfo* to access the file details.

## § 17.3 Digging into .NET

### §§ 17.3.1 Windows Forms

Now that we know how to create instances of .NET classes by *USING* the *System* namespace

```

(NEW DateTime(3+TS)).ToString ⍉ ↪ '11/04/2006 00:00:00'

```

and understand that objects may be created with different *constructors*

```

(NEW DateTime(6+TS)).ToString ⍉ ↪ '11/04/2006 10:07:21'

```

and can recognise some different categories of objects

```

DI←NEW IO.DirectoryInfo(←'C:\windows')∘NC←'DI' ↪ 9.3

```

and appreciate some of the idiosyncrasies and niceties of .NET by way of Dyalog calling syntax

```

(>DI.(GetFiles←'*.exe')).FullName ↪ 'C:\windows\alcrmv.exe'

```

and know how to read syntax from MetaData (being wary of *WS FULL* for this large assembly), we should be able to build applications based on .NET classes. Workspaces supplied with Dyalog in directory *..\samples\winforms\* give some excellent examples as does the *Dyalog.Net Interface Guide*. Here we do little more than skim the surface.

In order to create a Form in .NET it is necessary to access *System.Windows.Forms*  $\bar{n}$ .

```

USING←'System.Windows.Forms, System.Windows.Forms.dll'

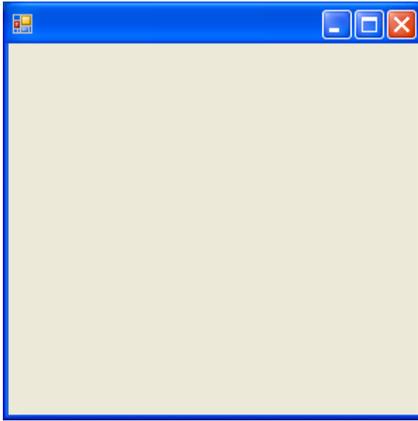
```

We can then immediately create a *Form*, and make it *Visible*

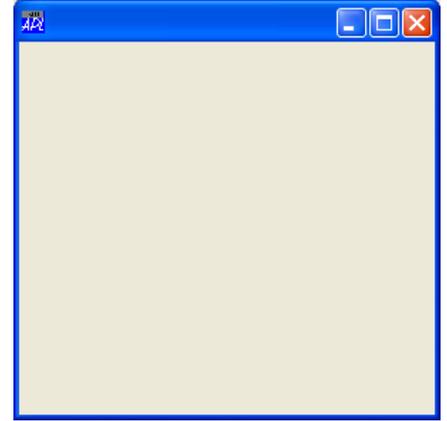
```

F←NEW Form ∘ F.Visible←1

```



compared with 'F2' [WC] Form'



The only visible difference is the default Icon. The differences between Windows GUI and Windows .NET Forms begin to diverge from this close (guess why;-) start.

The GUI *Form* has an *OnTop* property which becomes *TopMost* in .NET. The *Caption* property becomes the *Text* property and the *Size* and *Posn* properties are each a combination of two properties:

```
F.(Height Width)←200 300  A was Size
F.(Top Left)←100 200      A was Posn
```

There is also a *Location* property that has dataType *System.Drawing.Size*. This is more like the old *Posn* property in the sense that it accepts the *Top* and the *Left* coordinates in one argument. But in order to achieve this we have to create an instance of *System.Drawing.Size*  $\phi$  via *Point*  $\phi$ . One way of constructing an instance of *System.Drawing.Point*  $\phi$  is with a [NEW] object parameter of dataType *System.Drawing.Size* which is what we are trying to create in the first place! Luckily there is also a constructor with (Int32, Int32) for X and Y.

```
[USING,←←'System.Drawing, System.Drawing.dll'
F.Location
Pt←[NEW Point (10 10)
F.Location←Pt                A was Posn
```

17.3.1.1 Set the *Form* size in one statement using the *ClientSize* property.

A *Button* object, an instance of *Button*  $\phi$ , of dataType *System.Windows.Forms.Button* and of display form *System.Windows.Forms.Button*, *Text:...*, may be created by

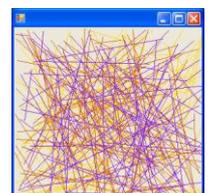
```
B←[NEW Button
```

.NET does not really have well-defined namespace hierarchies. For naming convenience, namespaces appear to be arranged hierarchically, but in fact, if a namespace called *A.B.C.D* exists in .NET then this does **not** imply that any of *A*, *A.B* or *A.B.C* has to exist. The .NET way of assigning a parent-child relationship to the *Form* and *Button* is by way of the *Controls* property of a *Form* which returns an instance of the class *System.Windows.Forms.Control*. This object has an *Add* method.

```
→F.Controls.MethodList ← 'Add'
```

This method takes an object as its argument and adds it to the collection of controls comprising the children of *F*.

```
F.Controls.Add B
```



17.3.1.2 Trace the `vscribble` function below and use *MetaData* to verify the comments.

```

▽ scribble;F;GR;PB
[1]  □USING+'System.Windows.Forms,System.Windows.Forms.dll'
[2]  □USING,←c'System.Drawing,System.Drawing.dll'
[3]  F←□NEW Form
[4]  F.Visible←1
[5]  PB←□NEW PictureBox
[6]  PB.Size←F.Size
[7]  F.Controls.Add PB
[8]  GR←PB.CreateGraphics
[9]  GR.{DrawLine(Pens.Gold,w)}''+?100 4p300
[10] GR.{DrawLine(Pens.Chocolate,w)}''+?100 4p300
[11] GR.{DrawLine(Pens.BlueViolet,w)}''+?100 4p300
[12] F.Close

```

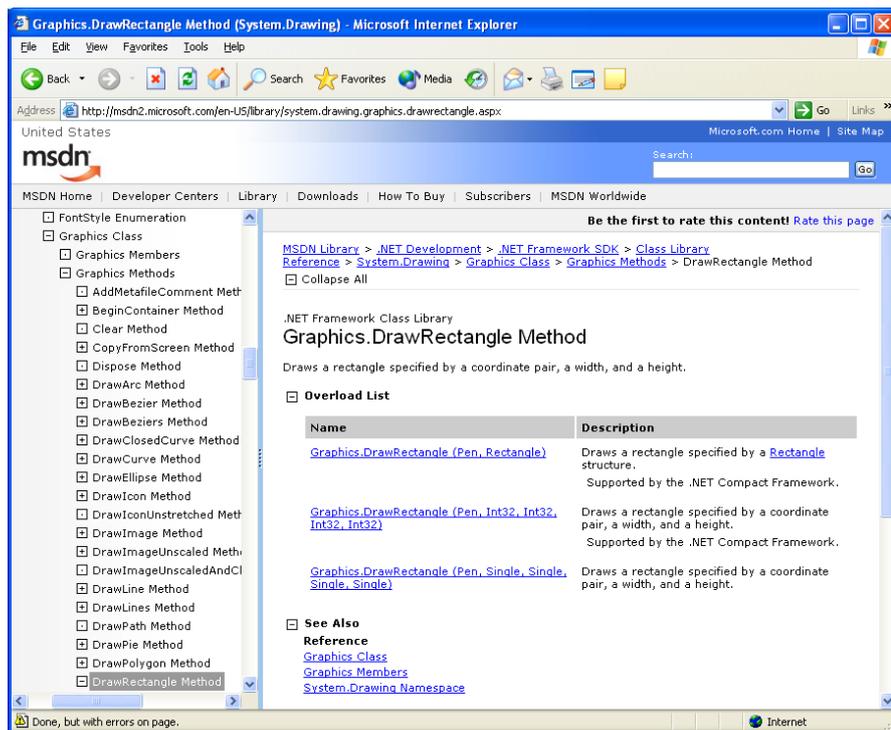
a Scribble lines on a Form.  
 a Initiate System.Windows.Forms and System.Drawing namespaces.  
 a Create inst.of System.Windows.Forms.Form and make it visible.  
 a Create instance of ..Forms.PictureBox and make its Size same as the Form.  
 a Add control to Form \* \*\*\* \*  
 a Create Graphics obj with CreateGraphics method  
 a Run DrawLine method of ..Drawing.Graphics class using Chocolate pen of System.Drawing.Pens class to create a System.Drawing.Pen object.  
 a Close the Form.

What class owns the *Add* method?

17.3.1.3 Trace the `▽_Grid2▽` function in supplied workspace `..\samples\winforms\winforms.dws`.

An *Edit* object becomes a *TextBox* class in Dyalog.Net, a *Grid* object becomes a *DataGrid* class but a *Label* is still called a *Label* and a *StatusBar* is still called a *StatusBar*. ☺

The definitive guide to .NET framework class libraries is MSDN (MicroSoft Dot Net), available on-line or as a download.



17.3.1.4 Assign a simple `▽show▽` function to the `onClicK` property of Button *B*. Trace the line `Application.Run F` and compare with `□DQ`. Notice that the message argument is a 2-vector of *objects*.

17.3.1.5 Trace the .NET-laced `▽RUN▽` function in workspace `..\samples\winforms\gdiplus.dws`, watching out for the instance of the *Timer*  $\phi$ .  
 Hint: Trace `Application.Run Form1` rather than using the Session implicit `□DQ`.

17.3.1.6 Play the pretty game of Tetris in workspace `..\samples\winforms\tetris.dws`, then trace the *exhibition-quality* Dyalog.Net code. Take care with the `onTicK` event and multi-threading when tracing.

## §§ 17.3.2 Communications

Computer communications is a huge topic these days. Once upon a time it might have covered simply the notion of conveying messages to the user, which today we might provide thus:

```
⊞USING←'System.Windows.Forms, System.Windows.Forms.dll'
  MessageBox.Show<'This in itself is the message.'
```



Press OK.

OK

Communications might have included requests for information about the local system environment.

```
⊞USING←'System'
  ⚡IO.Directory.GetCurrentDirectory ↳ 'C:\Dyalog\DWS'
```

which was previously covered by `⊞NA'kernel32|GetCurrentDirectoryA U4 >0T'` or before that by `⊞CMD'cd'`. More such communications with the environment can now be exemplified by

```
  ⚡IO.Directory.(GetParent GetCurrentDirectory) ↳ 'C:\Dyalog'
```

or

```
  IO.Directory.(GetDirectoryRoot GetCurrentDirectory) ↳ 'C:\'
```

or

```
  Environment.CurrentDirectory ↳ 'C:\Dyalog\DWS'
```

as an alternative to .NET above (except it does not return an object and so can be assigned directly), or

```
  Environment.CommandLine
```

"C:\Program Files\Dyalog\Dyalog APL 11.0\dyalog.exe"

which is the same as `#.GetCommandLine` in Dyalog GUI terms. Similarly,

```
  Environment.UserName ↳ 'ADENNY'
```

gives the same result as `⊞AN`.

But there are many other properties and methods in the Framework Class Library that give information that is not readily available in raw APL, although probably accessible via `⊞NA`.

```
  Environment.MachineName ↳ 'JCM5032483'
```

```
  Environment.(OSVersion Version)
```

```
  Microsoft Windows NT 5.1.2600 Service Pack 1 2.0.50215.44
```

```
  Environment.GetLogicalDrives
```

```
  A:\ C:\ D:\ K:\ L:\ M:\ O:\ P:\ Q:\ R:\ U:\ X:\ Y:\ Z:\
```

Sometimes APL gives information not directly available from Microsoft .NET, such as the inverse of a matrix, and sometimes APL just is not very concerned:

```
  †(Int16 Int32 Int64).(MinValue MaxValue)
      -32768          32767
     -2147483648     2147483647
    -9223372036854775808  9223372036854775807
```

Once upon a time extraction of data from a file system or database might have been classed as *communications*. In .NET the *System.Data...* namespaces contain facilities for ODBC, SQL...

But we all know that communications is much bigger and wider than all that. It means radio, TV, postal services and transport. But in particular these days for computing it means *eMail* and the *Internet*.

The .NET Framework has a namespace in the base class library *System.dll* called *System.Net* and another called *System.Net.Sockets*. These cover most of the TCP/IP functionality available through Dyalog *TCPsocket* objects. For example the *System.Net.Sockets.Socket* class has a *Send* method which is similar to the *TCPsend* method in the Dyalog GUI. But the *System.Net* namespace has a lot more functionality. For example, there are classes relating to Authentication, Cookie control and HTTP handling and a namespace *System.Net.Security* relating to security issues. The framework class library also has as a number of other assemblies, such as *System.Web.dll*, entirely devoted to Internet issues and *System.Web.Mail* relating to email issues.

Consider, for example, the *TCPGotAddr* event of *TCPsocket* objects in the Dyalog GUI. This event may be used to report the IP address associated with a host name. Alternatively, the DOS command `C:\WINDOWS\system32\nslookup.exe` may be used to find the same information. This information is retrieved via a Domain Name Server (DNS) located somewhere on the visible network. It is the job of this server to maintain an up-to-date list of site names (domains) and their IP addresses.

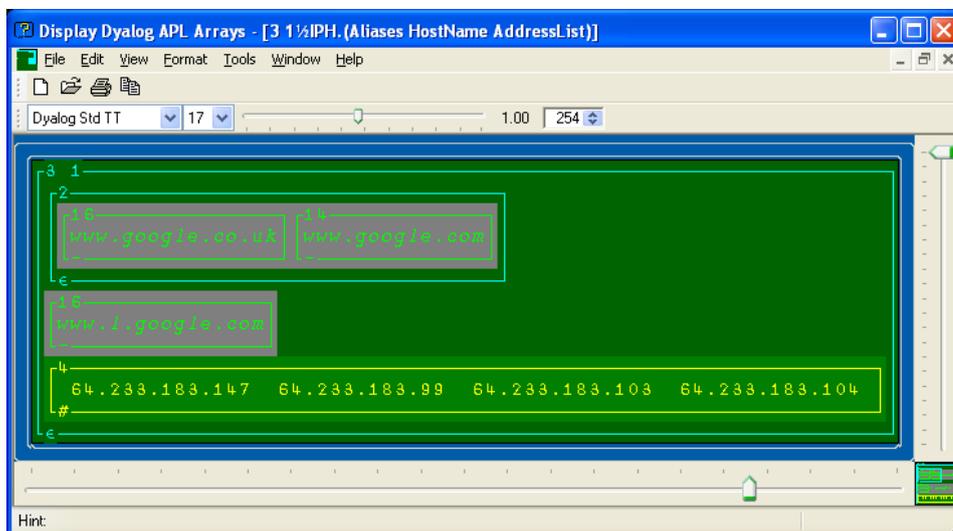
In .NET, the *System.Net.Dns* class has a method called *BeginGetHostByName* that takes a URL parameter and returns asynchronously, having made contact with a DNS server, an object containing information about the URL in question.

```
⊞USING←'System,system.dll'
URL←'www.google.co.uk'
RAR←System.Net.Dns.BeginGetHostByName URL(⊞NS'')(⊞NS'')
```

*RAR* is a namespace of data type *System.Net.Dns+ResolveAsyncResult*. This object is used as an argument to the *System.Net.Dns* class method *EndGetHostByName*. When the URL has been resolved, this method returns an object of data type *System.Net.IPHostEntry*. An instance of *System.Net.IPHostEntry*  $\phi$  has properties *Aliases*, *HostName* and *AddressList* that give some basic information about the domain, including all the related names and addresses.

```
IPH←Net.Dns.EndGetHostByName RAR
3 1pIPH.(Aliases HostName AddressList)
www.google.co.uk www.google.com
www.l.google.com
64.233.183.103 64.233.183.104 64.233.183.147 64.233.183.99
```

*Aliases* is of data type *System.String[]* which in APL translates into *VecCVec*. *HostName* is of data type *System.String* which in APL terms translates into *CVec*, and *AddressList* is of data type *System.Net.IPAddress[]* which returns a vector of instances of the *System.Net.IPAddress* class.



Based on this functionality, Stefano Lanzavecchia has given the [dotnet@dyalog.com](mailto:dotnet@dyalog.com) group, amongst many other treasures, the following function which determines the IP addresses of all 3-letter .com domains.

```

    ▽ r←ss1;step;name;list;n;⊞USING;blocks;b;x;t
[1]  step←300      a blocksize
[2]  ⊞USING←' ' ,system.dll'
[3]  name←{'www.' ,ω ,'.com'}
[4]  list←name`` ,>°.,/⊞A ⊞A ⊞A
[5]  r←list,[1.5]←' '
[6]  blocks←((ρlist)ρ(step+1))←ιρlist
[7]  t←⊞AI[3]
[8]  :For b :In blocks
[9]      ⊞←'n: '(>b)'/'(ρlist)
[10]     ⊞←'elapsed: '(0.001×⊞AI[3]-t)'estimated: '(0.001×(ρlist)×(⊞AI[3]-t)÷>b)
[11]     x←{System.Net.Dns.BeginGetHostByName ω(⊞NS'')(⊞NS'')}list[b]
[12]     r[b;2]←{0:;θ ◊ (System.Net.Dns.EndGetHostByName ω).AddressList.ToString}x
[13]  :EndFor
    ▽
    
```

17.3.2.1 Create a new instance of *MailMessage*  $\phi$  in *System.Web.Mail*  $\tilde{n}$ . Assign suitable values to the *To*, *From*, *Subject* and *Body* properties of the object. Run the *Send* method belonging to *System.Web.Mail.SmtpMail*  $\phi$  with the *MailMessage* object as its argument.

Hint: See the *Dyalog.Net Interface Guide* p20.

Note1: Lines in the *Body* string end in  $\⊞AV[3]$  (LF) and the *Body* is terminated with  $\⊞AV[4]$  (CR).

Note2: You might need to set the name of your SMTP relay mail server via *SmtpMail.SmtpServer*.

17.3.2.2 Use .NET to retrieve the string contents of a URL web site. Run the *Create* method of *System.Net.WebRequest*  $\phi$  from *System.Net*  $\tilde{n}$  in *system.dll*  $\grave{a}$  with an argument of some URL string such as 'http://www.dyalog.com'. The *GetResponse* method returns an object of data type *System.Net.WebResponse*. This object has a method called *GetResponseStream* that returns an object of data type *System.IO.Stream*. This object may then be used as a parameter when creating a new instance of *System.IO.StreamReader*  $\phi$ . Finally, the *ReadToEnd* method of this instance returns a string containing the contents of the URL home page. ☺

Hint: See the *Dyalog.Net Interface Guide* p21.

Note: You might need to create a suitable instance of *WebProxy*  $\phi$  and assign it to the *Proxy* property.

### §§ 17.3.3 Generalising APL Primitives

Many facilities in Dyalog version 9 are replicated in Dyalog.Net. Succession has occurred many times before and in many different contexts. If Microsoft continues its success then .NET is here to stay for the foreseeable future. It will replace the methodologies of the Dyalog GUI, Dyalog TCPSocket objects, APL threads,  $\⊞NA\dots$ , perhaps unnoticeably; like  $\⊞TS$  changed is clock and  $\⊞AN$  changed its data source (ask Geoff Streeter ☺).

Even APL primitive functions may be supplemented with .NET methods or replaced by .NET equivalents although the current mathematical offerings of .NET are far less extensive than those in APL 1. One might hope, for example, that complex arithmetic will be gifted to Dyalog APL through .NET although neither camp seems particularly motivated. However, it is the simple basic *grammar* of APL and not the underlying *algorithms* that distinguishes APL from all other less elegant, less regular languages.

Although APL originated as a notation for succinctly describing algorithms and was only later implemented as a computer language, it owes much to other computer languages in its later incarnations. For example the concepts of file systems, nested arrays, error trapping, control structures, multi-threading and the modern GUI interface are all derived directly by other computer languages.

Error trapping in **Dyalog.Net** follows the OO style. An error encountered within .NET signals an error number 90. This error may be trapped in the usual way with `⎕TRAP` or `:Trap`. `⎕DM` contains the usual diagnostic message, but many more details may potentially be found from the properties (and display form) of the new system object, `⎕EXCEPTION`, which is an instance of `System.Exception`  $\phi$ .

17.3.3.1 Force an error in .NET and examine the properties of the `⎕EXCEPTION` object.

Many people are developing .NET classes to cover various areas of computing which are not found in the framework library, eg <http://www.extremeoptimization.com/> or <http://www.strangelights.com/fsharp/>. Some of these extensions might one day be an intrinsic part of Dyalog APL. Microsoft .NET itself introduces methods which extend basic arithmetic and Boolean functions, to `DateTime` objects for example.

The meaning of adding days to dates or determining whether one date is greater than (after) another are intuitively clear and so .NET introduces methods such as `op_Addition` and `op_GreaterThan` that apply directly to instances of `System.DateTime`  $\phi$  and `System.TimeSpan`  $\phi$ . Dyalog has incorporated some of these 'operators' into the appropriate APL primitive functions.

17.3.3.2 Experiment with APL primitives `+` `-` `=` `≠` `>` `≥` `<` `≤` as applied to `DateTime` and `TimeSpan` objects and compare with corresponding methods in these classes. Dyalog goes further and provides natural extensions to primitives `⋈` `⋉` `⋊` `⋋`. Experiment with derived functions (such as `⋆/`) as applied to dates.  
Hint: See the *Dyalog.Net Interface Guide* p16.

17.3.3.3 Consider joining the [dotnet@dyalog.com](mailto:dotnet@dyalog.com) mailbox group and ask for the next module on **writing .Net classes**.

## Module18: Dyalog.Net Classes

### § 18.1 Writing Dyalog.Net Classes

#### §§ 18.1.1 Dyalog Namespaces and .NET Namespaces

.NET namespaces are similar to Dyalog namespaces, but unfortunately (for .NET) they are not identical. Nevertheless, when creating a .NET class in Dyalog APL, a Dyalog namespace is destined to become a .NET namespace (in a .NET assembly) containing the new .NET class.

18.1.1.1 Create a workspace called GENERAL.DWS containing a single namespace called `#.Maths`.

#### §§ 18.1.2 Creating a *NetType* Object

In Dyalog.Net, a .NET class is created through a *NetType* object. The *BaseClass* property of a *NetType* object may be set to the name of some particular class from which the new class is derived.

```
CVec [WC'NetType'          a Create a new NetType object, with name in CVec
```

18.1.2.1 In the namespace `#.Maths` create a Dyalog GUI *NetType* object called *Spectrum*.  
Hint: See GUI Help or *Object Reference* for description of *NetType* object.

#### §§ 18.1.3 Writing Functions and defining Variables

18.1.3.1 In namespace `#.Maths.Spectrum`, copy in function `∇Fourier∇` from distributed workspace MATH.DWS and write the following two functions:

```
∇ R←ft W a Fourier Transform
[1] R←Fourier W
∇
```

```
∇ R←ift W a Inverse Fourier Transform
[1] R←-1 Fourier W
∇
```

## § 18.2 Exporting Methods and Properties

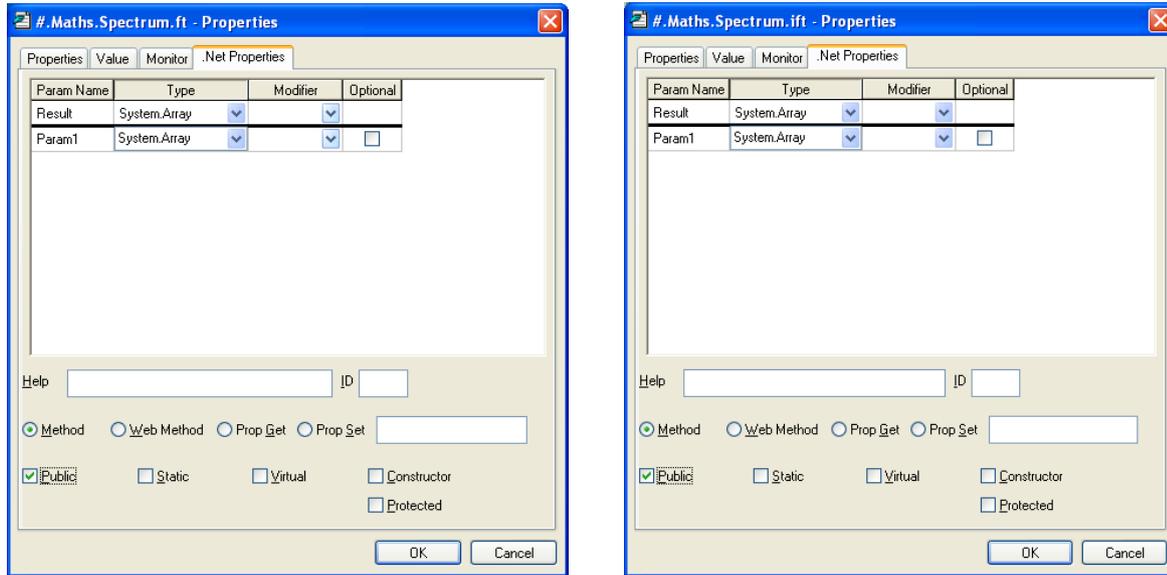
### §§ 18.2.1 Arguments and Result “dataTypes”

When exporting methods and properties it is necessary to specify the dataTypes of all interface variables. In order for .NET to interpret correctly the dataTypes being specified it is necessary for the classes associated with the specified dataTypes to be accessible to APL. The basic dataTypes are to be found in the core *System* namespace. They correspond to classes such as *System.Int32*, *System.Int64* and *System.Array*. These inherit from *System.ValueType* which inherits from *System.Object*.

To make these basic dataTypes visible to APL when exporting class members we might assign `[USING` to the core *System* namespace. Were we to do this then the classes would have to be called *Int32*, *Int64*. But the default names in the .NET properties dialogue are written with the "System." prefix. Therefore we must assign `[USING` to an empty character vector in order to match the names of the default entries below.

We can now specify the calling dataTypes of `∇ft∇` and `∇ift∇` as the default, *System.Array* and identify them as public methods. (Remember, methods may *not* be dyadic functions.)

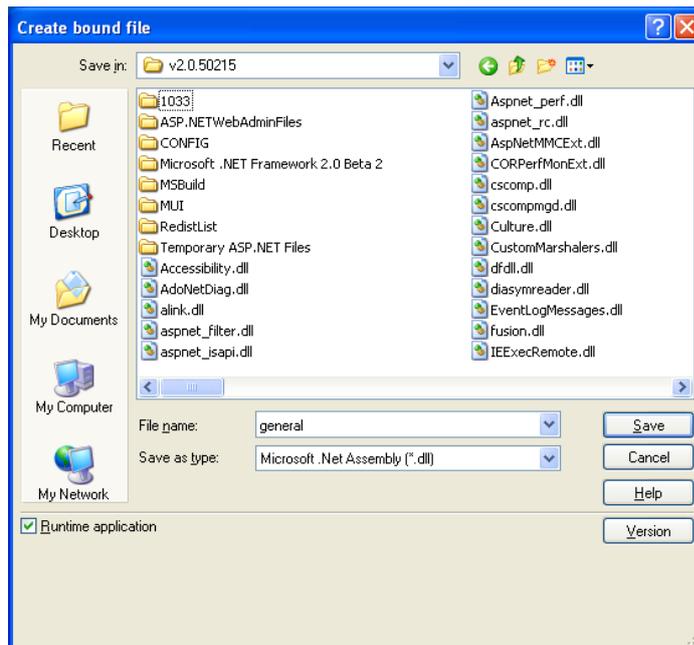
18.2.1.1 Assign `[USING←' ']`, place the cursor on `ft` in the Session and right-click the mouse. Select [Properties][.Net Properties] and set the information as below. Repeat for `ift`.



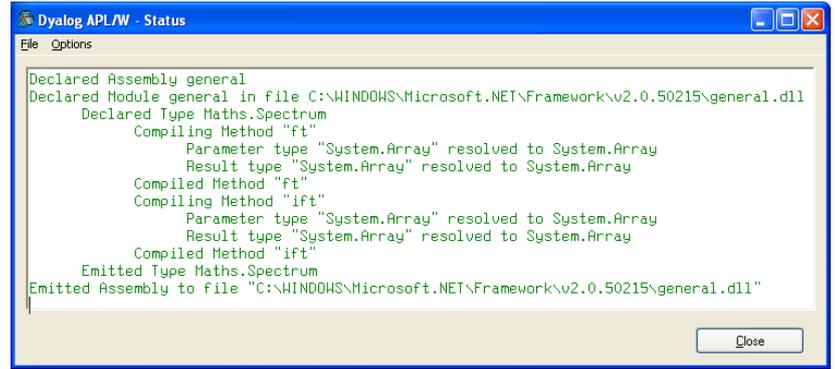
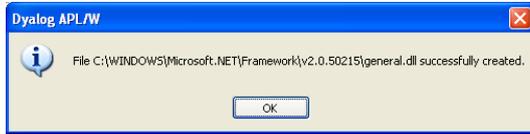
Alternatively, it would be possible to assign this information via `SetMethodInfo`. Also note that we could have identified the methods as `Static` in which case they would be useable directly from the class without the need to create an instance of `Spectrum` before calling them.

### §§ 18.2.2 Making an Assembly

18.2.2.1 Select [File][Export] from the Session menu and navigate to the framework directory. Choose a file name – the default is the name of the workspace with `.dll` rather than `.dws`. This will be the name of your `.NET` assembly. The Runtime application check box should be checked in order to create a distributable assembly, otherwise the development `.dll` (`dyalog110.dll` rather than `dyalog110rt.dll`) will be bound.

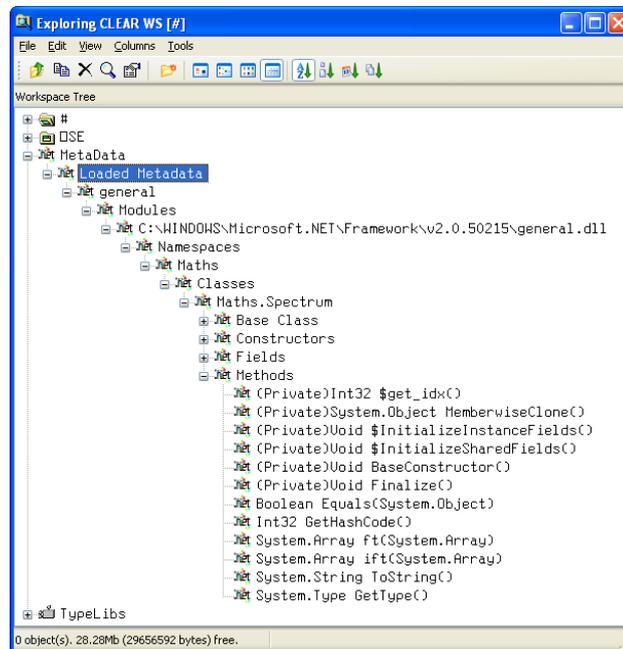


When you hit the Save button, if no problems are identified, then the following message box and status message dialogue appear.



### §§ 18.2.3 Checking the MetaData

18.2.3.1 In WS Explorer, load the MetaData of the assembly *General.dll*. Find the exported methods and check their entries.



Note that a number of other methods are present in the list. They have been inherited from *System.Object* ☿ which is the default *BaseClass* of a *NetType* object. It is at the root of every .NET class and therefore every class has the *ToString* and *GetType* methods (although their constructor syntax may be overridden by any of their descendants).

## § 18.3 Calling Dyalog.Net Classes

### §§ 18.3.1 Calling your Dyalog.Net Class from Dyalog APL

To use the .NET class that you have just created, you follow exactly the same procedure as for any other .NET class. First you have to set `⎕USING` to the appropriate value:

```
⎕USING←'Maths,General.dll'
```

By way of checking that the class is visible to APL, typing its name at this point will print the full namespace-qualified name of the class in the Session, surrounded by parentheses to indicate a class name.

```
Spectrum
(Maths.Spectrum)
```

An instance of the class is created using the `⎕NEW` system function:

```
S←⎕NEW Spectrum
```

Create an arbitrary rank array of data whose Fourier Transform is to be found.

```
⊖←R←2 2 10ρ400?400
```

```
180 95 12 213 350 139 187 273 82 243
150 331 177 34 341 69 371 345 332 28
```

```
391 126 301 61 336 319 31 325 254 36
233 263 43 148 246 153 274 333 305 306
```

Applying the transform followed by its inverse reproduces the original data, now with an explicit zero imaginary component.

```
S.ift←S.ft←R
```

```
180 0 95 0 12 0 213 0 350 0 139 0 187 0 273 0 82 0 243 0
150 0 331 0 177 0 34 0 341 0 69 0 371 0 345 0 332 0 28 0
```

```
391 0 126 0 301 0 61 0 336 0 319 0 31 0 325 0 254 0 36 0
233 0 263 0 43 0 148 0 246 0 153 0 274 0 333 0 305 0 306 0
```

(From Dyalog version 10 onwards it is possible to display ActiveX controls and .NET classes in the cells of a *Grid* object. This, thanks again to John Daintree, introduces considerable scope for new applications.)

### §§ 18.3.2 Calling your Dyalog.Net Class from C# and VB.NET

.NET classes created using Dyalog APL may be called from any .NET language just like any other .NET class. The assembly .dll file, in this case general.dll, and any supporting .dll files, in this case fftw.dll, have to be shipped. The only other Dyalog files that have to be shipped to the machine intending to use the class are bridge110.dll, dyalognet.dll and dyalog110rt.dll, the Dyalog APL runtime engine.

### §§ 18.3.3 Complications

We have attempted to give a straight-forward view of .NET from an APL point of view. Many of the 'new' concepts have been part of Dyalog APL since its inception – data encapsulation, method overloading, exception handling ... - and many have already been incorporated in a natural way into Dyalog APL – namespaces, threads, sockets ...

The reality of .NET is not as clear-cut as an APL programmer would wish. There are inevitably a number of complications. We have neglected at least those we can discern in the belief that the straightforward picture is the best foundation on which to build.

However, we note here a couple of the grey areas that you will encounter on deeper investigation of .NET.

- .NET namespaces may be spread over more than one assembly, as you already know.
- Some C functions do not return their results, but rather they store them in memory at specific locations indicated by pointers, which they *do* return. In Dyalog.Net this eventuality is resolved using *ByRef* ⚡ in *Dyalog* ñ supplied in *bridge110.dll* à.
- Classes are not the only members of assemblies. Assemblies may also contain Enumerations, Interfaces and Structs. We have ignored these topics because they do not appear to introduce anything particularly illuminating, but just complicate for an APLer an already complicated picture.

<sup>18.3.3.1</sup> See *Dylog.Net Interface Guide* chapters 1 to 4 and ..samples\APLClasses\... for further study. Please ask for the next module on ASP.NET 😊.

## Module19: Dyalog.Asp.Net

APL and its applications may yet lead the World through cyberspace! The empowering key is that Dyalog APL programs may be run remotely by, with or from practically any Internet browser on the Planet. There are (almost) no excuses left. If APL is as good as we think it is then it should begin to shine through IE7.

### § 19.1 Dynamic Web Pages

#### §§ 19.1.1 Active Server Pages in VBScript or JScript

The early Internet worked by a browser requesting a page from a server and the server delivering the requested page back to the browser in HTML format. A server is identified by its IP address (or the corresponding domain name translated via a DNS) and a port number (generally port 80).

In 1995 Sun and Netscape incorporated Java into Netscape Navigator. Essentially, Sun added some new tags into HTML that will run Java programs inside a web browser that is *Java-enabled*. Java programs that run in web browsers are called applets. They heralded *dynamic* Internet sites by incorporating **client-side programs**. When you use a Java-enabled browser to view a page that contains an applet, the applet's code is transferred to your system and executed by the browser's *Java Virtual Machine*. Employing local processing like this can enhance an Internet site in many ways, from generating complex drawings, graphics and animations, to getting better control of the mouse, keyboard and available fonts. However, all this requires local facilities that are **not yet enabled** on the vast majority of browsers.

Active Server Pages (ASP) introduced by Microsoft in 1996 added a complementary facility. ASP heralds *dynamic* Internet sites by incorporating **server-side programs**. When a browser requests an ASP, the web server generates, via some COM-enabled language, a page in HTML code, and sends it back to the browser. Dyalog APL can create OLEServer objects and can therefore gain access to this technology. Note that it **does not rely on** any special atypical local facilities on the client side.

<sup>19.1.1.1</sup>Type <http://81.187.162.51:8081> into your web browser or, equivalently, select [Webserver] in [www.dyalog.com](http://www.dyalog.com) to view the Dyalog.Asp example outlined below. The Dyalog web site on IP address 82.111.24.53, port 8081, if accessible from your (possibly prohibitively well protected) web location, demonstrates the Dyalog APL ASP server, called ASPSVR.

If not accessible by you from the remote Dyalog web site, the Dyalog APL ASPSVR may alternatively be installed and run locally on your computer.

<sup>19.1.1.2</sup>Download ASPSVR.ZIP from the [Download Zone][Document Download Zone][ASPSVR] section of [www.dyalog.com](http://www.dyalog.com) and follow the instructions for a good introduction to "classic" ASP with Dyalog APL version 9. A summary outline of the mechanism employed follows below.

The default HTML web page on this site is called **default.htm**. The web page involves frames and therefore it automatically requests a couple more sub-pages including **toc.htm** as highlighted below.

#### **default.htm**

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>...
<frameset cols="30%,70%">
..<frame name="toc" marginwidth="1" marginheight="1" src="toc.htm" target="main" />...
</html>
```

The DOCTYPE line is included so that web authors can validate their HTML documents. There are many variations. None is necessary, but one recommended choice favours HTML 4.01 validation:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 TRANSITIONAL//EN"  
"http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
```

The table of contents file, `toc.htm`, contains a link to an ASP file called **dyalog.asp**, as does another sub file in the web site, **loan.htm**. These two files call `dyalog.asp` in different ways. Both approaches ultimately lead to the main goal; execution of a program in Dyalog APL.

### toc.htm

```
..<td><a href="loan.htm"  
..<td><a href="dyalog.asp?DWSAction=driller.RUN" target="contents">Sage Driller</a></td>  
..<td><a href="dyalog.asp?DWSAction=rain.Fourier&format=PNG" target="contents">Fourier...
```

### loan.htm

```
..<form action="dyalog.asp" method="GET" name="LoanForm"  
..<td><input type="text" size="9" name="LoanAmt"></td>  
..<td><input type="text" size="6" maxlength="6" name="PercentDown"></td>  
..<p><input type="submit" value="Calculate Repayments"></p>  
..<input type="hidden" name="DWSAction" value="loan.RUN">...
```

**Dyalog.asp** being called here is an *active server page* file written in VBScript. The script creates an instance of an OLEServer called **dyalog.ASPSVR** that owns an exported method called **MakeHTML**. This function is passed an argument consisting of some string following a query (?), *eg*

**DWSAction=rain.Fourier&format=PNG** in `toc.htm` above. **MakeHTML** runs in Dyalog APL and may execute any APL code as long as its final result is HTML. However, VBScript is generally only supported by the Microsoft browser IE and the alternative language, JScript, is not enabled by default.

### dyalog.asp

```
<% @Language = "VBScript" %>  
.. Set aplsvr = Server.CreateObject("dyalog.ASPSVR")  
.. strQuery = Request.QueryString  
..Response.Write aplsvr.MakeHTML (strQuery)...
```

The Dyalog OLEServer **dyalog.ASPSVR** is created from workspace **aspsvr.dws**. [File][Export] generates **aspsvr.dll**, which must be registered on the server using **regsvr32.exe**. When used, **aspsvr.dll** is run in conjunction with **C:\WINDOWS\system32\dyalog.dll**. Within the **aspsvr.dws** workspace, the function **#.ASPSVR.MakeHTML** is exported as a method. The essence of this function is:

```
▽HTML←MakeHTML PARS;NS;FN;DWSPARS;INST;IO;ML  
.. DWSPARS PARS←SplitPars DeCode'''&='Split PARS  
.. HTML←FN, ' ↑PARS'  
...▽
```

Once registered, the OLEServer may be tested in Dyalog.

```
'ASV' WC'OLEServer' 'dyalog.ASPSVR'  
ρASV.MakeHTML'DWSAction=loan.RUN'↳3759  
ρASV.MakePicture'DWSAction=rain.Timeseries&format=PNG'↳204
```

In the `Dyalog.asp` example above, this OLEServer is used to drive a web server that can execute Dyalog APL code remotely. This scenario is reminiscent of mainframe timesharing for 1<sup>st</sup> generation APLs! 😊

## §§ 19.1.2 The *System.Web.UI.Page* Class

In 2002 "classic" ASP was superseded by ASP.NET. ASP.NET is part of Microsoft .NET and therefore has full .NET support for languages such as VisualBasic.NET, C#, .. and [Dyalog APL](#).

In Windows, an Internet site is hosted by IIS, the Internet Information Service. IIS must be installed from the Windows CD *before* .NET is installed. And .NET must be intalled *before* Dyalog APL and the Dyalog APL 11.0 .Net Interface Components. **Note that at least in beta versions of Dyalog APL 11.0, IIS 6.0 in combination with .NET 2.0 is not supported, and ASP.NET may only be used with Dyalog APL when using .NET 1.1 plus SP1, as matches the default version specified in [dyalog.exe.config](#).**

An ASP.NET page is identified by the extension **.aspx** (as opposed to **.asp** for classic ASP). All executable code is moved out of the <html> section and into a new <script> section of the file.

As far as the programmer is concerned, the dynamic part of ASP.NET pages is built with graphical controls in a way similar to a standard Windows user interface, and the program dynamics is event-driven like all Windows GUI applications. For example, a web button is assigned properties and responds to events. However, rather than being immediately drawn, web controls in segments of html-plus-script are built on the server and form part of the resulting page sent to the end-user's browser.

On receipt of a request for a .aspx page, the ASP.NET engine within IIS automatically creates a class that derives from the System.Web.UI.Page class. This dynamically created class is immediately compiled and run to produce html which is returned to the client.

Multi-user access is managed by IIS. In particular, IIS maintains one distinct AppDomain for each ASP.NET application currently running.

**19.1.2.1** Copy the following code into Notepad and save the file as **C:\Inetpub\wwwroot\VB1.aspx**, then type <http://localhost/vb1.aspx> in IE. A button asking to be clicked should appear in IE.

### **C:\Inetpub\wwwroot\VB1.aspx**

```
<html>
<body>
  <form runat="server">
    <asp:Button id="button1" Text="Click me!" runat="server" />
  </form>
</body>
</html>
```

In ASP.NET, all HTML server controls must be within a single <form> tag with the **runat** attribute set to "server". Note that there can only be one <**form** runat="server"> control per .aspx page. The runat="server" attribute indicates that the control is to be processed on the server. It also implies that controls enclosed in <asp: .. > may be accessed by server scripts.

**19.1.2.2** Add the VB script section in **pink** to vb1.aspx. Note the addition of the OnClick event that initiates the VB **submit** callback when the button is pressed. Save this as **C:\Inetpub\wwwroot\VB1.aspx** and press the button from <http://localhost/vb1.aspx>. Note that all executable code resides *outside* the <html> tags.

### **C:\Inetpub\wwwroot\VB1.aspx**

```
<script Language="VB" runat="server">
  Sub submit(Source As Object, e As EventArgs)
    button1.Text="You clicked me!"
  End Sub
```

```
</script>
<html>
<body>
  <form runat="server">
    <asp:Button id="button1" Text="Click me!" runat="server" OnClick="submit"/>
  </form>
</body>
</html>
```

Within the `submit` function all sorts of other VB code could be added, eg

```
button1.Style("background-color")="#0000ff"
button1.Style("color")="#ffffff"
button1.Style("width")="200px"
button1.Style("cursor")="hand"
button1.Style("font-family")="verdana"
button1.Style("font-weight")="bold"
button1.Style("font-size")="14pt"
button1.Text="A New Caption"
```

A second example of the `OnClick` event, this time involving a `TextBox` control, is shown in `VB2.aspx` below. You will convert this example from VB, the default language, to Dyalog in exercise 19.2.1.1.

#### C:\Inetpub\wwwroot\VB2.aspx

```
<script runat="server">
Sub submit(sender As Object, e As EventArgs)
lab1.Text="Your name is " & txt1.Text
End Sub
</script>
<html>
<body>
<form runat="server">
Enter your name:
<asp:TextBox id="txt1" runat="server" />
<asp:Button OnClick="submit" Text="Submit" runat="server" />
<p><asp:Label id="lab1" runat="server" /></p>
</form>
</body>
</html>
```

When a browser makes a request for an ASP.NET web page, the request is first sent to the server implicated by the URL. If it is a Windows server, IIS receives the request, recognises the `.aspx` extension, and passes the request on to ASP.NET for processing. ASP.NET creates an instance of the `System.Web.UI.Page` class from the `.aspx` file contents.

When a page is created the `Load` event is triggered. By default, ASP.NET tries to find the special method name `Page_Load` on the page. If a match is found, the function is considered to be a handler for the `Load` event. In other words, `Page_Load` is taken to be the callback attached to the `Load` event.

19.1.2.3 Add the following file to your default web site and run it in IE. Notice how the time changes on refresh.

#### C:\Inetpub\wwwroot\VB3.aspx

```
<script runat="server">
Sub Page_Load
lab1.Text="The date and time is " & now()
End Sub
```

```
</script>
<html>
<body>
<form runat="server">
<h3><asp:label id="lab1" runat="server" /></h3>
</form>
</body>
</html>
```

The Page\_Load subroutine runs every time the page is loaded. If you want to execute the code in the Page\_Load subroutine only the first time the page is loaded, you can use the IsPostBack property of a Page object – ie an instance of the Page class. If the IsPostBack property is false (0), then the page is being loaded for the first time. If IsPostBack is true (1), the page is being posted back again to the server.

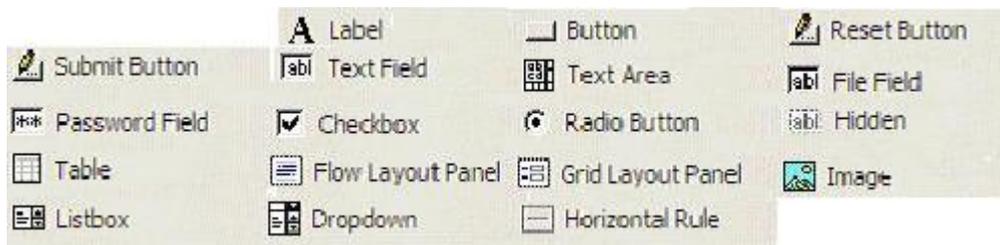
19.1.2.4 Incorporate the 'If' condition into **VB3.aspx** and notice the different response on [View][Refresh].

```
<script runat="server">
Sub Page_Load
If Not Page.IsPostBack then
lab1.Text="The date and time is " & now()
End If
End Sub
```

§§ 19.1.3 The *System.Web.UI.WebControls* Namespace

There are two groups of controls available to the web programmer. There are the standard ones used to present web pages, including dynamic ones for client-side scripts. And there is a new set of dynamic controls for ASP.NET server-side scripts.

Under .NET, the first group is located in the **System.Web.UI.HtmlControls** namespace. These map directly to standard HTML tags supported by all browsers. They allow simple programmatic control of HTML elements on any HTML or ASP.NET page. The second group is ASP.NET specific and is found in the **System.Web.UI.WebControls** namespace. Here is a list of some of them. They are distinguished by the fact that they may be used to *initiate a program on the server*.



Any of these web controls may be included in the .aspx <form> tag like this:

```
<asp:HyperLink id="HyperLink1" runat="server">HyperLink</asp:HyperLink>
<asp:RadioButtonList id="RadioButtonList1" runat="server"></asp:RadioButtonList>
<asp:DropDownList id="DropDownList1" runat="server"></asp:DropDownList>
<asp:ListBox id="ListBox1" runat="server"></asp:ListBox>
<asp:Image id="Image1" runat="server"></asp:Image>
<asp:AdRotator id="AdRotator1" runat="server"></asp:AdRotator>
<asp:Table id="Table1" runat="server"></asp:Table>
<asp:Calendar id="Calendar1" runat="server"></asp:Calendar>
<asp:DataGrid id="DataGrid1" runat="server"></asp:DataGrid>
```

Usually callbacks are set on appropriate events on these controls. For example, a form is most often submitted by clicking on a button.

```
<asp:Button id="id" text="label" OnClick="submit" runat="server" />
```

There is also a set of controls whose job it is to validate entry into certain web controls. eg

```
<asp:RequiredFieldValidator id="RequiredFieldValidator1" runat="server"
    ErrorMessage="RequiredFieldValidator"></asp:RequiredFieldValidator>
<asp:CustomValidator id="CustomValidator1" runat="server"
    ErrorMessage="CustomValidator"></asp:CustomValidator>
<asp:ValidationSummary id="ValidationSummary1" runat="server"></asp:ValidationSummary>
```

For example, to restrict the contents of a TextBox called `tbox1` to integers between 1 and 100, a RangeValidator may be set to:

```
<asp:RangeValidator ControlToValidate="tbox1" MinimumValue="1" MaximumValue="100"
    Type="Integer" EnableClientScript="false" Text="The value must be from 1 to 100!"
    runat="server" />
```

A more complete list of controls can be found at the official ASP.NET web site, <http://www.asp.net>.

## § 19.2 Dyalog Script Language

### §§ 19.2.1 Callbacks in Dyalog APL

The script in file `C:\Inetpub\wwwroot\VB1.aspx` is written in the default scripting language, VB .NET. In order to rewrite it in Dyalog APL it is necessary to set the Language attribute to **"dyalog"** in Dyalog version 11 (or "apl" in version 10). Then all that needs to be done is to convert the Visual Basic .NET code into **Dyalog APL** code ☺.

#### C:\Inetpub\wwwroot\APL1.aspx

```
<script language="dyalog" runat="server">
    vsubmit args
    :Access Public
    :Signature submit Object Source, EventArgs e
    button1.Text←'You clicked me!'
    ▽
</script>
<html><body>
    <form runat="server">
        <asp:Button id="button1" Text="Click me!" runat="server" OnClick="submit"/>
    </form>
</body></html>
```

The `:Access Public` statement means the function may be called from outside the script.

The `:Signature ..` statement is the equivalent of `Source As Object, e As EventArgs` and defines the dataTypes of the standard incoming event message arguments. These arguments are not actually used in this particular APL code, although the third line could have been coded as

```
(>args).Text←'You clicked me!'
```

#### 19.2.1.1 Convert C:\Inetpub\wwwroot\VB2.aspx to C:\Inetpub\wwwroot\APL2.aspx and test it in IE.

Hint: See [Control Panel][Regional and Language...][Languages][Details][Settings] and the *Dyalog.Net* Manual Chapter 10 for scripting APL in Notepad.

Note that `USING` may be assigned inside the `<script>` tags, indicating that the full power of the .NET framework, as well as the full power of Dyalog APL, may potentially be summonsed from any browser.

## §§ 19.2.2 Workspace behind ...

19.2.2.1 Start the Dyalog.Net tutorial at [www.dyalog.com](http://www.dyalog.com) by selecting [Products][Dyalog for Windows][Microsoft .NET Interface][Web Pages Tutorial] or by typing <http://81.187.162.51/tutorial.net> directly into your browser. Run the examples and view the explanation of each.

In the example `..\tutorial\intro6.aspx`, the entire `<script>` section is replaced with a reference to a workspace, `fruit.dws`, which contains a single namespace called `FruitSelection`.

19.2.2.2 Copy the file `..\Samples\asp.net\tutorial\intro6.aspx` to `C:\Inetpub\wwwroot\intro6a.aspx` and change the name of the workspace being called to `C:\Inetpub\wwwroot\fruity.dws`

### C:\Inetpub\wwwroot\intro6a.aspx

```
<%@Page Language="Dyalog"
    Inherits="FruitSelection"
    src="fruity.dws" %>
<html>
<h1>intro6: Workspace Behind</h1>
<p>This example illustrates how you can use an APL workspace.</p>
<body>
    <form runat="server" >
        <asp:DropDownList
            id="list"
            runat="server"
            autopostback="true"
            OnSelectedIndexChanged="Select"/>
        <p>
            <asp:Label
                id=out
                runat="server" />
        </p>
    </form>
</body>
</html>
```

The only function explicitly called from the workspace is the callback, `Select`, on the `DropDownList`.

19.2.2.3 Create a new workspace called `C:\Inetpub\wwwroot\fruity.dws` and within it create the following `NetType` object and functions (exported as methods), then navigate to <http://localhost/intro6a.aspx> .

```
)WSID C:\Inetpub\wwwroot\fruity.dws
was CLEAR WS
[using←' 'System.Web.UI,System.Web.dll'
    'FruitSelection'WC'NetType'('BaseClass' 'Page')
]cs FruitSelection
#.FruitSelection
```

The only function explicitly called from the workspace is the callback, `Select`, which we define as:

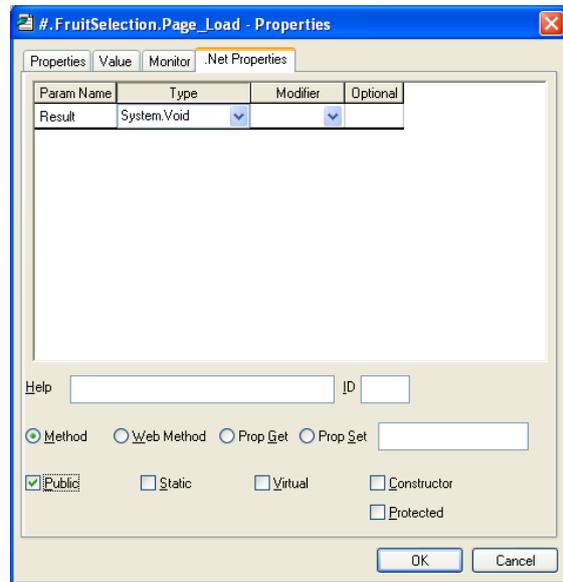
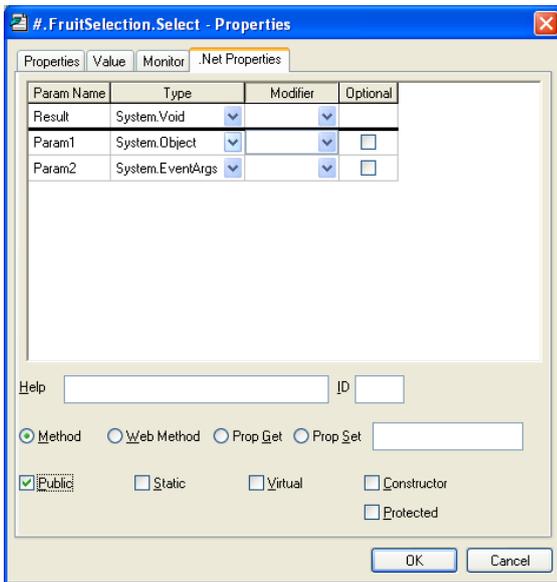
```
▽ Select args
    out.Text←'You selected ',list.SelectedItem.Text
▽
```

The contents of the DropDownList control is initially empty. We can use the `Page_Load` callback to initialise the control contents.

```

▼ Page_Load
:If 0=IsPostBack
    list.Items.Add<'Raspberry'
    list.Items.Add<'Blackberry'
    list.Items.Add<'Grape'
    list.Items.Add<'Mango'
:EndIf
▼
    
```

These two methods must be exported as Public methods and their calling structure must be set appropriately in the [Properties][.Net Properties] popup boxes invoked by right-clicking on the current (`CurObj`) function.



As shown in the online tutorial, in place of a workspace the APL code may be saved as a class in a *script file* with extension `.apl`. This is more consistent with standard language methodology, called *code behind* rather than *workspace behind*, but it loses the most advantages of working with APL workspaces.

(Alternatively, a *primitive APL class* called `FruitSelection` may replace the `NetType` object. This approach is more consistent with the latest standard language methodology and is pursued in §20.)

19.2.2.4 Convert `..\tutorial\into1.aspx` to use a "workspace behind" rather than a scripted function.

### §§ 19.2.3 The TextBox Control

We are now in a position to implement a very basic APL session hosted inside IE. The session window might be represented by a TextBox control whose `TextMode` attribute is set to "multiline".

```

.. <body style="font: 10pt verdana">
  <form runat="server">
    <h3>Dyalog ASCII</h3>
    ..<asp:TextBox id="txt1" textmode="multiline"
      runat="server" rows="20" cols="50"
      acceptsReturn="1"></asp:TextBox>...
  </form></body>...
    
```

A line typed into this TextBox may be executed on the server in **Dyalog APL**, using a button to initiate execution. Note: until Unicode is standard, no APL font can be assumed to exist on a typical client PC.

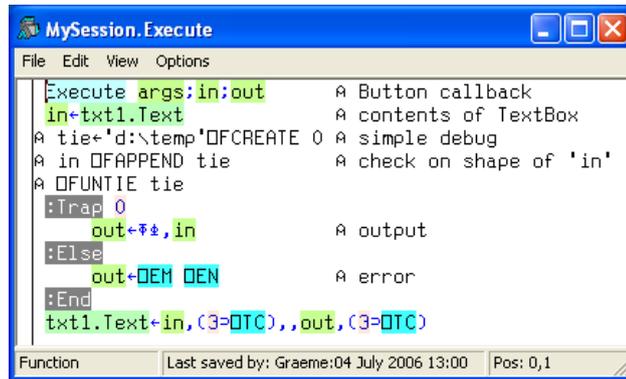
```
..<p>
<asp:Button id="btn1" Text="Execute" runat="server"
    NotifyDefault="1" onclick="Execute"/>
</p>...
```

The <script> section is replaced by a file with an opening line such as:

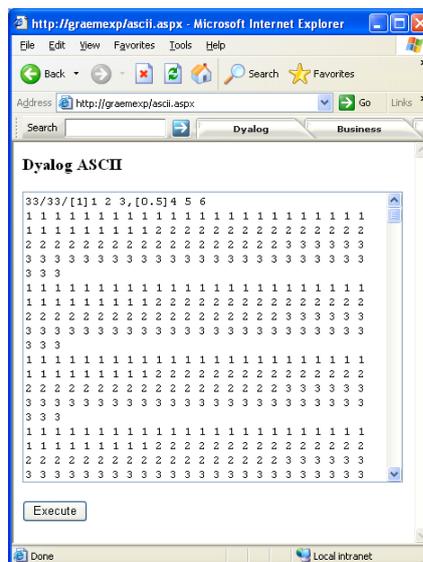
```
<%@Page Language="dyalog" Inherits="MySession" src="ascii.dws" %>
```

The workspace **ascii.dws** contains a *NetType* object called *MySession* whose *BaseClass* is the System.Web.UI.Page class. *MySession* contains just one function, *Execute* that is exported with properties similar to *#.FruitSelection.Select* above.

Inside the APL *Execute* function, the text in the TextBox is extracted from *txt1.Text*. The line of text is executed and the result (or **DM** in the event of an error) is inserted back into *txt1.Text*.



19.2.3.1 Write a page called **..\ascii.aspx** which invokes a "workspace behind" called **ascii.dws** that executes lines of ASCII text as if they were in some APL font.



Despite obvious font restrictions, **ascii** still allows parentheses, brackets, quotes and some primitives such as + - \* ! ? | , ~ < = > ^ \ / & . and # to be typed in directly.

Other APL primitives would be usable (with different symbols such as  $\frac{1}{2}$  for  $\rho$  or  $\frac{1}{4}$  for  $\iota$  or „ for  $\leftarrow$ ), if only they could be typed or pasted into a TextBox in IE. What *can* be typed into a typical Windows application depends on the Windows Language in operation. In XP, this is found in [Control Panel][Regional and Language ...][Languages][Details][Settings]. Notice the **Dyalog APL New Keyboard** is installed here. It enables you to write APL scripts in Notepad. Note, however, that the *Input Method Editor* is only present on a computer that has Dyalog installed and is unavailable in a general web setting. The availability of different scripts in IE is constantly improving, as can be seen from Microsoft web site:

<http://www.microsoft.com/windows/ie/ie6/downloads/recommended/ime/> . Perhaps, in Dyalog APL 12 under Vista, Unicode will be fully supported for input into IE in any language, but keyboard limitations are likely to be a constraint until they move from 8 bit to 16 bit devices. (Even applications such as Microsoft Word that store text in Unicode generally rely on 8-bit keyboards.)

Recipients of mail to the `dyalogusers` group on Yahoo and readers of **Vector** (the Journal of the British APL Association - see <http://www.Vector.org.uk>) will know that Stephen Taylor has developed an APL ‘sandbox’ along the lines outlined above, accessible through the Internet.

## § 19.3 Remote Applications

The future of most APLer’s will be programming APL applications for the Internet platform. Dyalog APL can already support this in a number of ways.

We first saw this achieved using *TCPSocket* objects in §§15.2.3 in the workspace **APLSERVE.DWS**. The main hurdle there was interpretation of the Hypertext Transfer Protocol that surrounds browser packets (as most recently defined in RFC2616 for HTTP/1.1, dated June 1999). This is what IIS handles and is therefore not necessarily a hurdle for an APL Windows web server. However, we still have to converse with browsers, which basically talk in Hypertext Markup Language (HTML, including scripting).

Then we executed APL remotely through **dyalog.ASPSRV**, an OLEServer control that we called through a classic ASP page in a file called `Dyalog.asp` written in VB Script. But the OLEServer has to be installed and registered on each local computer, and VB Script has to be enabled on the browser.

Now we are running APL and **ASP.NET** through IIS. This assumes very little about the client browser. The ability to support *zero-footprint clients* would seem to be the main road to ‘everywhere’ ☺.

All specifically APL web site considerations have been forced onto the server side. Browsers may therefore be viewed as extensions of APL-supported hardware - like keyboards, screens or printers - and the business of writing APL programs on the Internet platform now can begin without client-side impediments.

The primary thrust of .NET would seem to be **ASP.NET**, **ADO.NET** and the facilitation of **web services**. Microsoft has been aiming at the Internet since 1996, first through its abstract DNA (Distributed interNet Architecture) methodology - a way to think about writing applications – and then through ASP whereby a set of technologies implementing a DNA solution are glued together and distributed over the web.

The novelties in .NET seem to mostly concern the Internet. *System.Web* is the primary new functionality in .NET. The first Microsoft Internet site was born in early 1993 and Microsoft launched its public Internet Web domain with a home page in 1994. In 1995 Bill Gates commented, "Amazingly, it is easier to find information on the Web than it is to find [the] information on the Microsoft Corporate Network!" And in the same year Microsoft Internet Explorer 1.0 barged into Windows. As has been said of the evolution of mankind, Bill might say, "We got here as soon as we could!"

## §§ 19.3.1 The C:\Inetpub\wwwroot\ Directory

Who is going to serve your Active APL Host (Aah!)? In order to follow the route outlined above, you will need an ISP that supports the .NET framework, and ASP.NET under IIS (*ie* Windows), and runs [Dyalog APL](#). Each requirement reduces the choice of providers and increases the cost of the web site. The best solution is to **host you own site** on a dedicated box in the cellar, suitably isolated, and protected.

19.3.1.1 Obtain a static IP address that can be pinged from outside world. Give your IP address an available name, such as **apl4.net**, and register the name with a Domain Name Server. But, for limited use on a private intranet you may instead use your computer's *full name* as given in [Control Panel][System].

19.3.1.2 Deploy your web site by copying files such as **index.aspx** into directory **C:\Inetpub\wwwroot\**. Try to access your site from the outside world by typing <http://apl4.net>. Once you are able to run APL like this then it is time to prepare your welcome to *visitors from outer-cyberspace*.

In the mid '80s there was a debate within I.P.Sharp Associates (IPSA) of Canada as to which was the more important; the Sharp Communications Network which encircled the World and carried electronic mail and data, or Sharp APL, the leading APL language of the time. APL timesharing collapsed with the appearance of PCs and STSC APL\*PLUS/PC. Sharp APL/PC proved too slow to be useful and Reuters bought out IPSA in 1987. Not surprisingly, Reuters primarily desired the international network, not Sharp APL.

In fact, one complements the other. A communication system without storage of state information, like the old telephone system, is of great but limited use. And an undistributed computer language has great but limited value. The early manifestation of the Internet was like the old telephone network. Information was passed around but little if any input was saved and processed. The move now is towards a *stateful* Internet that remembers the prior state between messages.

If you look at [View][Source] in IE for a **.aspx** page you will find that **<asp: ..>** controls are converted to hidden standard HTML controls when the page is sent. These hidden controls store information about the state of the ASP.NET controls. For example, you might find source HTML like:

```
<form name="_ctl0" method="post" action="page.aspx" id="_ctl0">
  <input type="hidden" name="__VIEWSTATE"
    value="dDwtNTI0ODU5MDE1Ozs+ZBCF2ryjMpeVgUrY2eTj79HNI4Q=" />...
```

The **\_\_VIEWSTATE** field holds the detailed status of the page sent by the server. The status is defined through this hidden field placed by the server on each page which has a **<form runat="server">** control.

(A separate attempt to save basic state data is by way of local *cookies*. Therefore note that while developing a web page it is sometimes necessary to delete all saved cookies in IE by way of [Tools][Internet Options...][Delete Cookies...] in order to get IE to respect your program changes.)

ASP.NET adds another mechanism for storing state information. There is a .NET session class called *System.Web.SessionState.HttpSessionState* ☐. It allows storage between transactions. The excellent tutorial in <http://localhost/dyalog.net/tutorial/> shows examples of the *Session* object.

Another important leg of Microsoft's .NET march onto the web is **ADO.NET**. This is all about how to store and retrieve larger volumes of information. But once inside APL, storage of data is not a problem. Either native or APL component files are suitable for voluminous permanent records, although many other database systems can be handled through APL *eg* via **SQAPL**. The parent namespace of the current *AppDomain* (##) may also be used to store temporary individual user information...

## §§ 19.3.2 The *System.Drawing* Namespace

19.3.2.1 Convert the `▽scribble▽` function in §17.3.1 into a drawing on your web site.

Sierpinski's gasket is named after Polish mathematician Waclaw Sierpinski (1882-1969). His gasket, or fractal triangle, is constructed by taking an equilateral triangle, dividing it into four smaller equilateral triangles, removing the centre triangle and repeating the process with each of the smaller triangles.

An algorithmic version for creating an approximation to Sierpinski's gasket goes something like this:

1. Create a triangle, labelling each point of the triangle as P1, P2, and P3.
2. Pick a point within the triangle - call it CurrentPoint.
3. Randomly choose a number between 1 and 3.
4. If the value is 1, move CurrentPoint to the mid-point of the line between CurrentPoint and P1.
5. If the value is 2, move CurrentPoint to the mid-point of the line between CurrentPoint and P2.
6. If the value is 3, move CurrentPoint to the mid-point of the line between CurrentPoint and P3.
7. Draw a pixel at the new CurrentPoint.
8. Return to Step 3 (more returns give a sharper the image).

This algorithm is implemented in C# script. It may be run under IIS from IE to produce the picture below.

### C:\Inetpub\wwwroot\serp.aspx

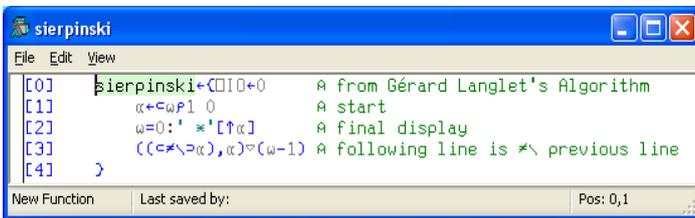
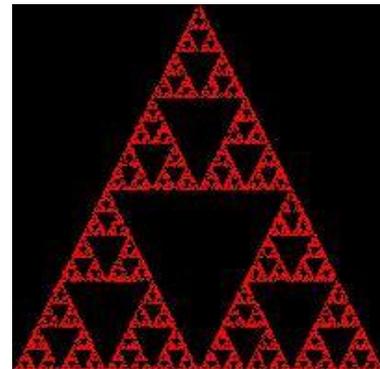
```
<%@ Page Language="c#" %>
<%@ import Namespace="System.Drawing" %>
<%@ import Namespace="System.Drawing.Imaging" %>
<script runat="server">
void Sierpinski(int width, int height, int iterations)
{ // create the Bitmap
  Bitmap bitmap = new Bitmap(width, height);
  // Create our triangle's three Points
  Point top = new Point(width / 2, 0),
    bottomLeft = new Point(0,height),
    bottomRight = new Point(width, height);
  // Now, choose our starting point
  Point current = new Point(width / 2, height / 2);
  // Iterate iterations times
  Random rnd = new Random();
  for (int iLoop = 0; iLoop < iterations; iLoop++)
  { // draw the pixel
    bitmap.SetPixel(current.X, current.Y, Color.Red);
    // Choose our next pixel
    switch (rnd.Next(3))
    { case 0:
      current.X -= (current.X - top.X) / 2;
      current.Y -= (current.Y - top.Y) / 2;
      break;
      case 1:
      current.X -= (current.X - bottomLeft.X) / 2;
      current.Y -= (current.Y - bottomLeft.Y) / 2;
      break;
      case 2:
      current.X -= (current.X - bottomRight.X) / 2;
      current.Y -= (current.Y - bottomRight.Y) / 2;
      break;
    }
  }
}
// Save the image to the OutputStream
```

```

        Response.ContentType = "image/jpeg";
        bitmap.Save(Response.OutputStream, ImageFormat.Jpeg);
        // clean up...
        bitmap.Dispose();
    }
    void Page_Load(Object sender, EventArgs e)
    {
        Sierpinski(200,200, 10000);
    }
</script>
<html>
<head></head>
<body></body>
</html>
    
```

### 19.3.2.2 Rewrite `serp.aspx` in 'Dyalog.Asp.Net'.

Hint: Consider incorporating the DFn below, beautifully crafted by Nicolas Delcros and Gérard Langlet.

## §§ 19.3.3 The *System.Web.Services* Namespace

A `.aspx` file is a prescription for a *Web Page* via classes in the `System.Web.UI` namespace. The client-server communication can then be considered at the HTML level. A *Web Page* is a class that expresses its functionality (properties/methods/events) through a standard web browser.

A `.asmx` file is a prescription for a *Web Service* via classes found in the `System.Web.Services` namespace. The client-server communication can be considered at the XML level (rather than the deeper HTTP level). A *Web Service* is a class that exposes its functionality (properties/methods/events) over the Internet. IE can give a basic rendering of such a service, but generally the client is expected to cater for the service through an explicit local client interface application.

For both *Web Pages* and *Web Services*, Dyalog APL code is controlled and run by the ASP.NET engine inside Microsoft IIS.

More generally, web services form the foundation of Microsoft's *interoperability* efforts. Thus Windows Vista supports XML level interaction with web-service-enabled devices, such as printers, digital cameras, and home control systems.

A `.asmx` file defines a class. It looks rather like a `.apl` file which defines a namespace in script form, except that it begins with a line looking like the opening line of a `.aspx` file. In the case of `.asmx`, this opening statement in the script file declares the language and the name of the service. For example, the following statement declares a Dyalog APL *Web Service* named `GolfService`.

```
<%@ WebService Language="Dyalog" Class="GolfService" %>
```

Details of this excellent example may be found in the *Dyalog.Net Interface* manual, chapters 6 and 7, and working code may be called from `C:\Program Files\Dyalog\Dyalog APL 11.0\Samples\asp.net\golf\`.

The following very simple example of a web service is to be found amongst the many good samples distributed with Dyalog APL.

**C:\Program Files\Dyalog\Dyalog APL 11.0\Samples\asp.net\webservicess\eg1.asmx**

```
<%@ WebService Language="Dyalog" Class="APLExample" %>

:Class APLExample: System.Web.Services.WebService
:Using System
:Access public
  ▽ R←Add arg
  :Access WebMethod
  :Signature Int32←Add Int32 arg1,Int32 arg2
  R←+/arg
  ▽
:EndClass
```

The precise interpretation of this script is deferred until Module 20.

19.3.3.1 Call this service in your browser by typing <http://localhost/dyalog.net/eg1.asmx?op=Add>.

Here is a summary of the various types of files that we have encountered here, and their possible contents.

We introduce here the symbol § to indicate the end of a control statement or tag.

Note the double and triple dot single-character symbols for implied missing code and other shorthands:

- αα... must end with (implicit or explicit) ), ], }, ', ◊, ℱ or §.
- αα...ωω can march right through ( ) ◄ or ▽'s but not ◊ ℱ or §.
- ...ωω can have been through ( ) ◄ ▽'s ◊ or ℱ but not §.

File Type	Script Summary {optional}
*.html	<html>◄...</html>
*.aspx	<script...ℱ▽...§<html>...§
*.aspx	<%@Page.. *.apl"%><html>... §
*.aspx	<%@Register..<html>..<dyalog: ...§... <b>A</b> for custom control
*.apl	{:Namespace...ℱ } :Class...ℱ⊞USING...ℱ▽...ℱ :Property...§
*.apl	:Namespace...ℱ⊞LX...ℱ▽...ℱ..⊞WC... <b>A</b> for console appl <sup>n</sup>
*.aspx	<%@Page Language="dyalog" Inherits="ñ" src="*.dws"%>◄{<html>..}
*.dws	¢ containing :Class...ℱ:Us ing...ℱ▽Page_Load...ℱ..§
*.dws	#.ñ.⊞WC'NetType' 'Page' containing ▽Page_Load... and other exported methods...
*.asmx	<%@WebService..Class=..%>..ℱ:Class...:Base...ℱ...§
*.asax	<script...ℱ▽App_Start...§
*.asmx	<%@WebService Class="ñ. ¢" § <b>A</b> call pre-defined .dll
*.dws	#.ñ.o.⊞WC'NetType' 'WebService' and exported web methods... <b>A</b> export to .dll

Orange rows are signposts to the way ahead. We shall explore in some detail the new APL semantics introduced for writing primitive APL classes in the next and final module of this course.

19.3.3.2 Please demand the final module ☹.

## Module20: Dyalog APL Classes

### § 20.1 User defined Classes

#### §§ 20.1.1 The `:Class` Structure

In Object-Oriented terminology, a *class* is a blueprint, or scripted template, describing how to build an *instance* of a certain type of object from a general programmatic definition. Such an instance, having been created from the class template, has certain properties and can perform certain methods.

In Dyalog APL, a class is created from a script in much the same way as a traditional user-defined program (function or operator) is created from a script. Both are editable via `⎕ED`, both may be fixed from a character representation and both may be **run** or **traced**.

In a sense there already are classes in Dyalog. The 70+ built-in GUI ‘objects’ are really object factories for particular types of objects (`Form`, `Button`, `Group` etc...). However, it is not currently possible to examine their class definitions directly. (The property named `Type` is essentially the *dataType* of an instance.)

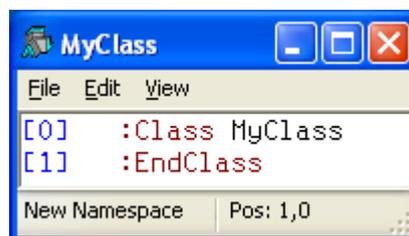
Dyalog version 11 allows you to write classes of your own and generate instances without replicating the underlying program code. Classes have a lot in common with pure namespaces, but with extra functionality. For example, editing and fixing a class, immediately changes all existing instances.

The essence of a class is a control structure in which methods, properties and fields (class *members*) may be defined between the opening and closing keywords. (A *structure* is an essentially multi-line construct that may or may not be amenable to diamondization, depending on the vagaries of the implementation.)

```
:Class MyClass⎕...⎕:EndClass§      Basic class structure wherein members will be defined
```

In an analogous manner to `⎕ED` `▽MyFunction`, a new class may be edited by the circle (o) key symbol.

```
⎕ED oMyClass
```



All class-specific details are defined inside this control structure.

```
{RefSc}←{BSc}⎕FIX VecCharVec      Fix a visible (if BSc=default=1) class (with Ref)
```

Alternatively, like function definition by `⎕FX`, a character representation of a class may be fixed by `⎕FIX`.

```
⎕FIX':Class MyClass' .. ':EndClass'
```

This creates a new sort of **OO program** called `MyClass` that is reported by objects and classes commands.

```
)Classes      Lists all classes in the space
```

```
)OBS
```

```
MyClass
```

```
)CLASSES
```

```
MyClass
```

As yet there exist no instances of our class, only the (empty) class definition and its reference (name).

```
InstRefSc←⎕NEW ClassRefSc      Creates an instance of a class (here with no arguments)
```

New instances of a class may be created (*instantiated*) with `⎕NEW`, and what is more, like `⎕DQ`,

### **YOU CAN TRACE INTO IT!**

In this way you can follow the steps in the formation of an instance of a class.

```
MyInst←⎕NEW MyClass
```

```

)Classes
MyClass
)Obs
MyClass MyInst

```

`InstRefVec←⊞INSTANCES ClassRefSc`    **Ⓜ** Return all instances of a class

All the instances of a particular class (and their ancestry) are returned by `⊞INSTANCES`.

```

⊞INSTANCES MyClass
#. [MyClass]
ρ⊞INSTANCES MyClass ↪ 1

```

We can create a vector of instances

```

MyInsts←⊞NEW`5ρMyClass
MyInsts ↪ #.[MyClass] #.[MyClass] #.[MyClass] #.[MyClass] #.[MyClass]
ρMyInsts ↪ 5

```

whose names are reported by the Name List system function applied to the given name class.

```

⊞NL 2 ↪ MyInsts
↓⊞NL 9 ↪ MyClass MyInst

```

`OldCharArr←⊞DF NewCharArr`    **Ⓜ** Sets the display form of the current space

The display form of all objects - namespaces, GUI objects, classes or instances – may be assigned to any character array. For example,

```

MyInsts.⊞DF 5↑2ρ`⊞A
MyInsts ↪ AA BB CC DD EE

```

Class definitions may be nested within broader classes, and class definitions may specify other (base) classes or interfaces from which methods and properties may be inherited.

`:Class MyClass : MyBase⍏...⍏:EndClass`    **Ⓜ** Classes with inherited characteristics from a base class

The base class may be a Dyalog user-defined class, a .NET class, an interface or a **Dyalog GUI class**. In this case the object following the irrational second colon in the first line of the class structure must be surrounded by **quotes** as in, for example, `:Class MyGUI : 'Form' ...`. Note that GUI objects may be created with `⊞NEW` (in addition to `⊞WC`) by again surrounding the class name in quotes, *eg* by `⊞NEW←'Form'`.

`VecCharVec←⊞SRC ClassRefSc`    **Ⓜ** Returns a character representation of a class

By analogy with

```
'foo'≡⊞FX ⊞CR'foo' ↪ 1
```

we can write

```
MyClass≡⊞FIX ⊞SRC MyClass ↪ 1
```

`RefSc←⊞THIS`    **Ⓜ** Returns a reference to the current space

This system variable may be defined by either identity

```
⊞THIS≡±'⊞NS'' ↪ 1
```

or

```
⊞THIS≡±⊞CS'' ↪ 1
```

20.1.1.1 Create some instances of a particular class and investigate the extent to which they mirror the behaviour of namespace clones.

## §§ 20.1.2 The `:Field` Statement

A class is like a namespace, and a field in a class is like an APL variable in the namespace.

```

:Field { Private } { Instance } {ReadOnly} MyField A Defines a field called MyField
      { Public  } { Shared  }
    
```

The keystings in the field statement may be `Private` (the default if elided) or `Public`, `Instance` (the default if elided) or `Shared`. A public field is visible outside an instance or outside a class (if `Shared`). It may also be defined as `ReadOnly` in which case it behaves rather like an ENUM.

The initial value of a field may be assigned in the `:Field` statement through any APL expression.

```

:Field Public MyField←expr A Defines a predefined public field called MyField
    
```

This is the most usual form of definition of a field. It is accessible from outside an instance of the class in which it is specified, and it is initialised by an assigned expression at the end of the `:Field` statement.

If the field statement includes the kestring `Public` and also the kestring `Shared` then the field is accessible from outside an instance of the class and from outside the class itself.

20.1.2.1 Write a class definition such as `FldClass` below and attempt to read and assign the values of the fields `A`, `B`, `C` and `D` directly from the class, and from an instance of the class. Find the result of `⊖NL -2` in the class and in an instance. Is there any operational difference between field `A` and variable `E`?

```

:Class FldClass
  :Field A
  :Field Public B
  :Field Public C←1
  :Field Public Shared D←2
E←3
:EndClass
    
```

20.1.2.2 Explain the resulting values of fields `C` and `D` after

```

RefArr←⊖NEW`3 3pFldClass
RefArr.C←3 3p16
RefArr.D←3 3p16
RefArr.A←3 3p16
    
```

Is `RefArr.A` above a field? Set the display form of each instance in `RefArr` individually.

20.1.2.3 In instances of the nested classes below, change the value of both of the fields.

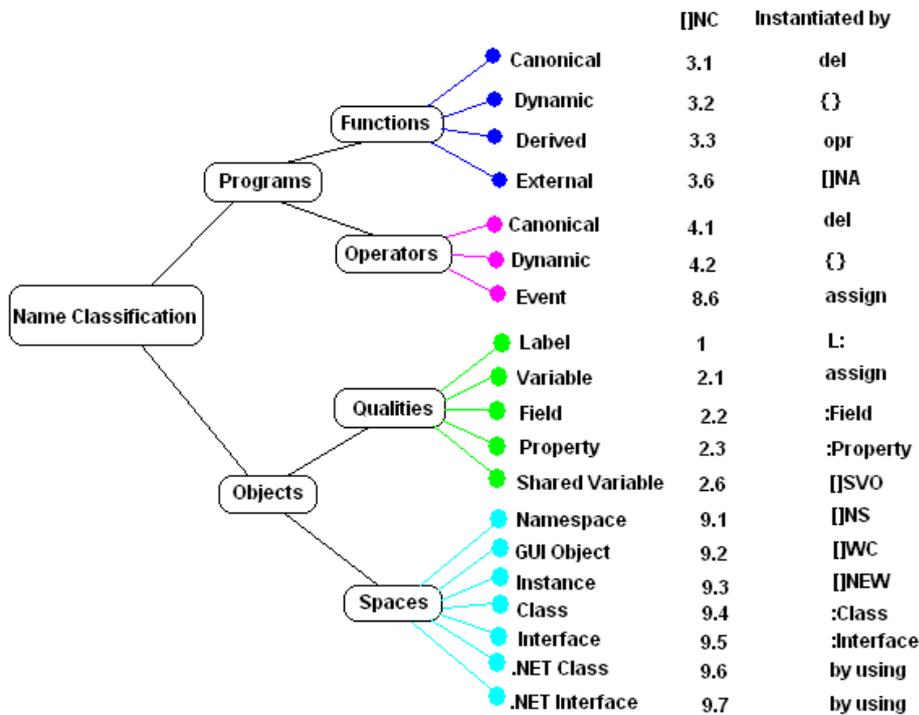
```

:Class MyClass1
  :Field Public MyField1←1
  :Class MyClass2
    :Field Public MyField2←2
  :EndClass
:EndClass
    
```

20.1.2.4 Make the fields `Shared` and note the difference. Is the value of a field in an instance always the same as that in the corresponding class?

### §§ 20.1.3 Name sub Classifications of `⎕NC`

In version 11, names are *sub classified*. Below is one attempt to categorise these sub classifications.



The above view of the new classification scheme is not authoritative but may be helpful in gaining a feel for the relative meanings of the new categories introduced in 3<sup>rd</sup> and 4<sup>th</sup> generation APLs. In particular events are categorised under operators as they essentially take an operand of a user defined (callback) function. Nameclass 2.6 also applies to external properties, eg properties of `OLEClient` objects or .NET instances. (Note that external properties have to be used or accessed first before they become visible to `⎕NC`.)

As well as by some script editor, categories 3.1 and 4.1 may also be created using `⎕FX`, and categories 9.1 and 9.4 may also be created using `⎕FIX`. And category 9.4 may alternatively be viewed as a program.

20.1.3.1 Examine the differences in, for example, `ws ..\samples\OO4APL\Chapter9.dws`, between the results of `(⎕NC ⎕NL 19)` and `(⎕NC ⎕NL -19)`.

APL namespaces can be defined in script files via the `:Namespace` structure. We first came across this structure in a .apl script file in §19. This structure can contain all the usual elements of namespaces as previously introduced, including classes and other namespaces.

```
:Namespace MyNamespace ⍝...⍝ :EndNamespace    Basic namespace script structure
```

Conversely, classes may include namespaces by means of the `:Include` keyword in a class definition.

```
:Include ñ                                    Makes contents of ñ accessible within a class
```

Classes that invoke .NET classes may incorporate the `:Using` keyword as an alternative to the system variable `⎕USING` inside a class definition.

```
:Using ñ{,ass}                                ⎕USING←ñ{,ass}
```

## § 20.2 Methods and Properties in Classes

### §§ 20.2.1 The ▽ (Method) Structure

An APL function may be defined inside a class definition between del (▽) symbols, like in APL 1. It is also possible to define dynamic functions using braces, or to fix a function definition with □FX.

```
▽MyFunctionHeader⍺...⍺▽      ⌘ Basic function definition structure
```

```
:Access { Private } { Instance }      ⌘ Declares the access attributes of a function
         {           } {           }
         { Public  } { Shared   }
```

A method is a non-dyadic function with proscribed access. Only **Public** methods may be called from an instance (or directly from a class if **:Access Public Shared**).

```
▽MyFunctionHeader⍺:Access Public⍺...⍺▽      ⌘ Basic method definition structure
```

20.2.1.1 Show that it is possible to call a **Public Shared** method, such as **MyMethod1** below, from instances or directly from the class. Add a field and include it in the method, perhaps as the left argument of iota (ι).

```
:Class MyClass1
  ▽ R←MyMethod1 Int
    :Access Public Shared
    R←ιInt
  ▽
:EndClass

MyClass1.MyMethod1 9 ↪ 1 2 3 4 5 6 7 8 9
(□NEW MyClass1).MyMethod1 9 ↪ 1 2 3 4 5 6 7 8 9
```

Classes may be fixed from vectors of character vectors. Control words and key strings are not case sensitive.

```
□FIX':class c1' ':field public shared var←0' '▽r←foo w' 'r←w*2' '▽' ':endclass'
```

If a method is to be called from a language other than **Dyalog APL** then it is necessary to define precisely the **dataType** of the arguments and result. This definition is achieved by means of the **:Signature** statement which makes use of the .NET dataTypes outlined in §0.

```
:Signature FunctionSyntax      ⌘ Signature declaration statement
```

An example of its use is given below. Notice that a **:Using** statement is required in order to access the **System.Int32** object from .NET.

```
:class c2
:using System
  :field public shared var←0
  ▽ r←foo w
  :Access public shared
  :Signature Int32←foo Int32
  r←w*2
  ▽
:endclass
```

When exploring ASP.NET in §19.3, with Dyalog APL as the scripting language, we came across a web service script, eg1.asmx, that contained a class based on the .NET `System.Web.Services.WebService`. The details of this script, in particular the `:Signature` statement, should now be clear.

20.2.1.2 Load workspace `..\Samples\asp.net\tutorial\fruit.dws` and examine the `FruitSelection` class.

```

:Class FruitSelection: Page
  :Using System.Web.UI, System.Web.dll
  :Using
  :Access Public
  ▽ Page_Load
    :Access Public
    :Signature System.Void+Page_Load
    :If 0=IsPostBack
      list.Items.Add<'Raspberries'
      list.Items.Add<'Blackberries'
      list.Items.Add<'Grapes'
      list.Items.Add<'Mangoes'
    :EndIf
  ▽
  ▽ Select args
    :Access Public
    :Signature System.Void+Select System.Object obj, System.EventArgs e
    out.Text<'You selected ', list.SelectedItem.Text
  ▽
:EndClass

```

Navigate to <http://81.187.162.51/tutorial.net/frintro6.htm>. This tutorial example is based on a version of file `..\Samples\asp.net\tutorial\intro6.aspx` which invokes the above class.

```

<%@Page Language="Dyalog" Inherits="FruitSelection" Src="Fruit.dws" %>
<html>..</html>

```

Run the tutorial online and study the explanations given. Notice that the class inherits from the .NET class `System.Web.UI.Page`. Hence the need for the first `:Using` statement. The second (empty) `:Using` statement is needed so that the `:Signature` statements can locate all dataTypes derived from `System.Object` class.

### §§ 20.2.2 The `:Implements Constructor` Statement

The `Page_Load` function is rather special in that, if it exists in the class definition, then it is run automatically every time the class is instantiated.

Generally however, in a user defined class one must declare a Public method to be a *constructor* function in order to have the function run on creation of an instance.

```

:Implements Constructor      A Statement declares a method to be run on instantiation

```

A function which is declared to be a constructor function can not return a result and must be `Public`. The function may be niladic as in the case of `Page_Load` above, or monadic as in the case of `MyClass2` below.

```

:Class MyClass2
  :Field Public MyField2
  ▽ MyMethod2 Int
    :Access Public
    :Implements Constructor
    MyField2←ιInt
  ▽
:EndClass

```

When instantiated with a **scalar** argument of 5, say, then `MyMethod2` sets the value of `MyField2` to `ι5`.

```

(⊞NEW MyClass2 5).MyField2 ↪ 1 2 3 4 5

```

It is possible to employ the iota symbol as *index generator* with a numeric **vector** argument (in which case the argument will match the shape of the result).

```
(⊖NEW MyClass2 (3 3)).MyField2
1 1 1 2 1 3
2 1 2 2 2 3
3 1 3 2 3 3
```

This behaviour is called *overloading*. In general it requires special treatment such as:

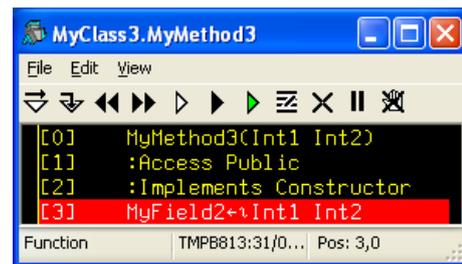
```
:Class MyClass3
  :Field Public MyField2
  ▽ MyMethod2 Int
    :Access Public
    :Implements Constructor
    MyField2←ιInt
  ▽
  ▽ MyMethod3(Int1 Int2)
    :Access Public
    :Implements Constructor
    MyField2←ιInt1 Int2
  ▽
:EndClass
```

Tracing the statement

```
p(⊖NEW MyClass3(5 5)).MyField2
```

shows that the constructor with 2 arguments is selected in this case, giving

```
5 5
```



In this way many monadic constructor functions may be specified in a class definition, each with a different right argument structure. The one that is actually run in any given situation is determined by the structure of the given argument. (This is what happens under the covers in the case of many primitive APL functions, such as the index generator, which encapsulate more than one underlying algorithm.)

20.2.2.1 Write a simple class that has a niladic constructor which initialises the value of some field. Change the constructor function valence to monadic and initialise the field with the argument given to the constructor (the second element of the argument given to `⊖NEW`).

```
:Implements Constructor :Base expr A Calls base constructor with arg given by expr
```

The `:Implements Constructor` statement may be supplemented with `:Base` followed by an APL expression. The result of this expression is taken as the argument to the constructor function of the class from which the current class inherits its behaviour (*viz* the class name, assuming there is one, following the *irrational* colon after the class name in the `:Class` header line). The constructor of the base class is immediately run with this argument.

```
:Implements { Constructor }
              { Destructor }
              { Trigger   }
              A Implements Statements
```

A method can contain a `:Implements Destructor` statement in which case the method is run when the last reference to an object is expunged.

A function can contain a `:Implements Trigger Name1,Name2,...` statement in which case the function is executed if any of the variables in the list `Name1,Name2,...` is changed.

20.2.2.2 Write a simple function with a trigger statement and show that this is run when the trigger variable is changed. See Dyalog version 11 [Help][Latest Enhancements] for an explanation of how to access the old and new values of the variable.

The primitive GUI objects built into Dyalog APL behave very like APL classes except that the name of the object must be placed in quotes when given as an argument to `□NEW`, or when referenced as a base class in a `:Class` definition.

20.2.2.3 Create an instance of a `Form` using syntax `□NEW 'Form' ''` or `□NEW <'Form'`. Hence rewrite `▽makeGrid▽` on the cover (page 1) of this course using `□NEW` rather than `□WC`.

20.2.2.4 Create a class based on a `Form` using syntax `:Class MyClass : 'Form'`. In the constructor function, create a black `Static` on the `Form`. Check the hierarchy using new system function `□CLASS`.  
Hint: `ST←□NEW'Static'(<'BCol'(0 0 0))`

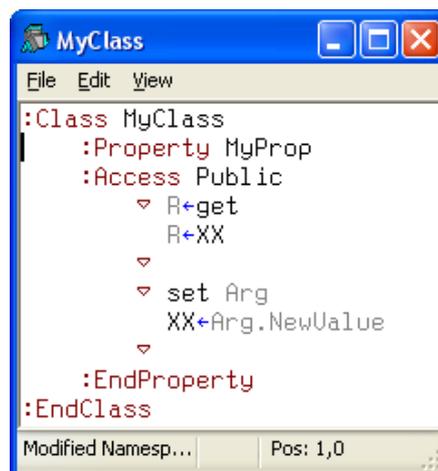
### §§ 20.2.3 The `:Property` Structure

A property is like an *adjective* that describes some attribute of an object. (A method is like a *verb* that specifies some action that an object can perform, and a field (and an object itself) is like a *noun*, which has some value.)

`:Property MyProperty I...I:EndProperty`    a Property Structure

A property of an object is implemented as a `:Property` structure. Inside the structure is a `:Access Public` statement in order to make the property accessible outside an instance. Also, within the property structure may be a case insensitive niladic `▽get▽` function that returns the value of the property, and a case insensitive monadic `▽set▽` function whose (internal instance) argument contains the new property value in the field `NewValue`.

For example, the class definition below bases the simple property `MyProp` on the value of a hidden variable `XX`.



```

MyInst←□NEW MyClass
MyInst.MyProp←19
MyInst.MyProp ↪ 1 2 3 4 5 6 7 8 9
  
```

Of course, any amount of processing could be included in the `▽get▽` and `▽set▽` functions (whose names are allowed to be postfixed with other characters).

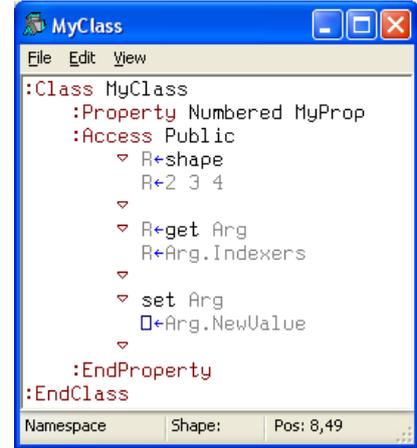
As well as **Simple** properties (the default if elided), there are **Numbered** properties and **Keyed** properties. In the case of a **Numbered** property, the property structure normally has a monadic  $\nabla\text{get}\nabla$  function and a niladic or monadic  $\nabla\text{shape}\nabla$  function.

```

      (Simple  )
:Property {Numbered } Name1{,Name2,...}  $\mathcal{I}$ :Access Public $\mathcal{I}$ ... $\mathcal{I}$ :EndProperty
      (Keyed   )
  
```

The definitions of these types of properties, and the extra **Default** keystring for **Numbered** properties should be explored in the [Help][Latest Enhancements] or in the new and very informative *Object Oriented Programming for APL Programmers* to be found in the file `..\manuals\OO for APL'ers, 2006-06-22.pdf`

20.2.3.1 Given a class with the **Numbered** property defined in the window on the right, trace and interpret the results of expressions:

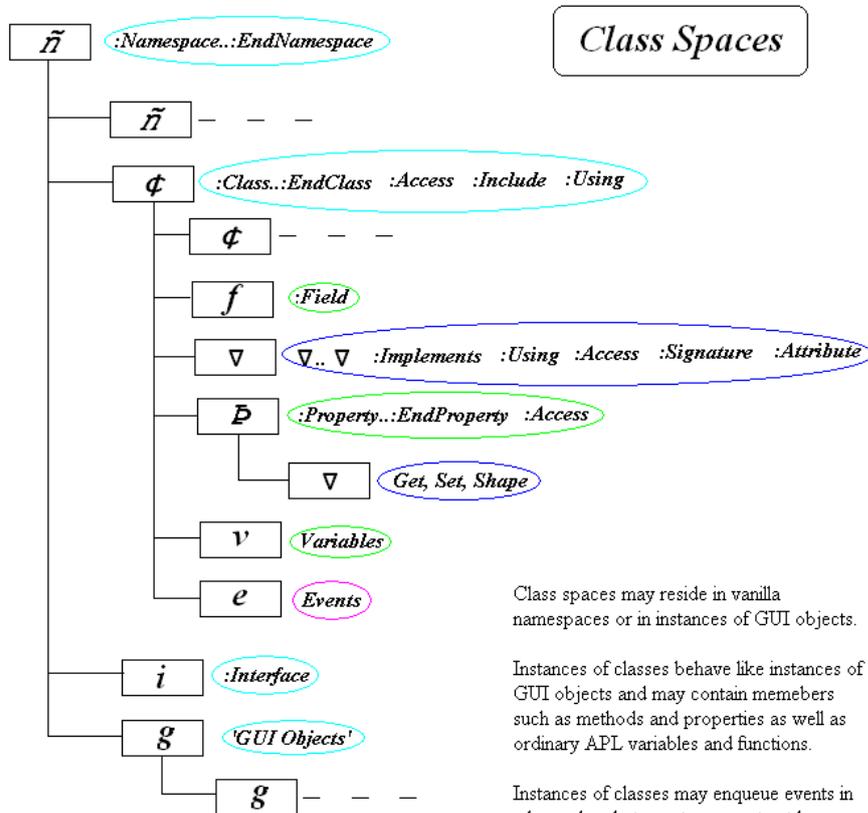


```
(NEW MyClass).MyProp
```

and

```
(NEW MyClass).MyProp←2 3 4π199
```

20.2.3.2 Create a class having a monadic constructor that takes an argument of a file name and ties the file, creating it if necessary. Introduce a property that returns or sets the value of the first component of the file.



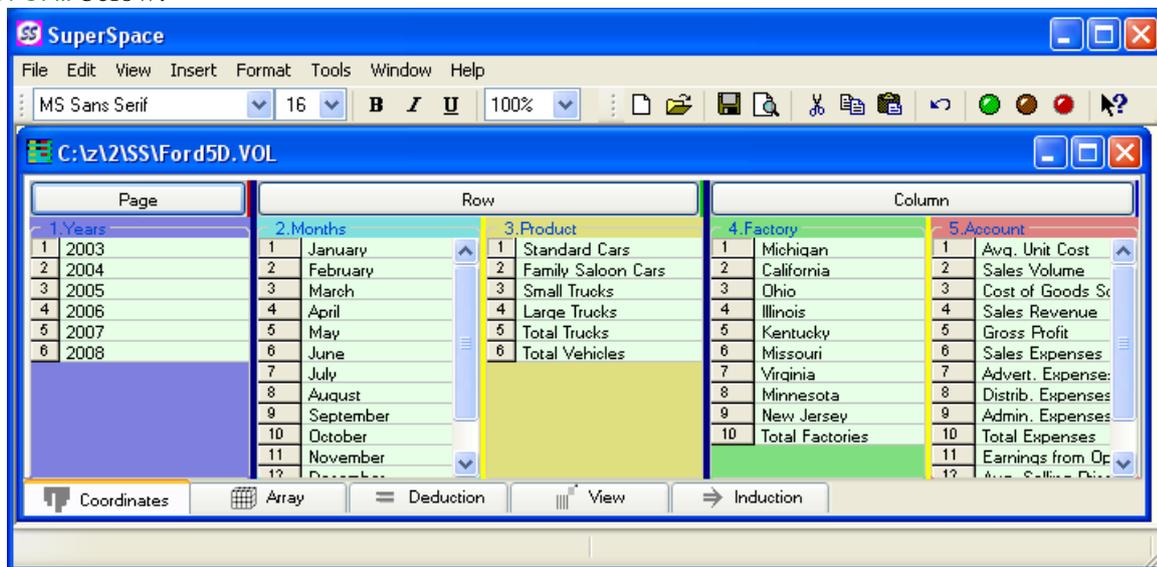
## § 20.3 Architecture with Class Factories

### §§ 20.3.1 Designing an Object Model

When writing an APL function, it is generally a good idea to carefully consider the header line. What goes in and what comes out and what to call the function and what to call the arguments and result. The first two questions are most important, and determine the functionality completely. The other questions are significant in that they facilitate use; you wouldn't invent a new verb meaning "to tidy up your garage" and call it *xyxy*, and you wouldn't call a new type of garden hose a *tttttt*, would you?

Likewise when constructing a new application based on classes it is a good idea to carefully consider what objects are needed and how they will fit together. A good layout for object-related concepts is given in the *Dyalog Object Reference* manual. Objects are summarised by their potential parent and child types and lists of the object's properties, events and methods. Once the basic idea of a GUI object has been grasped, a glance at such a summary often supplies all the information required to proceed with the application.

For example, let us imagine that we are designing a multi-dimensional application that uses an Index object to group dimensions into those on pages, those on rows and those on columns, in the manner shown on the MDI *SubForm* below.



We wish to use this same Index object in other contexts such as that below, and therefore it makes sense to build an Index class that blueprints an entire Index object (made up of *SubForms*, *Buttons*, *Splitters* etc...) for any suitable context (*ie* on any valid parent).



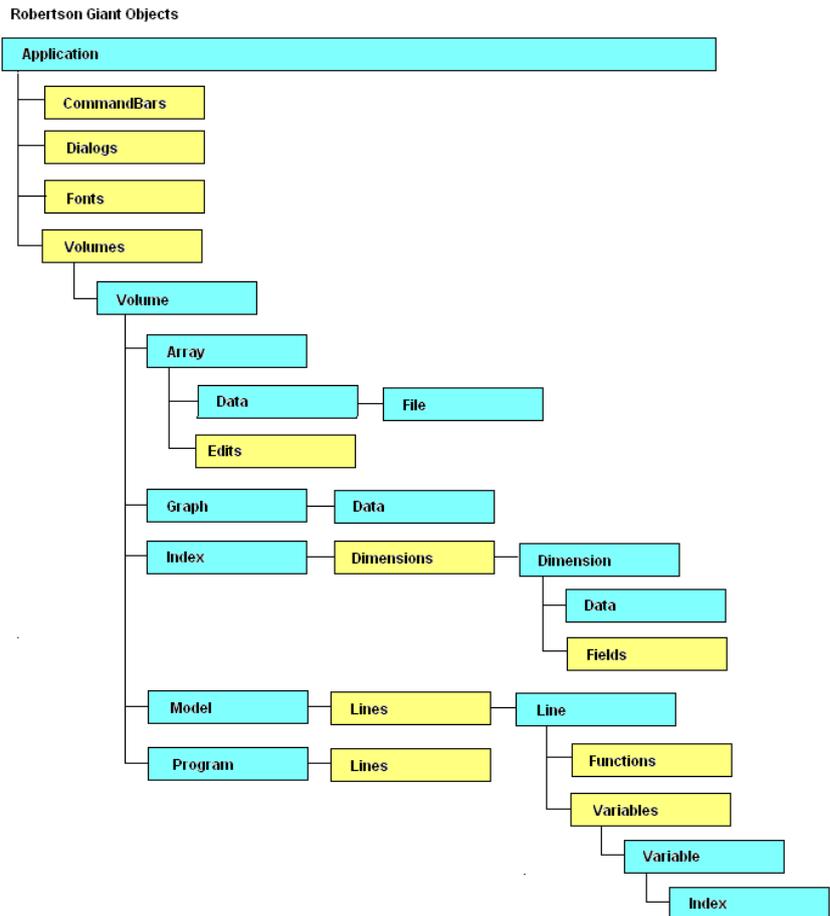
Therefore, before diving in to write APL code (which we should defer until we know reasonably well what we are supposed to be implementing), we should take a little time to specify exactly where we intend to go so that we are less likely to hit unforeseen design faults and other unnecessary limitations. We could start to do this by filling in the object summary below.

<i>Index</i>	<i>Object</i>
--------------	---------------

<b>Purpose</b>	The Index object is a container for three Dimensions collections. Each Dimensions collection contains a number of Dimension objects that specify labels along one axis.
<b>Parents</b>	Volume, Variable
<b>Children</b>	Dimensions
<b>Properties</b>	Type, PageRowSplitPosn, RowColumnSplitPosn, Event, SplitterColours, PRC, PRCColours
<b>Events</b>	SelectPage, SelectRow, SelectColumn, DragPageRowSplitter, DragRowColumnSplitter
<b>Methods</b>	AutoPosnSplitters

Designing systems based on classes introduces this new architectural discipline that can assist in specifying and coding applications. This *OOP* tool of thought is analogous to the familiar *arguments and results* design model for functional programming that can and does assist with lower level coding.

With these building blocks a top-level model might be constructed that summarises the entire application. Getting this object model ‘right’ is usually an iterative process.



See the Word and Excel help files such as **VBAWRD9.CHM** and **VBAXL10.CHM** for classic examples.

## §§ 20.3.2 Building with Objects

The Dyalog GUI affords some simple examples of GUI objects built from simpler GUI objects (themselves from the old API?). Consider for example the *Grid* object, built from *Button* and *Edit* objects.

20.3.2.1 Look at some other Dyalog GUI objects and try to find examples of multi-object constructs. Write a class definition that instantiates a new multi-GUI-object construct, for example a *LabelEdit* object.

The question arises, “Where in the workspace should I place my class definitions?” Should they be in namespaces in a hierarchy that mirrors the object model? No, because there is no unique position for objects that are used in more than one context. Should they be on file? Not initially, at least. Then where? Paul Mansour’s *flipdb* application (see <http://www.flipdb.com>) is beautifully designed and suggests that classes are placed at the root level and then simply need a *#.* in front of their name in order to invoke them in any part of the application. This seems a nice simple suggestion – in general classes have no absolute hierarchy until instantiated, therefore we may, not unreasonably, define them all at the root level.

## §§ 20.3.3 Encapsulating, Inheriting and Morphing

Encapsulation, Inheritance and Polymorphism are said to be the three pillars of Object Oriented Programming (OOP). APL 1 (core APL) already had significant examples of encapsulation and polymorphism, and APL 3 (GUI) has fine examples of inheritance. Classes in Dyalog version 11 bring explicit examples of OOP that are immediately recognisable to C++ and VB programmers.

*Encapsulation* is the practice of hiding internal workings from external scrutiny and revealing only those aspects that have been chosen as relevant to the outside world. This concept is most explicit in OOP where an object reveals its characteristics and behaviour through a set of well-defined ‘members’. The concept is, however, already well understood in APL 1 where good function definition encourages localisation of all variables that are irrelevant to the intended use. Even the simple plus sign (+) in any digital computer language hides the internal binary processes, whose revelation would confuse rather than enlighten the user.

*Inheritance* is a concept indicating that certain characteristics at one level are passed on to the next level. We have seen how classes may be based on other classes and acquire their members; for example *System.Int32* is based on *System.Object*. Even in APL 1 we can surmise that matrix divide ( $\boxdiv$ ) is based on simple division – even the symbol face has a family resemblance! Inheritance gives a new dimension to encapsulation where functionality of ancestors may be employed by future generations without the need to replicate the functions behind the behaviour or the data behind the family attributes.

*Polymorphism* means ‘having many forms’. Dogs (*Canis familiaris*), for example, take many forms, but they are all dogs. So a class may be *overloaded* with many different possible arguments (the second parameter in  $\boxnew$ ) to produce different objects with related, but not identical, attributes and behaviour. This is one example of polymorphism in OOP. Even in APL 1 primitive functions such as replicate ( $\box/$ ) and expand ( $\box\$ ) may take numeric or character right arguments and give a related, though different, form of result. This is polymorphism of a sort. This is called *operator overloading* in C++. APLers experience profound ‘operator overloading’ in APL notation;  $+ \cdot \times \wedge . = \lceil \cdot \lceil \dots \text{A} ; - \ominus$

As systems grow more complex, computer science borrows more terms from biology. The analogy between the human brain and computer systems has always been close. Now deeper analogies with life forms in general are being forged and there seems no end to the abstract correspondence which computing in general can achieve. APL is a powerful *tool of thought* with a star-studded history. The APL language still ahead of all other *executable arithmetic notations* and continues to *provide real solutions* of all sorts of disciplines.

20.3.3.1 See it as your operating system and take control of your computing needs with **Dyalog APL** ☺.

## FEEDBACK FORM

Name ..... eMail .....

Date ..... Location .....

Course ..... Instructor .....

Please indicate your assessment of the following:

poor    1    2    3    4    5    excellent

--	--	--	--	--

Location & Facilities

--	--	--	--	--

Course Content (by module if possible?)

--	--	--	--	--

Course Material (by module if possible?)

--	--	--	--	--

Instructor's Knowledge (by module?)

--	--	--	--	--

How useful was the course to your role?

Please suggest improvements to the course.

---

---

---

---

Any other critical comments are welcome.

---

---

---

Please give this form to your tutor, or send it to **ROBERTSON (Publishing)**.



# Certificate of Achievement

This is to certify that

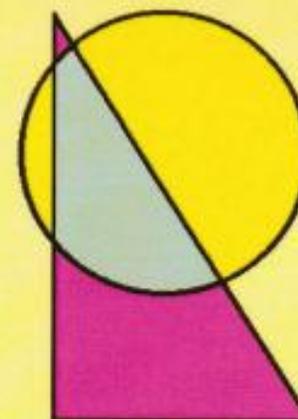
\_\_\_\_\_

has successfully completed the introduction to

## APL 3 & APL 4

on

\_\_\_\_\_



Graeme Robertson Ltd.

**R** £29.99 UK / \$59.99 US

ISBN 0-952-41672-7

02999



9 780952 416722

Signed \_\_\_\_\_

Instructor