The tool of thought for expert programming

Dyalog™ for Windows

Release Notes

Dyalog Limited

Grove House
Lutyens Close
Chineham Court
Basingstoke
Hampshire, RG24 8AG
United Kingdom

tel: +44 (0)1256 338461
fax: +44 (0)1256 316559
email: support@dyalog.com
http://www.dyalog.com

Version 11.0.1

First Edition September 2006

Contents

CHAPTER 1

General

Interoperability and Compatibility

Introduction

Workspaces and component files are stored on disk in a binary format (illegible to text editors). This format differs between machine architectures and among versions of Dyalog. For example a file component written by a PC will almost certainly have an internal format that is different from one written by a UNIX machine. Similarly, a workspace saved from Dyalog Version 11 will differ internally from one saved by a previous version of Dyalog APL.

It is convenient for versions of Dyalog APL running on different platforms to be able to *interoperate* by sharing workspaces and component files. However, this is not always possible. For example, if a new internal data structure is introduced in a particular version of Dyalog APL, previous versions could not be expected to make sense of it. In this case the load (or copy) from the older version would fail with the message:

```
      this WS requires a later version of the interpreter.
```

Similarly, *large* (64-bit-addressing) component files are inaccessible to versions of the interpreter that pre-dated their introduction.

The second item in the right argument of `⎕FCREATE` determines the addressing type of the file.

```
    'small'⎕fcreate 1 32    ⍝ create small file.
    'large'⎕fcreate 1 64    ⍝ create large file.
```

For the moment, if the second item is missing, the file type defaults to 32-bit-addressing (max 4GB file size), even on a 64-bit system for maximum inter-operability. We will change the default to 64-bit in Version 11.1, by which time we believe the bulk of our users will be running versions which can use 64-bit files (Version 11.0 or later)

Dyalog APL Version 11 adds to the interoperability problem by supplying versions for both 32-bit and 64-bit machine architectures.

Interoperability is summed up in the following tables. Table rows show the version that is attempting to access the file or workspace and columns show the version that saved it:

This version can access files created by this version →
↓

Version 10.1.5

Version **10.1.5** is an update to V10.1 issued specifically to enhance 10.1/11.0 compatibility. Version 10.1.5 is essentially version 10.1.2 with the addition of some Version 11.0 file system code. If Version 10.1 applications are moved to 10.1.5, they will be able to share files with code which has been moved to 64-bit Version 11.0. For example, this would allow a computational server to be moved to 64-bit Version 11.0 and provide data to an application which was still running Version 10.1.

The row and column titles show the Dyalog version **9.0**, **10.0**, etc; **(32)** and **(64)** indicate a version running on a 32-bit or 64-bit machine architecture, respectively.

Implementation

In general all writes are made in the format that is native to the writer. Readers do the work of any necessary translation. The exception is when writing from a 64 bit version to a 32 bit file. This has been allowed provided the machine architecture is the same. 32 bit files are the same architecture for the entire file. 64 bit files can have each component written differently.

Workspace interoperability

can load/copy from …

	9.0	**10.0**	**10.1**	**10.1.5**	**11.0(32)**	**11.0(64)**
9.0	Yes	-	-	-	-	-
10.0	Yes	Yes	-	-	-	-
10.1	Yes	Yes	Yes	Yes	-	-
10.1.5	Yes	Yes	Yes	Yes	-	-
11.0 (32)	Yes	Yes	Yes	Yes	Yes	Yes
11.0 (64)	-	-	Yes	Yes	Yes	Yes

Component files (32-bit) and External variables

can access …

	9.0	10.0	10.1	10.1.5	11.0(32)	11.0(64)
9.0	Yes ~t	Yes ~f~n~t	Yes ~f~n~t	Yes ~f~n~t	Yes ~f~n~t	Yes ~f~n~t
10.0	Yes ~t	Yes ~t	Yes ~t	Yes ~t	Yes ~n~t	Yes ~n~t
10.1	Yes ~t	Yes ~t	Yes ~t	Yes ~t	Yes ~n~t	Yes ~n~t
10.1.5	Yes ~w	Yes ~w	Yes ~w	Yes ~w	Yes ~n~w	Yes ~n~w
11.0 (32)	Yes ~w	Yes ~w	Yes ~w	Yes ~w	Yes ~w	Yes ~w
11.0 (64)	Yes ~w	Yes ~w	Yes ~w	Yes ~w	Yes ~w	Yes ~w

Notes
`~f`: Cannot read *⎕OR*s of **functions**.
`~n`: Cannot read *⎕OR*s of **namespaces**.
`~t`: Cannot **tie** files created on machines with different byte ordering (e.g. PC/UNIX).
`~w`: Can read from but cannot **write** to files created on machines with different byte ordering (attempting to write generates *`FILE ACCESS ERROR`*).

Component files (64-bit)

can access …

	9.0	10.0	10.1	10.1.5	11.0(32)	11.0(64)
9.0	-	-	-	-	-	-
10.0	-	-	-	-	-	-
10.1	-	-	Yes	Yes ~b	Yes ~n~b	-
10.1.5	-	-	Yes	Yes	Yes ~n	Yes ~n
11.0 (32)	-	-	Yes	Yes	Yes	Yes
11.0 (64)	-	-	Yes	Yes	Yes	Yes

Notes
`~b`: Cannot read a component with the wrong byte-ordering.
`~n`: Cannot read *⎕OR*s of **namespaces**.

Sockets (Type `'APL'`)

can decode from …

	9.0	10.0	10.1	10.1.5	11.0(32)	11.0(64)
9.0	Yes	Yes ~fn	Yes ~f~n	Yes ~f~n	Yes ~f~n	-
10.0	Yes	Yes	Yes	Yes	Yes ~n	-
10.1	Yes	Yes	Yes	Yes	Yes ~n	-
10.1.5	Yes	Yes	Yes	Yes	Yes ~n	-
11.0 (32)	Yes	Yes	Yes	Yes	Yes	Yes
11.0 (64)	Yes	Yes	Yes	Yes	Yes	Yes

Notes
`~f`: Cannot read ⎕*OR*s of **functions**.
`~n`: Cannot read ⎕*OR*s of **namespaces**.

Auxiliary Processes

A Dyalog APL process is restricted to starting an AP of exactly the same architecture. In other words, the AP must share the same word-width and byte-ordering as its interpreter process.

Session Files

Session (.dse) files may only be used on the platform on which they were created and saved.

Improved Workspace Management

Introduction

From version 11.0, Dyalog can reduce its process size by returning unused memory to the operating system.

This happens when one of the following occurs:

```
      )CLEAR
      )LOAD
      )SAVE
      )RESET
      ⎕WA
```

Notice that `⎕WA` could be called under program control to reduce the process size after returning from a memory-intensive section of an application.

Workspace Size and Compaction

The *maximum* amount of memory allocated to a Dyalog APL workspace is defined by the **maxws** parameter.

Upon `)LOAD` and `)CLEAR`, APL allocates an amount of memory corresponding to the size of the workspace being loaded (which is zero for a clear ws) plus the *workspace delta*.

The workspace delta is 1/16th of **maxws**, except if there is less than 1/16th of **maxws** in use, delta is 1/64th of **maxws**. This may also be expressed as follows:

```
      delta←maxws{⌈⍺÷⊃(⍵>⍺÷16)⌽64 16}ws
```

where `maxws` is the value of the **maxws** parameter and `ws` is the currently allocated amount of workspace. If **maxws** is 16384KB, the workspace delta is either 256KB or 1024 KB, and when you start with a `clear ws` the workspace occupies 256KB.

When you erase objects or release symbols, areas of memory become free. APL manages these free areas, and tries to reuse them for new objects. If an operation requires a contiguous amount of workspace larger than any of the available free areas, APL reorganises the workspace and amalgamates all the free areas into one contiguous block as follows:

1. Any un-referenced memory is discarded. This process, known as *garbage collection*, is required because whole cycles of refs can become un-referenced.
2. Numeric arrays are *demoted* to their tightest form. For example, a simple numeric array that happens to contain only values 0 or 1, is demoted or *squeezed* to have a ⎕*DR* type of 11 (Boolean).
3. All remaining used memory blocks are copied to the low-address end of the workspace, leaving a single free block at the high-address end. This process is known as *compaction*.
4. In addition to any extra memory required to satisfy the original request, an additional amount of memory, equal to the workspace delta, is allocated. This will always cause the process size to increase (up to the **maxws** limit) but means that an application will typically achieve its working process size with at most 4+15 memory reorganisations.
5. However, if after compaction, the amount of used workspace is less than 1/16 of the Maximum workspace size (MAXWS), the amount reserved for working memory is reduced to 1/64th MAXWS. This means that workspaces that are operating within 1/16th of MAXWS will be more frugal with memory

Note that if you try to create an object which is larger than free space, APL reports `WS FULL`.

The following system function and commands force a workspace reorganisation as described above :

```
⎕WA, )RESET, )SAVE, )LOAD, )CLEAR
```

However, in contrast to the above, **any spare workspace above the workspace delta is returned to the Operating System**. On a Windows system, you can see the process size changing by using Task Manager.

The system function `⎕WA` may therefore be used judiciously (workspace reorganisation takes time) to reduce the process size after a particularly memory-hungry operation.

Note that in Dyalog APL, the SYMBOL TABLE is entirely dynamic and grows and shrinks in size automatically. There is no `SYMBOL TABLE FULL` condition.

Number Conversion

The conversion of numbers between internal form and display form has been significantly improved for Version 11.0.

Numbers (such as 3.14), input in the session or used as constants in source code, are converted to a binary (IEEE) format for storage in the workspace. If the internal number is subsequently displayed, the reverse conversion takes place as the number is formatted.

The conversion of numbers between internal form and display form has been significantly improved and in Version 11.0:

a) When ⎕PP is set to its maximum value of 17, distinct numbers have distinct display forms.

b) Such forms use the smallest number of digits possible.

A consequence of this is that if ⎕PP is set to its maximum value of 17, floating-point numbers may be converted between binary and character representation without loss of precision. In particular, if ⎕PP is 17 and ⎕CT is 0 (to ensure exact comparison), for any floating-point number *N* the expression *N*=⍎⍕*N* is true (except for denormal numbers).

Denormal Numbers

Numbers, very close to zero, in the range 2.2250738585072009E¯308 to 4.9406564584124654E¯324 are called *denormal* numbers.

Such numbers can occur as the result of calculations and are displayed correctly. However, denormals cannot be specified as literals and are converted to zero on input.

Numbers below the lower end of this range (4.94E¯324) are indistinguishable from zero in IEEE double floating point format.

Note that the converse of (a) is not necessarily true: distinct input forms may convert to the same internal binary number. This is clearly the case if we supply more digits than the 64-bit internal format is capable of representing. In particular, a decimal number such as ÷3 (or ÷10), which has an infinite binary representation must necessarily be represented internally, only approximately. This can lead to a slightly surprising (though correct) display, if ⎕PP is set to 17. For example, we might wonder why only 16 digits of accuracy are displayed in the following:

```
      ⎕PP←17
      ÷3  ⍝ why only 16 digits
0.3333333333333333
```

We see the reason if we display the internal format of `÷3` together with its display form and the display forms of its immediate IEEE neighbours:

```
0x3FD5555555555554 -> 0.33333333333333326
0x3FD5555555555555 -> 0.3333333333333333
0x3FD5555555555556 -> 0.33333333333333337
```

Rule (b) constrains us to format using the smallest number of digits that would convert back to internal number `0x3FD5555555555555` (which is just less than one third).

The Problem

To avoid any loss of precision, literal numbers in source code are displayed with maximum print precision, `⎕PP=17`, by the function editor and `⎕CR`.

Version 11 input conversion is very slightly more accurate than in previous versions. For example, in Version 10.1, a number input as 0.6 would have been converted to internal binary IEEE format as:

```
0.6 -> 0x3FE3333333333334    // V10
```

where in V11, it is converted to the marginally more accurate:

```
0.6 -> 0x3FE3333333333333    // V11
```

In V11, using `⎕PP=17`, IEEE number `0x3FE3333333333333` displays (correctly) as `0.6`, while its neighbour `0x3FE3333333333334` displays (correctly) as `0.6000000000000001`.

This means that source code from versions prior to Version 11 may occasionally show strange-looking numbers such as `0.6000000000000001`, when viewed in Version 11.

Note that this **DOES NOT CHANGE** the accuracy of any calculations: refixing the function with the longer (`0.6000000000000001`) number in Version 11 will continue to convert to the same internal number (`0x3FE3333333333334`) as before. However, changing the number in the source code to `0.6` would convert to a different number (`0x3FE3333333333333`), which might be very slightly closer to what you had in mind.

Changes to)COPY

Namespaces Containing Refs

Version 11 makes a small change to the way that namespaces containing *refs* are copied using `)COPY` or `⎕CY`.

We can show the change most easily by using a simple example.

Suppose a saved workspace contains two namespaces `target` and `sibling`, and `target` contains a ref to `sibling`, which contains variable `var`.

```
.target-----.  .--> .sibling----.
|           |  |    |           |
|   sibref--+--'    | var       |
|           |       |           |
'-----------'       '-----------'
```

Old behaviour

Prior to Version 11, `target`'s outward-pointing ref would be *inverted* on copy.

```
    )copy myws target

.target----------.
|                |
|   .sibling---.  |
|  |           |  |
|  | var       |  |
|  |           |  |
|   '----------'  |
|                |
'----------------'
```

New behaviour

```
    )copy myws target

.target-----.  .--> .(sibling)--.
|           |  |    |           |
|   sibref--+--'    | var       |
|           |       |           |
'-----------'       '-----------'
```

In this case, the original parent/child relationships are preserved. Note however, that the name `'sibling'` is not fixed in the space from which the copy occurred.

More generally, an incoming space, whose ancestry does not include the target of the copy (e.g. `sibling` in the above example), is fixed as an anonymous child of the current space.

Copying Dependant Objects

`)COPY` now issues a warning message when it copies dependant objects into the active workspace.

If you `)COPY` an object without including the names of any objects upon which it depends in the list of names to be copied, such as:

a) an Instance of a Class but not the Class itself
b) a Class but not a Class upon which it depends
c) an array or a namespace that contains a ref to another namespace, but not the namespace to which it refers

the dependant object(s) **will also be copied** but will be **unnamed** and **hidden**. In such as case, the system will issue a warning message.

For example, if a saved workspace named CFWS contains a Class named `#.CompFile` and an Instance (of `CompFile`) named `icf`,

```
      )COPY CFWS icf
.\CFWS saved Fri Mar 03 10:21:36 2006
copied object created an unnamed copy of class #.CompFile
```

The existence of a hidden copy can be confusing, especially if it is a hidden copy of an object which had a name which is in use in the current workspace. In the above example, if there is a class called `CompFile` in the workspace into which `icf` is copied, the copied instance may *appear* to be an instance of the *visible* `CompFile`, but it will actually be an instance of the hidden `CompFile` - which may have very different (or perhaps worse: very slightly different) characteristics to the named version.

If you copy a Class without copying its Base Class, the Class can be used (it will use the invisible copy of the Base Class), but if you edit the Class, you will either be unable to save it because the editor cannot find the Base Class, or - if there is a visible Class of that name in the workspace - it will be used as the Base Class. In the latter case, the invisible copy which was brought in by `)COPY` will now disappear, since there are no longer any references to it - and if these two Base Classes were different, the behaviour of the derived Class will change (and any changes made to the invisible Base Class since it was copied will be lost).

Deferred Prototype Generation

Unlike earlier versions of Dyalog APL, Version 11.0 does not generate a prototype when an empty array is created. The prototype is instead created when an operation on the empty array needs a prototype to generate a result.

This means that Version 11.0 allows the creation of an empty array from an array of namespace references. In previous Versions of Dyalog APL, this gives a `NONCE ERROR`

```
      a←0⍴#
```

The deferred requirement for prototypes is especially useful when making selections from lists of namespaces. For example, if `CDDB` is a vector of namespaces representing a CD database, an expression such as:

```
      CDDB←(CDDB.Artist∊⊂'Dylan')/CDDB
```

.. will always work in version 11.0, while it would have given a `NONCE ERROR` in earlier versions in the unlikely event that there had been no CD's by Bob Dylan in the collection. However, the following expressions still gives a `NONCE ERROR`:

```
      1↑0⍴#
      (0/CDDB).Title
```

… because both expressions require inspection of a prototype in order to compute the result. So it may still be necessary to check the length of the result before using the selection.

Note that, if the `CDDB` was an array of instances of a Class, it would be possible to specify a prototypical CD entry and get the latter expression to return the "expected result" `(0⍴⊂'')`. See Empty Arrays of Instances: How?

Matching Refs

In version 11.0, match (≡) and not match (≢) return 0 and 1 respectively if used to compare two refs which do not point to the same object.

Earlier Versions returned 1 and 0 respectively if the refs were identical, but *NONCE ERROR* if they were different.

Note:

At some point in the future, it is likely that it will be possible for a class definition to contain definitions of comparison functions, and that the existence of such functions would replace pointer matching. This is already the case for instances of .NET Classes, for example:

```
      dt1←⎕NEW System.DateTime (2006 8 28)
      dt2←⎕NEW System.DateTime (2006 8 28)
      dt1≡dt2
1
```

These two instances match, even though they are NOT the same object, because the DateTime class defines a comparison function for instances of this class. Therefore, it is NOT safe to write code which assumes that, if two refs match, they refer to the same object (unless you are the author of the classes being compared).

Changing Name-Class on Assignment

Version 11 allows you to overwrite a ref (name class 9) with a variable (name class 2) and vice-versa.

For example, if we have:

```
      ref←⎕ns''       ⍝ class 9 namespace ref.
```
and
```
      var←3.14        ⍝ class 2 variable.
```

We can now do either:

```
      ref←var         ⍝ change class 9->2.
```
or
```
      var←ref         ⍝ change class 2->9.
```

In particular, if `ref` is a namespace reference, we can say:

```
      ref←,ref        ⍝ 1-item vector: class 9->2.
```
then
```
      ref←⊃ref        ⍝ scalar ref: class 2->9.
```

The table of permitted re-assignments is as follow.

	Ref	**Variable**	**Function**	**Operator**
Ref	Yes	Yes		
Variable	Yes	Yes		
Function			Yes	Yes
Operator			Yes	Yes

Properties that refer to GUI Objects

In the Dyalog GUI, a number of Properties are used to specify and/or report references between GUI objects. For example, the FontObj property of a Label object specifies the Font object used to display its text. In previous versions of Dyalog APL, such objects are referenced *by name*. In Version 11, all the Properties that are used to reference another object accept/report *refs* as well as names.

The list of such Properties includes the following:

BtnPix	CellFonts	ColSortImages	CursorObj
DockChildren	FontObj	Fstyle	HintObj
IconObj	ImageListObj	Input	MDIMenu
Picture	Popup	RowTreeImages	SplitObj1
SplitObj2	TipObj	TabObj	ToolboxBitmap

Note that when you query the value of any of these Properties, APL will report whatever you specified (name or ref) when you set the Property. However, if you have not previously set the value, it will be reported as an empty character vector.

There are however some Properties, whose values are automatically updated by the system, for which this approach would not be appropriate. In these cases, an additional Property is provided with the same name followed by the suffix *Ref*. For example, MDIActive continues to report the name of the active SubForm whereas the new property MDIActiveRef provides a *ref* to it.

The list of such Properties includes the following:

MDIAtive	MDIActiveRef
PageActive	PageActiveRef

New Fonts and Keyboard Files

Version 11.0 includes a new set of APL fonts and an additional set of keyboard files.

The new fonts include the ⍣ symbol (Power operator) and ∊ symbol which replaces left tack.

The Dyalog APL TrueType fonts (Dyalog Std TT and Dyalog Alt TT) have improved appearance characteristics, especially as larger sizes.

The APL385 Unicode font is also included with Version 11. This font is recommended for use with Notepad and other text editors when viewing and editing APLScript files.

The separate unified and traditional mode keyboards provided in previous versions of Dyalog APL have been combined into a single table. For each supported country, the new table is named `cc.din`, where `cc` is the country code. These tables are provided in additional to the ones supplied with previous versions of Dyalog APL.

The keyboards start in unified mode and can be switched to traditional mode (Shift+r for ⍴) by clicking the Uni/Apl field in the status bar or by keying * on the Numeric-Keypad.

The new tables support the entry of the ⍣ and ▯ symbols by pressing Ctrl+Shift+p and Ctrl+Shift+l in either mode. However, the new tables do not support the entry of APL underscored characters ⍙Ⓐ*A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲Q̲R̲S̲T̲U̲V̲W̲X̲Y̲Z̲*

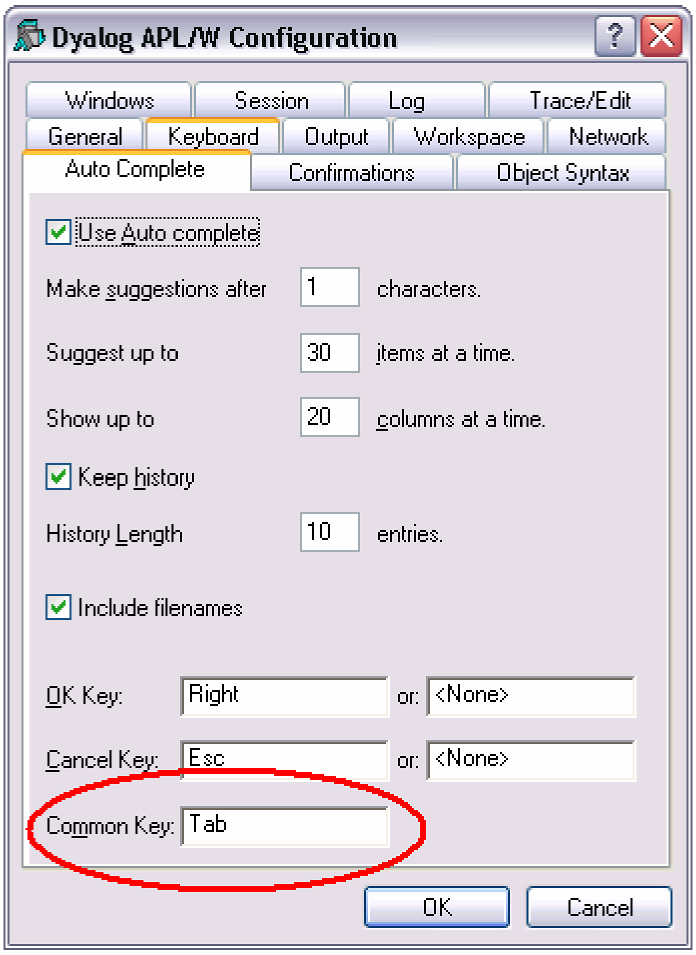
During installation, the appropriate new table will be presented as the default choice.

Note that the standard keyboard tables provided with earlier versions of Dyalog APL are provided in the `\old` sub-directory.

New AutoComplete Feature

Version 11.0 includes an enhancement to AutoComplete that is designed to cater for the use of common prefixes in names.

It is not unusual for developers to adopt a convention of prefixing a group of APL names with a common string of characters. To improve the usefulness of the AutoComplete feature in these circumstances, a new *Common key* has been provided.



[✓] Use Auto complete

Make suggestions after 1 characters.

Suggest up to 30 items at a time.

Show up to 20 columns at a time.

[✓] Keep history

History Length 10 entries.

[✓] Include filenames

OK Key: Right or: <None>

Cancel Key: Esc or: <None>

Common Key: Tab

If you are typing and the AutoComplete window is displayed, pressing the *Common key* will auto-complete the *common prefix*. This is defined to be the longest string of leading characters in the currently selected name that is shared by at least one other name in the list.

For example, if the workspace contains variables `AAaa1`, `AAaa2`, `AAaa3`, `ABbb1`, `ABbb2`, `ABbb3`, `ACcc1` and the system is set to AutoComplete on 1 character, typing `A` will cause the AutoComplete window to display all 7 names.

If `AAaa1` is the currently selected name, clicking the *Common key* will fill in the rest of the common prefix, namely `Aaa`, and reduce the list to `AAaa1`, `AAaa2` and `AAaa3`.

If `ABbb1` is the currently selected name, clicking the *Common key* will fill in the characters `Abb`, and reduce the list to `AAbb1`, `AAbb2` and `AAbb3`.

Whereas if the currently selected name is `ACcc1`, clicking the *Common key* has no effect.

Isolation Mode

There is a new option on the Create Bound File dialog box, when exporting a Class as a Microsoft .Net Assembly (dll), labelled *Isolation Mode*.

For *each* application which uses a class written in Dyalog APL, at least one copy of either `dyalog110.dll` or `dyalog110rt.dll` will be started in order to host and execute the appropriate APL code. Each of these *engines* will have an APL workspace associated with it, and this workspace will contain classes and instances of these classes. The number of engines (and associated workspaces) which are started will depend on the Isolation Mode which was selected when the APL assemblies used by the application were generated. Isolation modes are:

- Each host process has a single workspace
- Each appdomain has its own workspace
- Each assembly has its own workspace

The last two Isolation Modes are new in version 11.0. Previously, each application always used a single engine to run all classes and instances used by that application.

Note that, in this context, Microsoft Internet Information Services (IIS) is a *single* application, even though it may be hosting a large number of different web pages. Each ASP.Net application will be running in a separate *AppDomain*, a .NET object which is an isolated subdivision of the application. Other .NET applications may also be divided into different AppDomains.

In other words, if you use the first option, ALL classes and instances used by any IIS web page will be hosted in the same workspace and share a single copy of the interpreter. The second option will start a new Dyalog engine for each ASP.Net application. The final option will start a new Dyalog engine for each assembly containing APL classes.

Export to Memory

There is a new option on the Session File menu labelled *Export to Memory*.

If you create an APL Class based upon a .Net Type, you must export it as a .Net Assembly before you can use it.

This option allows you to create an in-memory .Net Assembly that you can use to test the Class, without having to repeatedly go through the entire exercise of saving it as a .Net Assembly on disk (as a DLL file) as you develop the code.

Furthermore, using this option, it is not necessary to Close the AppDomain (see Close AppDomain) each time you replace the Assembly. However, be aware that each time you export (to memory), additional memory is used and it may be appropriate to free it (using *Close AppDomain*) periodically.

Note that to *use* an in-memory Assembly, it is not necessary to set `⎕USING`.

You only need to re-export to memory if you make a change to your class which changes the *public interface* of the class. So changes to functions do not require re-exporting, but if you add a new method or change a signature, you must re-export.

Note that, APL will only allow you to use a Class based upon a .Net Type via a .Net Assembly, either an in-memory Assembly or a DLL file on disk. You may not use a Class directly.

Close AppDomain

There is a new option on the Session File menu labelled *Close AppDomain*.

When APL uses a Class which is in a .Net Assembly (normally a DLL file), that Assembly is loaded into a .Net memory area known as the Application Domain, or the *AppDomain* for short. When an assembly is in use by any application, including the current APL workspace, you cannot overwrite the DLL file on disk, so you cannot make changes to classes which need to be exported in order to be used.

Previously, during the development of a Class based upon a .Net Type (`NetType`) it was necessary to `)CLEAR`, re-load the workspace or terminate Dyalog APL each time you needed to update the DLL. Note that APL automatically closes the AppDomain when a workspace is loaded, or on `)CLEAR`.

In order to speed up the development cycle, Version 11 provides a menu item which closes the AppDomain so that you can then overwrite the DLL you were using with a new version.

Note that any instances of .Net classes become null pointers when you do this:

```
      dt←System.DateTime.Now
      dt
04-05-2006 14:06:46

[Close AppDomain]

      dt
(NULL)
```

External Object (COM and .Net) Behaviour

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

Old behaviour:

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

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

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

New behaviour

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

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

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

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

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

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

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

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

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

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

Examples

```
      ⎕WX←1
      'XLW' ⎕WC 'OleClient' 'Excel.Application'
      XLW.Workbooks.Add ⍬
      XLW.ActiveWorkbook.Sheets.(Item 'Sheet2').Index
2

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

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

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

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

Configuring for different Versions of the .Net Framework

Dyalog APL Version 11.0 is compatible with versions 1.1 and 2.0 of the Microsoft.Net Framework.

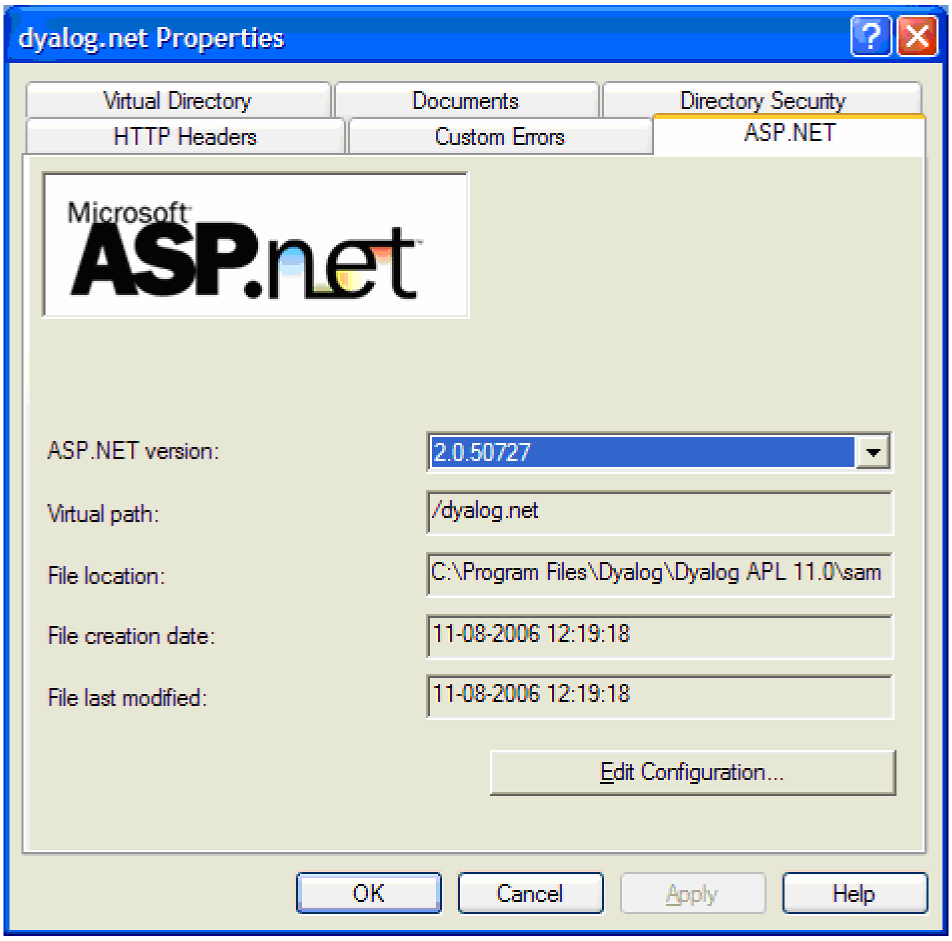
On a machine that has multiple versions of the .Net framework installed, Dyalog APL will use the most recent version by default. If you have both versions and need to use version 1.1 for any reason, there are two configuration files which are distributed with version 11 but need to renamed in order to take effect:

`dyalog.exe.config.1.1` (in the main Dyalog folder, and)
`bin\dyalogc.exe.config.1.1` (which controls the script compiler)

If you rename these files and remove the trailing ".1.1", so that the final file extension becomes "config", they will take effect, and cause the development environment and the script compiler (respectively) to use version 1.1 of the framework. The contents of these files are as follows:

```
<?xml version ="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v1.1.4322" />
  </startup>
</configuration>
```

Note that ASP.NET may be configured to assume that version 1.1 of the framework should be used. This can be configured differently for each "virtual folder", like the dyalog.net virtual folder which contains tutorials and examples distributed with Dyalog APL version 11.0. You can verify the ASP.NET setting by starting the IIS Control Panel, selecting Properties for a virtual folder, and navigating to the ASP.NET tab (see the picture below).



ASP.NET version: 2.0.50727

Virtual path: /dyalog.net

File location: C:\Program Files\Dyalog\Dyalog APL 11.0\sam

File creation date: 11-08-2006 12:19:18

File last modified: 11-08-2006 12:19:18

Note that it is not currently possibly for a single version of Dyalog APL to be used to compile and run web pages using different versions of the framework simultaneously. If you have web pages which need to continue to run under version 1.1, but wish to develop new ones under 2.0, you must either host the pages on different machines, or use version 10.1 of Dyalog APL to host the old web pages.

In theory, you could use version 1.1 for the script compiler in order to run web pages under 1.1 and the development environment under 2.0 in order to experiment with non-IIS functionality, but this is not recommended.

We hope to relax this restriction in a future version of Dyalog APL.

For more information, see
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconSide-by-SideExecution.asp

System Errors

Introduction

Dyalog APL will display a System Error Dialog and (normally) terminate in one of two circumstances:

1. As a result of the failure of a workspace integrity check
2. As a result of a System Exception

Workspace Integrity

When you `)SAVE` your workspace, Dyalog APL first performs a workspace integrity check. If it detects any discrepancy or violation in the internal structure of your workspace, APL does not overwrite your existing workspace on disk. Instead, it displays the System Error dialog box and saves the workspace, together with diagnostic information, in an **aplcore** file before terminating.

A System Error code is displayed in the dialog box and should be reported to Dyalog for diagnosis.

Note that the internal error that caused the discrepancy could have occurred at any time prior to the execution of `)SAVE` and it may not be possible for Dyalog to identify the cause from this **aplcore** file.

If APL is started in debug mode with the **–Dc**, **-Dw** or **–DW** flags, the Workspace Integrity check is performed more frequently, and it is more likely that the resulting aplcore file will contain information that will allow the problem to be identified and corrected.

System Exceptions

Non-specific System Errors are the result of Operating System exceptions that can occur due to a fault in Dyalog APL itself, an error in a Windows or other DLL, or even as a result of a hardware fault. The following system exceptions are separately identified.

Code	**Description**	**Suggested Action**
900	A Paging Fault has occurred	As the most likely cause is a temporary network fault, recommended course of action is to restart your program.
990 & 991	An exception has occurred in `dyalog11.dll` or `dyalog11rt.dll`	
995	An exception has occurred in a DLL function called via ⎕NA	Carefully check your ⎕NA statement and the arguments that you have passed to the DLL function
996	An exception has occurred in a DLL function called via a *threaded* ⎕NA call	As above
997	An exception has occurred while processing an incoming OLE call	
999	An exception has been caused by Dyalog APL or by the Operating System	

Recovering Data from aplcore files

Objects may often (but not always) be recovered from **aplcore** using `)COPY`. Note that because (by default) the aplcore file has no extension, it is necessary to explicitly add a "dot", or APL will attempt to find the non-existent file aplcore.DWS, i.e.

```
      )COPY aplcore.
```

Reporting Errors to Dyalog

If APL crashes and saves an aplcore file, please email the following information to support@dyalog.com:

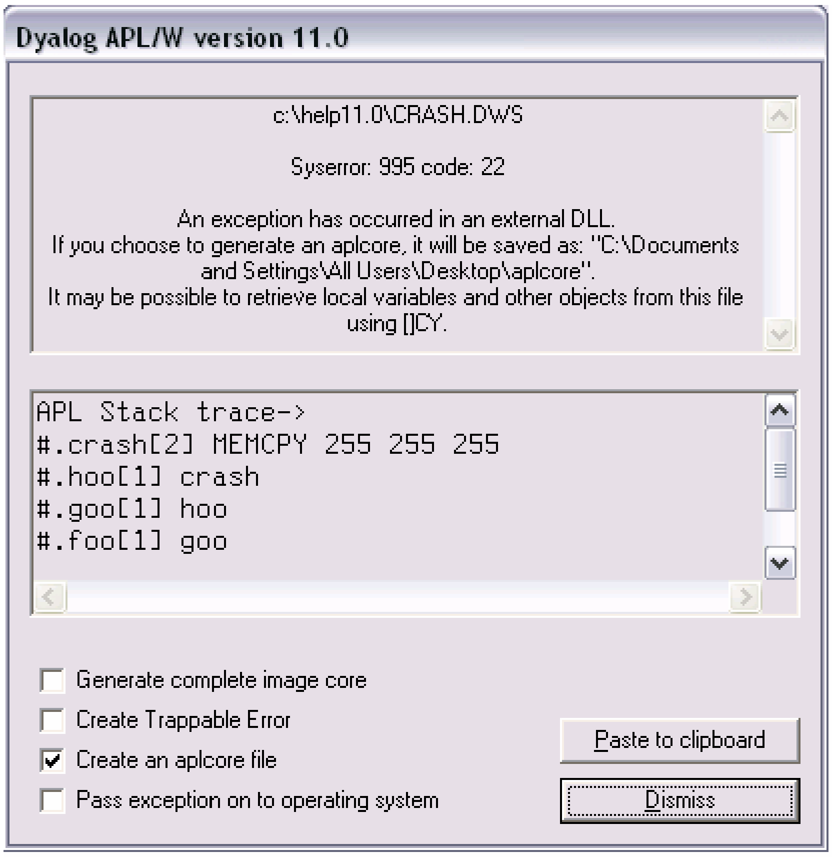
- a brief description of the circumstances surrounding the error
- your Dyalog APL Version number and Build ID (see Help/About)
- the aplcore file itself

If the problem is reproducible, i.e. can be easily repeated, please also send the appropriate description, workspace, and other files required to do so.

System Error Dialog Box

The System Error Dialog illustrated below was produced by deliberately inducing a system exception in the Windows DLL function `memcpy()`. The functions used were:

```
     ∇ foo
[1]    goo
     ∇
     ∇ goo
[1]    hoo
     ∇
     ∇ hoo
[1]    crash
     ∇

     ∇ crash
[1]    ⎕NA'dyalog32|MEMCPY u u u'
[2]    MEMCPY 255 255 255
     ∇
```



c:\help11.0\CRASH.DWS

Syserror: 995 code: 22

An exception has occurred in an external DLL.
If you choose to generate an aplcore, it will be saved as: "C:\Documents and Settings\All Users\Desktop\aplcore".
It may be possible to retrieve local variables and other objects from this file using []CY.

```
APL Stack trace->
#.crash[2] MEMCPY 255 255 255
#.hoo[1] crash
#.goo[1] hoo
#.foo[1] goo
```

[ ] Generate complete image core
[ ] Create Trappable Error
[✓] Create an aplcore file
[ ] Pass exception on to operating system

Options

Item	Description
Generate complete image core	Dumps a complete core image with the *User Mode Process Dumper* (a Microsoft tool) - see below.
Create Trappable Error	If you check this box (only enabled on System Error codes 995 and 996), APL will not terminate but will instead generate an error 91 (`EXTERNAL DLL EXCEPTION`) when you press *Dismiss*.
Create an aplcore file	If this box is checked, an aplcore file will be created.
Pass exception on to operating system	If this box is checked, the exception will be passed on to your current debugging tool (e.g. Visual Studio).
Paste to clipboard	Copies the contents of the APL stack trace window to the Clipboard.

Generate complete image core

The *Generate complete image core* option attempts to execute `[SYSDIR]\userdump.exe`, where `[SYSDIR]` is the windows system directory (typically `c:\windows\system32`, and `userdump.exe` is the User Mode Process Dumper, a Microsoft tool that can be downloaded from the following url (which you may copy from Winhelp and paste into a browser):

http://www.microsoft.com/downloads/details.aspx?FamilyID=e23cd741-d222-48df-9cd8-28796f414256&DisplayLang=en

The process creates a file called dyalog.core in the current directory. This file contains much more debug information than a normal aplcore (and is much larger than an aplcore) and can be sent to Dyalog Limited (zip it first please). Alternatively the file can be loaded into Visual Studio .Net to do your own debugging.

Debugging your own DLLs

If you are using Visual Studio on Microsoft Windows XP (or similar), the following procedure should be used to debug your own DLLs when an appropriate Dyalog APL System Error occurs.

Ensure that the *Pass Exception* box is checked, then click on *Dismiss* to close the System Error dialog box.

The system exception dialog box appears. Click on *Debug* to start the process in the Visual Studio debugger.

After debugging, the system exception dialog box appears again. Click on *Don't send* to terminate Microsoft Windows XP's exception handling.

ErrorOnExternalException Parameter

This parameter allows you to prevent APL from displaying the System Error dialog box (and terminating) when an exception caused by an external DLL occurs. The following example illustrates what happens when the functions above are run, but with ErrorOnExternalException set to 1.

```
      ⎕←2 ⎕NQ'.' 'GetEnvironment' 'ErrorOnExternalException'
1
      foo
EXTERNAL DLL EXCEPTION
crash[2] MEMCPY 255 255 255
          ^
      ⎕EN
91
      )SI
crash[2]*
hoo[1]
goo[1]
foo[1]
```

WorkspaceLoaded Event (525)

Applies to Session (⎕SE)

If enabled, this event is reported when a workspace is loaded or on a `clear ws`. You may not nullify or modify the event with a 0-returning callback, nor may you generate the event using ⎕NQ, or call it as a method.

The event message reported as the result of ⎕DQ, or supplied as the right argument to your callback function, is a 2-element vector as follows :

[1] Object name :	character vector ('⎕SE')
[2] Event name or code:	'WorkspaceLoaded' or 525

This event is fired immediately after a workspace has been loaded and before the execution of ⎕LX.

The callback function you attach should be defined in ⎕SE.

Miscellaneous

Miscellaneous changes introduced in Version 11.0 are as follows:

Keyboard Viewer

The Version 11 Session includes the *Kibitzer* keyboard viewer from Kai Jäger.
See Tools-> Keyboard Viewer ...

PassSingletonAsScalar

In Version 11.0, the default value of PassSingletonAsScalar is 0. In previous versions of Dyalog APL it was 1.

CHAPTER 2

Object Oriented Programing

Introducing Classes

A Class is a blueprint from which one or more *Instances* of the Class can be created (instances are sometimes also referred to as *Objects).*

A Class may optionally derive from another Class, which is referred to as its Base Class.

A Class may contain *Methods*, *Properties* and *Fields* (commonly referred to together as *Members*) which are defined within the body of the class script or are inherited from other Classes. This version of Dyalog APL does not support *Events* although it is intended that these will be supported in a future release. However, Classes that are derived from .Net types may generate events using `4 ⎕NQ`.

A Class that is defined to derive from another Class automatically acquires the set of Properties, Methods and Fields that are defined by its Base Class. This mechanism is described as inheritance.

A Class may extend the functionality of its Base Class by adding new Properties, Methods and Fields or by substituting those in the Base Class by providing new versions with the same names as those in the Base Class.

Members may be defined to be Private or Public. A Public member may be used or accessed from outside the Class or an Instance of the Class. A Private member is internal to the Class and (in general) may not be referenced from outside.

Although Classes are generally used as blueprints for the creation of instances, a class can have Shared members which can be used without first creating an instance

Defining Classes

A Class is defined by a script that may be entered and changed using the editor. A class script may also be constructed from a vector of character vectors, and fixed using `⎕FIX`.

A class script begins with a `:Class` statement and ends with a `:EndClass` statement.

For example, using the editor:

```
      )CLEAR
clear ws
      )ED ○Animal
```

[an edit window opens containing the following skeleton Class script ...]

```
:Class Animal
:EndClass
```

[the user edits and fixes the Class script]

```
      )CLASSES
Animal
      ⎕NC⊂'Animal'
9.4
```

Editing Classes

Between the `:Class` and `:EndClass` statements, you may insert any number of function bodies, Property definitions, and other elements. When you fix the Class Script from the editor, these items will be fixed inside the Class namespace.

Note that the contents of the Class Script defines the Class *in its entirety*. You may not add or alter functions by editing them independently and you may not add variables by assignment or remove objects with `⎕EX`.

When you *re-fix* a Class Script using the Editor or with `⎕FIX`, the original Class is discarded and the new definition, as specified by the Script, replaces the old one in its entirety.

Note:

Associated with a Class (or an instance of a class) there is a completely separate namespace which *surrounds* the class and can contain functions, variables and so forth that are created by actions external to the class.

For example, if `X` is *not* a public member of the class `MyClass`, then the following expression will insert a variable `X` into the namespace which surrounds the class:

```
      MyClass.X←99
```

The namespace is analogous to the namespace associated with a GUI object and will be re-initialised (emptied) whenever the Class is re-fixed. Objects in this parallel namespace are not visible from inside the Class or an Instance of the Class.

Inheritance

If you want a Class to derive from another Class, you simply add the name of that Class to the `:Class` statement using colon+space as a separator.

The following example specifies that `CLASS2` derives from `CLASS1`.

```
:Class CLASS2: CLASS1
:EndClass
```

Note that `CLASS1` is referred to as the *Base Class* of `CLASS2`.

If a Class has a Base Class, it automatically acquires all of the **Public** Properties, Methods and Fields defined for its Base Class unless it replaces them with its own members of the same name. This principle of inheritance applies throughout the Class hierarchy. Note that **Private** members are **not** subject to inheritance.

Warning: When a class is fixed, it keeps a reference (a pointer) to its base class. If the global name of the base class is expunged, the derived class will still have the base class reference, and the base class will therefore be kept *alive* in the workspace. The derived class will be fully functional, but attempts to edit it will fail when it attempts to locate the base class as the new definition is fixed.

At this point, if a new class with the original base class name is created, the derived class has no way of detecting this, and it will continue to use the *old and invisible* version of the base class. Only when the derived class is refixed, will the new base class be detected.

If you edit, refix or copy an existing base class, APL will take care to patch up the references, but if the base class is expunged first and recreated later, APL is unable to detect the substitution. You can recover from this situation by editing or refixing the derived class(es) after the base class has been substituted.

Classes that derive from .Net Types

You may define a Class that derives from any of the .Net Types by specifying the name of the .Net Type and including a `:USING` statement that provides a path to the .Net Assembly in which the .Net Type is located.

Example

```
:Class APLGreg: GregorianCalendar
:Using System.Globalization
...
:EndClass
```

Classes that derive from the Dyalog GUI

You may define a Class that derives from any of the Dyalog APL GUI objects by specifying the *name* of the Dyalog APL GUI Class in quotes.

For example, to define a Class named `Duck` that derives from a `Poly` object, the Class specification would be:

```
:Class Duck:'Poly'
:EndClass
```

The Base Constructor for such a Class is the `⎕WC` system function.

For further details see Writing Classes Based on the Dyalog GUI.

Instances

A Class is generally used as a blueprint or model from which one or more Instances of the Class are constructed. Note however that a class can have Shared members which can be used directly without first creating an instance.

You create an instance of a Class using the `⎕NEW` system function which is monadic.

The 1-or 2-item argument to `⎕NEW` contains a reference to the Class and, optionally, arguments for its Constructor function.

When `⎕NEW` executes, it first creates an empty instance namespace and tags it with an internal pointer to its Class.

When `⎕NEW` executes, it creates a regular APL namespace to contain the Instance, and within that it creates an Instance space, which is populated with any Instance Fields defined by the class (with default values if specified), and pointers to the Instance Method and Property definitions specified by the Class.

If a monadic Constructor is defined, it is called with the arguments specified in the second item of the argument to `⎕NEW`. If `⎕NEW` was called without Constructor arguments, and the class has a niladic Constructor, this is called instead.

The Constructor function is typically used to initialise the instance and may establish variables in the instance namespace.

The result of `⎕NEW` is a reference to the instance namespace. Instances of Classes exhibit the same set of Properties, Methods and Fields that are defined for the Class.

Constructors

A Constructor is a special function defined in the Class script that is to be run when an Instance of the Class is created by `⎕NEW`. Typically, the job of a Constructor is to initialise the new Instance in some way.

A Constructor is identified by a `:Implements Constructor` statement. This statement may appear anywhere in the body of the function after the function header. The significance of this is discussed below.

Note that it is also *essential* to define the Constructor to be *Public*, with a `:Access Public` statement, because like all Class members, Constructors default to being *Private*. Private Constructors currently have no use or purpose, but It is intended that they will be supported in a future release of Dyalog APL.

A Constructor function may be niladic or monadic and must not return a result.

A Class may specify any number of different Constructors of which one (and only one) may be niladic. This is also referred to as the *default* Constructor.

There may be any number of monadic Constructors, but each must have a differently defined argument list which specifies the number of items expected in the Constructor argument. See Constructor Overloading for details.

A Constructor function may not call another Constructor function and a constructor function may not be called directly from outside the Class. The only way a Constructor function may be invoked is by `⎕NEW`. See Base Constructors for further details.

When `⎕NEW` is executed *with a 2-item argument,* the appropriate monadic Constructor is called with the second item of the `⎕NEW` argument.

The niladic (default) Constructor is called when `⎕NEW` is executed with a 1-item argument, a Class reference alone, or whenever APL needs to create a fill item for the Class.

Note that `⎕NEW` first creates a new instance of the specified Class, and then executes the Constructor inside the instance.

Example

The `DomesticParrot` Class defines a Constructor function `egg` that initialises the Instance by storing its name (supplied as the 2nd item of the argument to `⎕NEW`) in a Public Field called `Name`.

```
:Class DomesticParrot:Parrot
    :Field Public Name

    ∇ egg name
      :Implements Constructor
      :Access Public
      Name←name
    ∇
    ...
:EndClass ⍝ DomesticParrot

      pol←⎕NEW DomesticParrot 'Polly'
      pol.Name
Polly
```

Constructor Overloading

NameList header syntax is used to define different versions of a Constructor each with a different number of parameters, referred to as its *signature*. The Clover Class illustrates this principle.

In deciding which Constructor to call, APL matches the shape of the Constructor argument with the signature of each of the Constructors that are defined. If a constructor with the same number of arguments exists (remembering that 0 arguments will match a niladic Constructor), it is called. If there is no exact match, and there is a Constructor with a general signature (an un-parenthesised right argument), it is called. If no suitable constructor is found, a `LENGTH ERROR` is reported.

There may be one and only one constructor with a particular signature.

A Constructor function may not call another Constructor function and a constructor function may not be called directly from outside the Class. The only way a Constructor function may be invoked is by `⎕NEW`. See Base Constructors for further details.

In the Clover Class example Class, the following Constructors are defined:

Constructor	**Implied argument**
`Make1`	1-item vector
`Make2`	2-item vector
`Make3`	3-item vector
`Make0`	No argument
`MakeAny`	Any array accepted

Clover Class Example

```
:Class Clover ⍝ Constructor Overload Example
    :Field Public Con
    ∇ Make0
      :Access Public
      :Implements Constructor
      make 0
    ∇
    ∇ Make1(arg)
      :Access Public
      :Implements Constructor
      make arg
    ∇
    ∇ Make2(arg1 arg2)
      :Access Public
      :Implements Constructor
      make arg1 arg2
    ∇
    ∇ Make3(arg1 arg2 arg3)
      :Access Public
      :Implements Constructor
      make arg1 arg2 arg3
    ∇
    ∇ MakeAny args
      :Access Public
      :Implements Constructor
      make args
    ∇
    ∇ make args
      Con←(⍴args)(2⊃⎕SI)args
    ∇
:EndClass ⍝ Clover
```

In the following examples, the `Make` function (see Clover Class listing for details) displays:

```
<shape of argument> <name of Constructor called><argument>
(see function Make)
```

Creating a new Instance of Clover with a 1-element vector as the Constructor argument, causes the system to choose the `Make1` Constructor.Note that, although the argument to `Make1` is a 1-element vector, this is disclosed as the list of arguments is unpacked into the (single) variable `arg1`.

```
      (⎕NEW Clover(,1)).Con
   Make1  1
```

Creating a new Instance of Clover with a 2- or 3-element vector as the Constructor argument causes the system to choose `Make2`, or `Make3` respectively.

```
      (⎕NEW Clover(1 2)).Con
 2  Make2  1 2
      (⎕NEW Clover(1 2 3)).Con
 3  Make3  1 2 3
```

Creating an Instance with any other Constructor argument causes the system to choose `MakeAny`.

```
      (⎕NEW Clover(⍳10)).Con
 10  MakeAny  1 2 3 4 5 6 7 8 9 10
      (⎕NEW Clover(2 2⍴⍳4)).Con
 2 2  MakeAny  1 2
               3 4
```

Note that a scalar argument will call `MakeAny` and not `Make1`.

```
      (⎕NEW Clover 1).Con
   MakeAny  1
```

and finally, creating an Instance without a Constructor argument causes the system to choose `Make0`.

```
      (⎕NEW Clover).Con
   Make0  0
```

Niladic (Default) Constructors

A Class may define a niladic Constructor and/or one or more Monadic Constructors. The niladic Constructor acts as the default Constructor that is used when `⎕NEW` is invoked without arguments and when APL needs a fill item.

```
:Class Bird
    :Field Public Species

    ∇ egg spec
      :Access Public Instance
      :Implements Constructor
      Species←spec
    ∇
    ∇ default
      :Access Public Instance
      :Implements Constructor
      Species←'Default Bird'
    ∇
    ∇ R←Speak
      :Access Public
      R←'Tweet, tweet!'
    ∇

:EndClass ⍝ Bird
```

The niladic Constructor (in this example, the function `default`) is invoked when `⎕NEW` is called without Constructor arguments. In this case, the Instance created is no different to one created by the monadic Constructor `egg`, except that the value of the `Species` Field is set to `'Default Bird'`.

```
      Birdy←⎕NEW Bird
      Birdy.Species
Default Bird
```

The niladic Constructor is also used when APL needs to make a fill item of the Class. For example, in the expression `(3↑Birdy)`, APL has to create two fill items of `Birdy` (one for each of the elements required to pad the array to length 3) and will in fact call the niladic Constructor twice.

In the following statement:

```
      TweetyPie←3⊃10↑Birdy
```

The `10↑` (temporarily) ceates a 10-element array comprising the single entity `Birdy` padded with 9 fill-elements of Class `Bird`. To obtain the 9 fill-elements, APL calls the niladic Constructor 9 times, one for each separate prototypical Instance that it is required to make.

```
      TweetyPie.Species
Default Bird
```

Empty Arrays of Instances: Why ?

In APL it is natural to use *arrays* of Instances. For example, consider the following example.

```
:Class Cheese
    :Field Public Name←''
    :Field Public Strength←⍬
    ∇ make2(name strength)
      :Access Public
      :Implements Constructor
      Name Strength←name strength
    ∇
    ∇ make1 name
      :Access Public
      :Implements Constructor
      Name Strength←name 1
    ∇
    ∇ make_excuse
      :Access Public
      :Implements Constructor
      ⎕←'The cat ate the last one!'
    ∇
:EndClass
```

We might create an array of Instances of the Cheese Class as follows:

```
      cdata←('Camembert' 5)('Caephilly' 2) 'Mild Cheddar'
      cheeses←{⎕NEW Cheese ⍵}¨cdata
```

Suppose we want a range of medium-strength cheese for our cheese board.

```
      board←(cheeses.Strength<3)/cheeses
      board.Name
 Caephilly  Mild Cheddar
```

But look what happens when we try to select really strong cheese:

```
      board←(cheeses.Strength>5)/cheeses
      board.Name
The cat ate the last one!
```

Note that this message is not the result of the expression, but was explicitly displayed by the `make_excuse` function. The clue to this behaviour is the shape of `board`; it is empty!

```
      ⍴board
0
```

When a reference is made to an empty array of Instances (strictly speaking, a reference that requires a *prototype*), APL creates a new Instance by calling the *niladic* (default) Constructor, uses the new Instance to satisfy the reference, and then discards it. Hence, in this example, the reference:

```
      board.Name
```

caused APL to run the *niladic* Constructor `make_excuse`, which displayed:

```
The cat ate the last one!
```

Notice that the behaviour of empty arrays of Instances is modelled VERY closely after the behaviour of empty arrays in general. In particular, the Class designer is given the task of deciding what the type of the members of the prototype are.

Empty Arrays of Instances: How?

To cater for the need to handle empty arrays of Instances as easily as non-empty arrays, a reference to an empty array of Class Instances is handled in a special way.

Whenever a reference or an assignment is made to the content of an *empty array of Instances*, the following steps are performed:

1. APL creates a *new Instance* of the same Class of which the empty Instance belongs.
2. the default (niladic) Constructor is run in the new Instance
3. the appropriate value is obtained or assigned:
    a. if it is a reference is to a Field, the value of the Field is obtained
    b. if it is a reference is to a Property, the PropertyGet function is run
    c. if it is a reference is to a Method, the method is executed
    d. if it is an assignment, the assignment is performed or the PropertySet function is run
4. if it is a reference, the result of step 3 is used to generate an empty result array with a suitable prototype by the application of the function `{0⍴⊂⍵}` to it
5. the Class Destructor (if any) is run in the new Instance
6. the New Instance is deleted

Example

```
:Class Bird
    :Field Public Species

    ∇ egg spec
      :Access Public Instance
      :Implements Constructor
      ⎕DF Species←spec
    ∇
    ∇ default
      :Access Public Instance
      :Implements Constructor
      ⎕DF Species←'Default Bird'
      #.DISPLAY Species
    ∇
    ∇ R←Speak
      :Access Public
      #.DISPLAY R←'Tweet, Tweet, Tweet'
    ∇

:EndClass ⍝ Bird
```

First, we can create an empty array of Instances of Bird using `0⍴`.

```
      Empty←0⍴⎕NEW Bird 'Robin'
```

A reference to `Empty.Species` causes APL to create a new Instance and invoke the niladic Constructor `default`. This function sets `Species` to `'Default Bird'`*and* calls `#.DISPLAY` which displays output to the Session.

```
      DISPLAY Empty.Species
.→-----------.
|Default Bird|
'------------'
```

APL then retrieves the value of `Species` (`'Default Bird'`), applies the function `{0⍴⊂⍵}` to it and returns this as the result of the expression.

```
.⊖---------------.
| .→-----------. |
| |            | |
| '------------' |
'∊---------------'
```

A reference to `Empty.Speak` causes APL to create a new Instance and invoke the niladic Constructor `default`. This function sets `Species` to `'Default Bird'`*and* calls `#.DISPLAY` which displays output to the Session.

```
      DISPLAY Empty.Speak
.→-----------.
|Default Bird|
'------------'
```

APL then involes function `Speak` which displays `'Tweet, Tweet, Tweet'` and returns this as the result of the function.

```
.→------------------.
|Tweet, Tweet, Tweet|
'-------------------'
```

APL then applies the function `{0⍴⊂⍵}` to it and returns this as the result of the expression.

```
.⊖----------------------.
| .→------------------. |
| |                   | |
| '-------------------' |
'∊----------------------'
```

Base Constructors

Constructors in a Class hierarchy, are not inherited in the same way as other members. However, there is a mechanism for all the Classes in the Class inheritance tree to participate in the initialisation of an Instance.

Every Constructor function contains a `:Implements Constructor` statement which may appear anywhere in the function body. The statement may optionally be followed by the `:Base` control word and an arbitrary expression.

The statement:

```
      :Implements Constructor :Base expr
```

calls *a monadic* Constructor in the Base Class. The choice of Constructor depends upon the rank and shape of the result of `expr` (see Constructor Overloading for details).

Whereas, the statement:

```
      :Implements Constructor
```
or
```
      :Implements Constructor :Base
```

calls *the niladic* Constructor in the Base Class.

Note that during the instantiation of an Instance, these calls potentially takes place in every Class in the Class hierarchy.

If, anywhere down the hierarchy, there is a *monadic* call and there is no matching monadic Constructor, the operation fails with a `LENGTH ERROR`.

If there is a *niladic* call on a Class that defines **no Constructors**, the niladic call is simply repeated in the next Class along the hierarchy.

However, if a Class defines a monadic Constructor and no niladic Constructor it implies that that Class **cannot be instantiated without Constructor arguments**. Therefore, if there is a call to a niladic Constructor in such a Class, the operation fails with a `LENGTH ERROR`. Note that it is therefore impossible for APL to instantiate a fill item or process a reference to an empty array for such a Class or any Class that is based upon it.

A Constructor function may not call another Constructor function and a constructor function may not be called directly from outside the Class or Instance. The only way a Constructor function may be invoked is by `⎕NEW`. The fundamental reason for these restrictions is that there must be one and only one call on the Base Constructor when a new Instance is instantiated. If Constructor functions were allowed to call one another, there would be several calls on the Base Constructor. Similarly, if a Constructor could be called directly it would potentially duplicate the Bse Constructor call.

Niladic Example

In the following example, `DomesticParrot` is derived from `Parrot` which is derived from `Bird`. They all share the Field `Desc` (inherited from `Bird`). Each of the 3 Classes has its own *niladic* Constructor called `egg0`.

```
:Class Bird
    :Field Public Desc
    ∇ egg0
      :Access Public
      :Implements Constructor
      Desc←'Bird'
    ∇
:EndClass ⍝ Bird

:Class Parrot: Bird
    ∇ egg0
      :Access Public
      :Implements Constructor
      Desc,←'→Parrot'
    ∇
:EndClass ⍝ Parrot

:Class DomesticParrot: Parrot
    ∇ egg0
      :Access Public
      :Implements Constructor
      Desc,←'→DomesticParrot'
    ∇
:EndClass ⍝ DomesticParrot

      (⎕NEW DomesticParrot).Desc
Bird→Parrot→DomesticParrot
```

Explanation

`⎕NEW` creates the new instance and runs the niladic Constructor `DomesticParrot.egg0`. As soon as the line:

```
:Implements Constructor
```

is encountered, `⎕NEW` calls the niladic constructor in the Base Class `Parrot.egg0`

`Parrot.egg0` starts to execute and as soon as the line:

```
:Implements Constructor
```

is encountered, `⎕NEW` calls the niladic constructor in the Base Class `Bird.egg0`.

When the line:

```
:Implements Constructor
```

is encountered, ⎕*NEW* cannot call the niladic constructor in the Base Class (there is none) so the chain of Constructors ends. Then, as the State Indicator unwinds …

`Bird.egg0`	executes	`Desc←'Bird''`
`Parrot.egg0`	executes	`Desc,←'→Parrot''`
`DomesticParrot.egg0`	execute	`Desc,←'→DomesticParrot''`

Monadic Example

In the following example, `DomesticParrot` is derived from `Parrot` which is derived from `Bird`. They all share the Field `Species` (inherited from `Bird`) but only a `DomesticParrot` has a Field `Name`. Each of the 3 Classes has its own Constructor called `egg`.

```
:Class Bird
    :Field Public Species
    ∇ egg spec
      :Access Public Instance
      :Implements Constructor
      Species←spec
    ∇
    ...
:EndClass ⍝ Bird

:Class Parrot: Bird
    ∇ egg species
      :Access Public Instance
      :Implements Constructor :Base 'Parrot: ',species
    ∇
    ...
:EndClass ⍝ Parrot

:Class DomesticParrot: Parrot
    :Field Public Name
    ∇ egg(name species)
      :Access Public Instance
      :Implements Constructor :Base species
      ⎕DF Name←name
    ∇
    ...
:EndClass ⍝ DomesticParrot

      pol←⎕NEW DomesticParrot('Polly' 'Scarlet Macaw')
      pol.Name
Polly
      pol.Species
Parrot: Scarlet Macaw
```

Explanation

⎕*NEW* creates the new instance and runs the Constructor `DomesticParrot.egg`. The `egg` header splits the argument into two items `name` and `species`. As soon as the line:

```
:Implements Constructor :Base species
```

is encountered, ⎕*NEW* calls the Base Class constructor `Parrot.egg`, passing it the result of the expression to the right, which in this case is simply the value in `species`.

`Parrot.egg` starts to execute and as soon as the line:

```
:Implements Constructor :Base 'Parrot: ',species
```

is encountered, ⎕*NEW* calls *its* Base Class constructor `Bird.egg`, passing it the result of the expression to the right, which in this case is the character vector `'Parrot: '` catenated with the value in `species`.

`Bird.egg` assigns its argument to the Public Field `Species`.

At this point, the State Indicator would be:

```
       )SI
[#.[Instance of DomesticParrot]] #.Bird.egg[3]*
[constructor]
:base
[#.[Instance of DomesticParrot]] #.Parrot.egg[2]
[constructor]
:base
[#.[Instance of DomesticParrot]] #.DomesticParrot.egg[2]
[constructor]
```

`Bird.egg` then returns to `Parrot.egg` which returns to `DomesticParrot.egg`.

Finally, `DomesticParrot.egg[3]` is executed, which establishes Field `Name` and the Display Format (⎕*DF*) for the instance.

Destructors

A *Destructor* is a function that is called just before an Instance of a Class ceases to exist and is typically used to close files or release external resources associated with an Instance.

An Instance of a Class is destroyed when:
- The Instance is expunged using `⎕EX` or `)ERASE`.
- A function, in which the Instance is localised, exits.

But be aware that a destructor will also be called if:
- The Instance is re-assigned (see below)
- The result of `⎕NEW` is not assigned (the instance gets created then immediately destroyed).
- APL creates (and then destroys) a new Instance as a result of a reference to a member of an empty Instance. The destructor is called after APL has obtained the appropriate value from the instance and no longer needs it.
- The constructor function fails. Note that the Instance is actually created before the constructor is run (inside it), and if the constructor fails, the fledgling Instance is discarded. Note too that this means a destructor *may* need to deal with a partially constructed instance, so the code may need to check that resources were actually acquired, before releasing them.
- On the execution of `)CLEAR`, `)LOAD`, `⎕LOAD` or `⎕OFF`.

Note that an Instance of a Class only disappears when the *last reference* to it disappears. For example, the sequence:

```
      I1←⎕NEW MyClass
      I2←I1
      )ERASE I1
```

will not cause the Instance of `MyClass` to disappear because it is still referenced by `I2`.

A Destructor is identified by the statement `:Implements Destructor` which must appear immediately after the function header in the Class script.

```
:Class Parrot
    ...
    ∇ kill
      :Implements Destructor
      'This Parrot is dead'
    ∇
    ...
:EndClass ⍝ Parrot

      pol←⎕NEW Parrot 'Scarlet Macaw'
      )ERASE pol
This Parrot is dead
```

Note that reassignment to `pol` causes the Instance referenced by `pol` to be destroyed and the Destructor invoked:

```
      pol←⎕NEW Parrot 'Scarlet Macaw'
      pol←⎕NEW Parrot 'Scarlet Macaw'
This Parrot is dead
```

If a Class inherits from another Class, the Destructor in its Base Class is automatically called after the Destructor in the Class itself.

So, if we have a Class structure:

```
      DomesticParrot => Parrot => Bird
```

containing the following Destructors:

```
:Class DomesticParrot: Parrot
    ...
    ∇ kill
      :Implements Destructor
      'This ',(⍕⎕THIS),' is dead'
    ∇
     ...
:EndClass ⍝ DomesticParrot

:Class Parrot: Bird
    ...
    ∇ kill
      :Implements Destructor
      'This Parrot is dead'
    ∇
    ...
:EndClass ⍝ Parrot

:Class Bird
    ...
    ∇ kill
      :Implements Destructor
      'This Bird is dead'
    ∇
    ...
:EndClass ⍝ Bird
```

Destroying an Instance of `DomesticParrot` will run the Destructors in `DomesticParrot`, `Parrot` and `Bird` and in that order.

```
      pol←⎕NEW DomesticParrot
      )CLEAR
This Polly is dead
This Parrot is dead
This Bird is dead
clear ws
```

Class Members

A Class may contain *Methods*, *Fields* and *Properties* (commonly referred to together as *Members*) which are defined within the body of the Class script or are inherited from other Classes.

Methods are regular APL defined functions, but with some special characteristics that control how they are called and where they are executed. D-fns may not be used as Methods.

Fields are just like APL variables. To get the Field value, you reference its name; to set the Field value, you assign to its name, and the Field value is stored *in* the Field. However, Fields differ from variables in that they possess characteristics that control their accessibility.

Properties are similar to APL variables. To get the Property value, you reference its name; to set the Property value, you assign to its name. However, Property values are actually accessed via *PropertyGet* and *PropertySet* functions that may perform all sorts of operations. In particular, the value of a Property is not stored *in* the Property and may be entirely dynamic.

All three types of member may be declared as *Public* or *Private* and as *Instance* or *Shared*.

Public members are visible from outside the Class and Instances of the Class, whereas Private members are only accessible from within.

Instance Members are unique to every Instance of the Class, whereas Shared Members are common to all Instances and Shared Members may be referenced directly on the Class itself.

Fields

A Field behaves just like an APL variable.

To get the value of a Field, you reference its name; to set the value of a Field, you assign to its name. Conceptually, the Field value is stored *in* the Field. However, Fields differ from variables in that they possess characteristics that control their accessibility.

A Field may be declared anywhere in a Class script by a `:Field` statement. This specifies:

- the name of the Field
- whether the Field is Public or Private
- whether the Field is Instance or Shared
- whether or not the Field is ReadOnly
- optionally, an initial value for the Field.

Note that Triggers may be associated with Fields. See Trigger Fields for details.

Public Fields

A *Public* Field may be accessed from outside an Instance or a Class. Note that the default is *Private*.

Class `DomesticParrot` has a `Name` Field which is defined to be Public and Instance (by default).

```
:Class DomesticParrot: Parrot
    :Field Public Name

    ∇ egg nm
      :Access Public
      :Implements Constructor
      Name←nm
    ∇
    ...
:EndClass ⍝ DomesticParrot
```

The Name field is initialised by the Class constructor.

```
      pet←⎕NEW DomesticParrot'Polly'
      pet.Name
Polly
```

The Name field may also be modified directly:

```
      pet.Name←⌽pet.Name
      pet.Name
ylloP
```

Initialising Fields

A Field may be assigned an initial value. This can be specified by an arbitrary expression that is executed when the Class is fixed by the Editor or by `⎕FIX`.

```
:Class DomesticParrot: Parrot
    :Field Public Name←'Dicky'
        :Field Public Talks←1

    ∇ egg nm
      :Access Public
      :Implements Constructor
      Name←nm
    ∇
    ...
:EndClass ⍝ DomesticParrot
```

Field `Talks` will be initialised to `1` in every instance of the Class.

```
      pet←⎕NEW DomesticParrot 'Dicky'

      pet.Talks
1
      pet.Name
Dicky
```

Note that if a Field is ReadOnly, this is the only way that it may be assigned a value.

See also: Shared Fields

Private Fields

A Private Field may only be referenced by code running inside the Class or an Instance of the Class. Furthermore, Private Fields are not inherited.

The ComponentFile Class (see page 69) has a Private Instance Field named `tie` that is used to store the file tie number in each Instance of the Class.

```
:Class ComponentFile
    :Field Private Instance tie

    ∇ Open filename
      :Implements Constructor
      :Access Public Instance
      :Trap 0
          tie←filename ⎕FTIE 0
      :Else
          tie←filename ⎕FCREATE 0
      :EndTrap
      ⎕DF filename,'(Component File)'
    ∇
```

As the field is declared to be Private, it is not accessible from outside an Instance of the Class, but is only visible to code running inside.

```
      F1←⎕NEW ComponentFile 'test1'
      F1.tie
VALUE ERROR
      F1.tie
     ^
```

Shared Fields

If a Field is declared to be *Shared*, it has the same value for every Instance of the Class. Moreover, the Field may be accessed from the Class itself; an Instance is not required.

The following example establishes a Shared Field called `Months` that contains abbreviated month names which are appropriate for the user's current International settings. It also shows that an arbitrarily complex statement may be used to initialise a Field.

```
:Class Example
    :Using System.Globalization
    :Field Public Shared ReadOnly Months←12↑(⎕NEW DateTime
FormatInfo).AbbreviatedMonthNames
:EndClass ⍝ Example
```

A Shared Field is not only accessible from an instance …

```
      EG←⎕NEW Example
      EG.Months
 Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov...
```

… but also, directly from the Class itself.

```
      Example.Months
Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov...
```

Notice that in this case it is necessary to insert a `:Using` statement (or the equivalent assignment to `⎕USING`) in order to specify the .Net search path for the DateTimeFormatInfo type. Without this, the Class would fail to fix.

You can see how the assignment works by executing the same statements in the Session:

```
      ⎕USING←'System.Globalization'
    12↑(⎕NEW DateTimeFormatInfo).AbbreviatedMonthNames
 Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov...
```

Trigger Fields

A Fields may act as a Trigger so that a function may be invoked whenever the value of the Field is changed.

As an example, it is often useful for the Display Form of an Instance to reflect the value of a certain Field. Naturally, when the Field changes, it is desirable to change the Display Form. This can be achieved by making the Field a Trigger as illustrated by the following example.

Notice that the Trigger function is invoked both by assignments made within the Class (as in the assignment in `ctor`) and those made from outside the Instance.

```
:Class MyClass
    :Field Public Name
    :Field Public Country←'England'
    ∇ ctor nm
      :Access Public
      :Implements Constructor
      Name←nm
    ∇
    ∇ format
      :Implements Trigger Name,Country
      ⎕DF'My name is ',Name,' and I live in ',Country
    ∇
:EndClass ⍝ MyClass

      me←⎕NEW MyClass 'Pete'
      me
My name is Pete and I live in England

      me.Country←'Greece'
      me
My name is Pete and I live in Greece

      me.Name←'Kostas'
      me
My name is Kostas and I live in Greece
```

Methods

Methods are implemented as regular defined functions, but with some special attributes that control how they are called and where they are executed.

A Method is defined by a contiguous block of statements in a Class Script. A Method begins with a line that contains a `∇`, followed by a valid APL defined function header. The method definition is terminated by a closing `∇`.

The behaviour of a Method is defined by an `:Access` control statement.

Public or Private

Methods may be defined to be Private (the default) or Public.

A Private method may only be invoked by another function that is running inside the Class namespace or inside an Instance namespace. The name of a Private method is not visible from outside the Class or an Instance of the Class.

A Public method may be called from outside the Class or an Instance of the Class.

Instance or Shared

Methods may be defined to be Instance (the default) or Shared.

An Instance method runs in the Instance namespace and may only be called via the instance itself. An Instance method has direct access to Fields and Properties, both Private and Public, in the Instance in which it runs.

A Shared method runs in the Class namespace and may be called via an Instance or via the Class. However, a Shared method that is called via an Instance does not have direct access to the Fields and Properties of that Instance.

Shared methods are typically used to manipulate Shared Properties and Fields or to provide general services for all Instances that are not Instance specific.

Overridable Methods

Instance Methods may be declared with `:Access Overridable`.

A Method declared as being Overridable is replaced in situ (i.e.within its own Class) by a Method of the same name that is defined in a higher Class which itself is declared with the Override keyword. See *Superseding Base Class Methods*.

Shared Methods

A Shared method runs in the Class namespace and may be called via an Instance or via the Class. However, a Shared method that is called via an Instance does not have direct access to the Fields and Properties of that Instance.

Class `Parrot` has a `Speak` method that does not require any information about the current Instance, so may be declared as Shared.

```
:Class Parrot:Bird

    ∇ R←Speak times
      :Access Public Shared
      R←⍕timesρ⊂'Squark!'
    ∇

:EndClass ⍝ Parrot

      wild←⎕NEW Parrot
      wild.Speak 2
 Squark!  Squark!
```

Note that `Parrot.Speak` may be executed directly from the Class and does not in fact require an Instance.

```
      Parrot.Speak 3
 Squark!  Squark!  Squark!
```

Instance Methods

An Instance method runs in the Instance namespace and may only be called via the instance itself. An Instance method has direct access to Fields and Properties, both Private and Public, in the Instance in which it runs.

Class `DomesticParrot` has a `Speak` method defined to be Public and Instance. Where `Speak` refers to `Name`, it obtains the value of `Name` in the current Instance.

Note too that `DomesticParrot.Speak` supersedes the inherited `Parrot.Speak`.

```
:Class DomesticParrot: Parrot
    :Field Public Name

    ∇ egg nm
      :Access Public
      :Implements Constructor
      Name←nm
    ∇

    ∇ R←Speak times
      :Access Public Instance
      R←⊂Name,', ',Name
      R←↑R,times⍴⊂' Who's a pretty boy,then!'
    ∇

:EndClass ⍝ DomesticParrot

      pet←⎕NEW DomesticParrot'Polly'
      pet.Speak  3
Polly, Polly
 Who's a pretty boy,then!
 Who's a pretty boy,then!
 Who's a pretty boy,then!

      bil←⎕NEW  DomesticParrot'Billy'
      bil.Speak  1
Billy, Billy
 Who's a pretty boy,then!
```

Superseding Base Class Methods

Normally, a Method defined in a higher Class supersedes the Method of the same name that is defined in its Base Class, but only for calls made from above or within the higher Class itself (or an Instance of the higher Class). The base method remains available *in the Base Class* and is invoked by a reference to it *from within the Base Class*. This behaviour can be altered using the Overridable and Override key words in the `:Access` statement but only applies to Instance Methods.

If a Public Instance method in a Class is marked as *Overridable*, this allows a Class which derives from the Class with the Overridable method to supersede the Base Class method **in the Base Class**, by providing a method which is marked *Override*. The typical use of this is to replace code in the Base Class which handles an event, with a method provided by the derived Class.

For example, the base class might have a method which is called if any error occurs in the base class:

```
     ∇ ErrorHandler
[1]    :Access Public Overridable
[2]    ⎕←↑⎕DM
     ∇
```

In your derived class, you might supersede this by a more sophisticated error handler, which logs the error to a file:

```
     ∇ ErrorHandler;TN
[1]    :Access Public Override
[2]    ⎕←↑⎕DM
[3]    TN←'ErrorLog'⎕FSTIE 0
[4]    ⎕DM ⎕FAPPEND TN
[5]    ⎕FUNTIE TN
     ∇
```

If the derived class had a function which was not marked Override, then function in the derived class which called `ErrorHandler` would call the function as defined in the derived class, but if a function in the base class called `ErrorHandler`, it would still see the base class version of this function. With Override specified, the new function supersedes the function as seen by code in the base class. Note that different derived classes can specify different Overrides.

In C#, Java and some other compiled languages, the term *Virtual* is used in place of *Overridable*, which is the term used by Visual Basic and Dyalog APL.

Properties

A Property behaves in a very similar way to an ordinary APL variable. To obtain the value of a Property, you simply reference its name. To change the value of a Property, you assign a new value to the name.

However, *under the covers*, a Property is accessed via a *PropertyGet* function and its value is changed via a *PropertySet* function. Furthermore, Properties may be defined to allow partial (indexed) retrieval and assignment to occur.

There are three types of Property, namely *Simple*, *Numbered* and *Keyed*.

A *Simple Property* is one whose value is accessed (by APL) in its entirety and re-assigned (by APL) in its entirety.

A *Numbered Property* behaves like an array (conceptually a vector) which is only ever *partially* accessed and set (one element at a time) via indices. The Numbered Property is designed to allow APL to perform selections and structural operations on the Property.

A *Keyed Property* is similar to a Numbered Property except that its elements are accessed via arbitrary keys instead of indices.

The following cases illustrate the difference between Simple and Numbered Properties.

If Instance `MyInst` has a Simple Property `Sprop` and a Numbered Property `Nprop`, the expressions

```
      X←MyInst.SProp
      X←MyInst.SProp[2]
```

both cause APL to call the PropertyGet function to retrieve the entire value of `Sprop`. The second statement subsequently uses indexing to extract just the second element of the value.

Whereas, the expression:

```
      X←MyInst.NProp[2]
```

causes APL to call the PropertyGet function with an additional argument which specifies that only the second element of the Property is required. Moreover, the expression:

```
      X←MyInst.NProp
```

causes APL to call the PropertyGet function successively, for every element of the Property.

A Property is defined by a `:Property ... :EndProperty` section in a Class Script.

Within the body of a Property Section there may be:

- one or more `:Access` statements
- a single PropertyGet function.
- a single PropertySet function
- a single PropertyShape function

Simple Instance Properties

A Simple Instance Property is one whose value is accessed (by APL) in its entirety and re-assigned (by APL) in its entirety. The following examples are taken from the ComponentFile Class (see page 69).

The Simple Property `Count` returns the number of components on a file.

```
     :Property Count
     :Access Public Instance
         ∇ r←get
           r←¯1+2⊃⎕FSIZE tie
         ∇
     :EndProperty ⍝ Count

       F1←⎕NEW ComponentFile 'test1'
       F1.Append'Hello World'
1
       F1.Count
1
       F1.Append 42
2
       F1.Count
2
```

Because there is no `set` function defined, the Property is read-only and attempting to change it causes `SYNTAX ERROR`.

```
       F1.Count←99
SYNTAX ERROR
       F1.Count←99
      ^
```

The `Access` Property has both `get` and `set` functions which are used, in this simple example, to get and set the component file access matrix.

```
    :Property Access
    :Access Public Instance
        ∇ r←get
          r←⎕FRDAC tie
        ∇
        ∇ set am;mat;OK
          mat←am.NewValue
          :Trap 0
              OK←(2=⍴⍴mat)∧(3=2⊃⍴mat)∧∧/,mat=⌊mat
          :Else
              OK←0
          :EndTrap
          'bad arg'⎕SIGNAL(~OK)/11
          mat ⎕FSTAC tie
        ∇
    :EndProperty ⍝ Access
```

Note that the `set` function **must** be monadic. Its argument, supplied by APL, will be an Instance of `PropertyArguments`. This is an internal Class whose `NewValue` field contains the value that was assigned to the Property.

Note that the set function does not have to accept the new value that has been assigned. The function may validate the value reject or accept it (as in this example), or perform whatever processing is appropriate.

```
      F1←⎕NEW ComponentFile 'test1'
      ⍴F1.Access
0 3
        F1.Access←3 3⍴28 2105 16385 0 2073 16385 31 ¯1 0
      F1.Access
28 2105 16385
 0 2073 16385
31   ¯1     0

      F1.Access←'junk'
bad arg
      F1.Access←'junk'
     ∧

      F1.Access←1 2⍴10
bad arg
      F1.Access←1 2⍴10
     ∧
```

Simple Shared Properties

The ComponentFile Class (see page 69) specifies a Simple Shared Property named `Files` which returns the names of all the Component Files in the current directory.

The previous examples have illustrated the use of Instance Properties. It is also possible to define *Shared* properties.

A Shared property may be used to handle information that is relevant to the Class as a whole, and which is not specific to any a particular Instance.

```
    :Property Files
    :Access Public Shared
        ∇ r←get
          r←⎕FLIB''
        ∇
    :EndProperty
```

Note that `⎕FLIB` (invoked by the `Files get` function) does not report the names of *tied* files.

```
      F1←⎕NEW ComponentFile 'test1'
      ⎕EX'F1'
      F2←⎕NEW ComponentFile 'test2'
      F2.Files ⍝ NB ⎕FLIB does not report tied files
test1
      ⎕EX'F2'
```

Note that a Shared Property may be accessed from the Class itself. It is not necessary to create an Instance first.

```
      ComponentFile.Files
test1
test2
```

Numbered Properties

A Numbered Property behaves like an array which is only ever *partially* accessed and set (one element at a time) via indices.

To implement a Numbered Property, you **must** specify a PropertyShape function and either or both a PropertyGet and PropertySet function.

When an expression references or makes an assignment to a Numbered Property, APL first calls its PropertyShape function which returns the dimensions of the Property. Note that the shape of the result of this function determines the *rank* of the Property except that a scalar result implies a vector.

APL then calls the PropertyGet or PropertySet function once for each element of the index set, supplying an argument of type PropertyArguments.

Note that when a numbered property is accessed, APL is responsible for validating the indices and ensuring that the value assigned or retrieved is a scalar. When PropertySet is called, NewValue will always be a scalar, and APL will validate that you return a scalar as the result of a PropertyGet.

If the expression references or assigns the entire Property (without indexing) APL generates a set of indices for every element of the Property and calls the PropertyGet or PropertySet function successively for every element in the Property. Future versions of APL may provide ways to specify that numbered accessor functions work one more than one element at a time.

Note that APL generates a `RANK ERROR` if an index contains the wrong number of elements or an `INDEX ERROR` if an index is out of bounds.

Example

The ComponentFile Class (see page 69) specifies a Numbered Property named `Component` which represents the contents of a specified component on the file.

```
:Property Numbered Component
:Access Public Instance
    ∇ r←shape
      r←¯1+2⊃⎕FSIZE tie
    ∇
    ∇ r←get arg
      r←⊂⎕FREAD tie arg.Indexers
    ∇
    ∇ set arg
      (⊃arg.NewValue) ⎕FREPLACE tie,arg.Indexers
    ∇
:EndProperty
```

```
      F1←⎕NEW ComponentFile 'test1'

      F1.Append¨(⍳5)×⊂⍳4
1 2 3 4 5

      F1.Count
5

      F1.Component[4]
 4 8 12 16

      4⊃F1.Component
4 8 12 16

      (⊂4 3)⌷F1.Component
 4 8 12 16  3 6 9 12
```

Referencing a Numbered Property in its entirety causes APL to call the `get` function successively for every element.

```
      F1.Component
 1 2 3 4  2 4 6 8  3 6 9 12  4 8 12 16  5 10 15 20

      ((⊂4 3)⌷F1.Component)←'Hello' 'World'

      F1.Component[3]
 World
```

Attempting to access a Numbered Property with inappropriate indices generates an error:

```
      F1.Component[6]
INDEX ERROR
      F1.Component[6]
     ^
      F1.Component[1;2]
RANK ERROR
      F1.Component[1;2]
     ^
```

The Default Property

A single Numbered Property may be identified as the *Default* Property for the Class. If a Class has a Default Property, indexing with the `⌷` primitive function and `[...]` indexing may be applied to the Property directly via a reference to the Class or Instance.

The Numbered Property example of the ComponentFile Class (see page 69) can be extended by adding the control word `Default` to the `:Property` statement for the `Component` Property.

Indexing may now be applied directly to the Instance `F1`. In essence, `F1[n]` is simply shorthand for `F1.Component[n]` and `n⌷F1` is shorthand for `n⌷F1.Component`

```
    :Property Numbered Default Component
    :Access Public Instance
        ∇ r←shape
          r←¯1+2⊃⎕FSIZE tie
        ∇
        ∇ r←get arg
          r←⊂⎕FREAD tie arg.Indexers
        ∇
        ∇ set arg
          (⊃arg.NewValue) ⎕FREPLACE tie,arg.Indexers
        ∇
    :EndProperty
```

```
      F1←⎕NEW ComponentFile 'test1'
      F1.Append¨(⍳5)×⊂⍳4
1 2 3 4 5
      F1.Count
5

      F1[4]
 4 8 12 16
      (⊂4 3)⌷F1
 4 8 12 16  3 6 9 12
      ((⊂4 3)⌷F1)←'Hello' 'World'
      F1[3]
 World
```

Note however that this feature applies only to indexing.

```
      4⊃F1
DOMAIN ERROR
      4⊃F1
     ^
```

Component File Class Example

```
:Class ComponentFile
    :Field Private Instance tie

    ∇ Open filename
      :Implements Constructor
      :Access Public Instance
      :Trap 0
          tie←filename ⎕FTIE 0
      :Else
          tie←filename ⎕FCREATE 0
      :EndTrap
      ⎕DF filename,'(Component File)'
    ∇

    ∇ Close
      :Access Public Instance
      ⎕FUNTIE tie
    ∇

    ∇ r←Append data
      :Access Public Instance
      r←data ⎕FAPPEND tie
    ∇

    ∇ Replace(comp data)
      :Access Public Instance
      data ⎕FREPLACE tie,comp
    ∇

:Property Count
    :Access Public Instance
        ∇ r←get
          r←¯1+2⊃⎕FSIZE tie
        ∇
    :EndProperty ⍝ Count
```

Component File Class Example (continued)

```
:Property Access
    :Access Public Instance
        ∇ r←get arg
          r←⎕FRDAC tie
        ∇
        ∇ set am;mat;OK
          mat←am.NewValue
          :Trap 0
              OK←(2=⍴⍴mat)∧(3=2⊃⍴mat)∧∧/,mat=⌊mat
          :Else
              OK←0
          :EndTrap
          'bad arg'⎕SIGNAL(~OK)/11
          mat ⎕FSTAC tie
        ∇
    :EndProperty ⍝ Access

    :Property Files
    :Access Public Shared
        ∇ r←get
          r←⎕FLIB''
        ∇
    :EndProperty

    :Property Numbered Default Component
    :Access Public Instance
        ∇ r←shape args
          r←¯1+2⊃⎕FSIZE tie
        ∇
        ∇ r←get arg
          r←⊂⎕FREAD tie,arg.Indexers
        ∇
        ∇ set arg
          (⊃arg.NewValue) ⎕FREPLACE tie,arg.Indexers
        ∇
    :EndProperty

    ∇ Delete file;tie
      :Access Public Shared
      tie←file ⎕FTIE 0
      file ⎕FERASE tie
    ∇
:EndClass ⍝ Class ComponentFile
```

Keyed Properties

A Keyed Property is similar to a Numbered Property except that it may **only** be accessed by indexing (so-called square-bracket indexing) and indices are not restricted to integers but may be arbitrary arrays.

To implement a Keyed Property, only a `get` and/or a `set` function are required. APL does not attempt to validate or resolve the specified indices in any way, so does not require the presence of a `shape` function for the Property.

However, APL **does** check that the rank and lengths of the indices correspond to the rank and lengths of the array to the right of the assignment (for an indexed assignment) and the array returned by the get function (for an indexed reference). If the rank or shape of these arrays fails to conform to the rank or shape of the indices, APL will issue a `RANK ERROR` or `LENGTH ERROR`.

Note too that indices **may not be elided**. If `KProp` is a Keyed Property of Instance `I1`, the following expressions would all generate `NONCE ERROR`.

```
      I1.KProp
      I1.KProp[]←10
      I1.KProp[;]←10
      I1.KProp['One' 'Two';]←10
      I1.KProp[;'One' 'Two']←10
```

When APL calls a monadic `get` or a `set` function, it supplies an argument of type PropertyArguments.

The Sparse2 Class illustrates the implementation and use of a Keyed Property.

`Sparse2` represents a 2-dimensional sparse array each of whose dimensions are indexed by arbitrary character keys. The sparse array is implemented as a Keyed Property named `Values`. The following expressions show how it might be used.

```
      SA1←⎕NEW Sparse2
      SA1.Values[⊂'Widgets';⊂'Jan']←100
      SA1.Values[⊂'Widgets';⊂'Jan']
100
      SA1.Values['Widgets' 'Grommets';'Jan' 'Mar' 'Oct']←1
0×2 3⍴⍳6
      SA1.Values['Widgets' 'Grommets';'Jan' 'Mar' 'Oct']
10 20 30
40 50 60
      SA1.Values[⊂'Widgets';'Jan' 'Oct']
10 30
      SA1.Values['Grommets' 'Widgets';⊂'Oct']
60
30
```

Sparse2 Class Example

```
:Class Sparse2  ⍝ 2D Sparse Array
    :Field Private keys
    :Field Private values
    ∇ make
      :Access Public
      :Implements Constructor
      keys←0⍴⊂'' ''
      values←⍬
    ∇
    :Property Keyed Values
    :Access Public Instance
        ∇ v←get arg;k
          k←arg.Indexers
          ⎕SIGNAL(2≠⍴k)/4
          k←fixkeys k
          v←(values,0)[keys⍳k]
        ∇
        ∇ set arg;new;k;v;n
          v←arg.NewValue
          k←arg.Indexers
          ⎕SIGNAL(2≠⍴k)/4
          k←fixkeys k
          v←(⍴k)(⍴⍣(⊃1=⍴,v))v
          ⎕SIGNAL((⍴k)≠⍴v)/5
          k v←,¨k v
          :If ∨/new←~k∊keys
              values,←new/v
              keys,←new/k
              k v/⍨←⊂~new
          :EndIf
          :If 0<⍴k
              values[keys⍳k]←v
          :EndIf
        ∇
    :EndProperty

    ∇ k←fixkeys k
      k←(2≠≡¨k){,(⊂⍣⍺)⍵}¨k
      k←⊃(∘.{⊃,/⊂¨⍺ ⍵})/k
    ∇
:EndClass ⍝ 2D Sparse Array
```

Internally, `Sparse2` maintains a list of keys and a list of values which are initialised to empty arrays by its constructor.

When an indexed assignment is made, the `set` function receives a list of keys (indices) in `arg.Indexer` and values in `arg.NewValue`. The function updates the values of existing keys, and adds new keys and their values to the internal lists.

When an indexed reference is made, the `get` function receives a list of keys (indices) in `arg.Indexer`. The function uses these keys to retrieve the corresponding values, inserting 0s for non-existent keys.

Note that in the expression:

```
SA1.Values['Widgets' 'Grommets';'Jan' 'Mar' 'Oct']
```

the structure of `arg.Indexer` is:

```
.→-----------------------------------------------.
| .→---------------------. .→------------------. |
| | .→------. .→-------. | | .→--. .→--. .→--. | |
| | |Widgets| |Grommets| | | |Jan| |Mar| |Oct| | |
| | '-------' '--------' | | '---' '---' '---' | |
| '∊---------------------' '∊------------------' |
'∊-----------------------------------------------'
```

Example

A second example of a Keyed Property is provided by the `KeyedFile` Class which is based upon the ComponentFile Class (see page 69) used previously.

```
:Class KeyedFile: ComponentFile
    :Field Public Keys
    eis←{,(⊂⍣(1=≡⍵))⍵}

    ∇ Open filename
      :Implements Constructor :Base filename
      :Access Public Instance
      :If Count>0
          Keys←{⊃⍵⊃⎕BASE.Component}¨⍳Count
      :Else
          Keys←0⍴⊂''
      :EndIf
    ∇

    :Property Keyed Component
    :Access Public Instance
        ∇ r←get arg;keys;sink
          keys←eis⊃arg.Indexers
          ⎕SIGNAL(~∧/keys∊Keys)/3
          r←{2⊃⍵⊃⎕BASE.Component}¨Keys⍳keys
        ∇
        ∇ set arg;new;keys;vals
          vals←arg.NewValue
          keys←eis⊃arg.Indexers
          ⎕SIGNAL((⍴keys)≠⍴vals)/5
          :If ∨/new←~keys∊Keys
              sink←Append¨↓⍉↑(⊂new)/¨keys vals
              Keys,←new/keys
              keys vals/⍨←⊂~new
          :EndIf
          :If 0<⍴keys
              Replace¨↓⍉↑(Keys⍳keys)(↓⍉↑keys vals)
          :EndIf
        ∇
    :EndProperty

:EndClass ⍝ Class KeyedFile

      K1←⎕NEW KeyedFile 'ktest'
      K1.Count
0
      K1.Component[⊂'Pete']←42
      K1.Count
1
      K1.Component['John' 'Geoff']←(⍳10)(3 4⍴⍳12)
      K1.Count
3
```

```
      K1.Component['Geoff' 'Pete']
 1  2  3  4  42
 5  6  7  8
 9 10 11 12
      K1.Component['Pete' 'Morten']←(3 4⍴'∘')(⍳⍳3)
      K1.Count
4
      K1.Component['Morten' 'Pete' 'John']
  1 1 1  1 1 2  1 1 3   ∘∘∘∘  1 2 3 4 5 6 7 8 9 10
  1 2 1  1 2 2  1 2 3   ∘∘∘∘
                        ∘∘∘∘
```

Interfaces

An Interface is defined by a Script that contains skeleton declarations of Properties and/or Methods. These members are only *place-holders*; they have no specific implementation; this is provided by each of the the Classes that support the Interface.

An Interface contains a collection of methods and properties that together represents a *protocol* that an application must follow in order to manipulate a Class in a particular way.

An example might be an Interface called Icompare that provides a single method (Compare) which compares two Instances of a Class, returning a value to indicate which of the two is greater than the other. A Class that implements Icompare must provide an appropriate Compare method, but every Class will have its own individual version of Compare. An application can then be written that sorts Instances of any Class that supports the ICompare Interface.

An Interface is implemented by a Class if it includes the name of the Interface in its :Class statement, and defines a corresponding set of the Methods and Properties that are declared in the Interface.

To implement a Method, a function defined in the Class must include a `:Implements Method` statement that maps it to the corresponding Method defined in the Interface:

```
    :Implements Method <InterfaceName.MethodName>
```

Furthermore, the syntax of the function (whether it be result returning, monadic or niladic) must exactly match that of the method described in the Interface. The function name, however, need not be the same as that described in the Interface.

Similarly, to implement a Property the type (Simple, Numbered or Keyed) and syntax (defined by the presence or absence of a PropertyGet and PropertySet functions) must exactly match that of the property described in the Interface. The Property name, however, need not be the same as that described in the Interface.

Example

The Penguin Class example illustrates the use of Interfaces to implement *multiple inheritance.*

```
:Interface FishBehaviour
∇ R←Swim ⍝ Returns description of swimming capability
  :Access Public
∇
:EndInterface ⍝ FishBehaviour

:Interface BirdBehaviour
∇ R←Fly ⍝ Returns description of flying capability
  :Access Public
∇
∇ R←Lay ⍝ Returns description of egg-laying behaviour
  :Access Public
∇
∇ R←Sing ⍝ Returns description of bird-song
  :Access Public
∇
:EndInterface ⍝ BirdBehaviour

:Class Penguin: Animal,BirdBehaviour,FishBehaviour
    ∇ R←NoCanFly
      :Implements Method BirdBehaviour.Fly
      R←'Although I am a bird, I cannot fly'
    ∇
    ∇ R←LayOneEgg
      :Implements Method BirdBehaviour.Lay
      R←'I lay one egg every year'
    ∇
    ∇ R←Croak
      :Implements Method BirdBehaviour.Sing
      R←'Croak, Croak!'
    ∇
    ∇ R←Dive
      :Implements Method FishBehaviour.Swim
      R←'I can dive and swim like a fish'
    ∇
:EndClass ⍝ Penguin
```

In this case, the `Penguin` Class derives from `Animal` but additionally supports the `BirdBehaviour` and `FishBehaviour` Interfaces, thereby inheriting members from both.

```
      Pingo←⎕NEW Penguin
      ⎕CLASS Pingo
  #.Penguin  #.FishBehaviour  #.BirdBehaviour    #.Animal

      (FishBehaviour ⎕CLASS Pingo).Swim
I can dive and swim like a fish
      (BirdBehaviour ⎕CLASS Pingo).Fly
Although I am a bird, I cannot fly
      (BirdBehaviour ⎕CLASS Pingo).Lay
I lay one egg every year
      (BirdBehaviour ⎕CLASS Pingo).Sing
Croak, Croak!
```

Including Namespaces

A Class may import methods from one or more plain Namespaces. This allows several Classes to share a common set of methods, and provides a degree of multiple inheritance.

To import methods from a Namespace `NS`, the Class Script must include a statement:

```
:Include NS
```

When the Class is fixed by the editor or by `⎕FIX`, all the defined functions and operators in Namespace `NS` are included as methods in the Class. The functions and operators which are brought in as methods from the namespace `NS` are treated exactly as if the source of each function/operator had been included in the class script at the point of the `:Include` statement. For example, if a function contains `:Signature` or `:Access` statements, these will be taken into account. Note that such declarations have no effect on a function/operator which is in an ordinary namespace.

D-fns and D-ops in `NS` are also included in the Class but as *Private members*, because D-fns and D-ops may not contain `:Signature` or `:Access` statements. Variables and Sub-namespaces in `NS` are **not** included.

Note that objects imported in this way are not actually *copied*, so there is no penalty incurred in using this feature. Additions, deletions and changes to the functions in `NS` are immediately reflected in the Class.

If there is a member in the Class with the same name as a function in `NS`, the Class member takes precedence and supersedes the function in `NS`.

Conversely, functions in `NS` will supersede members of the same name that are inherited from the Base Class, so the precedence is:

Class supersedes
  Included Namespace, supersedes
    Base Class

Any number of Namespaces may be included in a Class and the `:Include` statements may occur anywhere in the Class script. However, for the sake of readability, it is recommended that you have `:Include` statements at the top, given that any definitions in the script will supersede included functions and operators.

Example

In this example, Class `Penguin` inherits from `Animal` and includes functions from the plain Namespaces `BirdStuff` and `FishStuff`.

```
:Class Penguin: Animal
    :Include BirdStuff
    :Include FishStuff
:EndClass ⍝ Penguin
```

Namespace `BirdStuff` contains 2 functions, both declared as Public methods.

```
:Namespace BirdStuff
    ∇ R←Fly
      :Access Public Instance
      R←'Fly, Fly ...'
    ∇
    ∇ R←Lay
      :Access Public Instance
      R←'Lay, Lay ...'
    ∇
:EndNamespace ⍝ BirdStuff
```

Namespace `FishStuff` contain a single function, also declared as a Public method.

```
:Namespace FishStuff
    ∇ R←Swim
      :Access Public Instance
      R←'Swim, Swim ...'
    ∇
:EndNamespace ⍝ FishStuff

      Pingo←⎕NEW Penguin
      Pingo.Swim
Swim, Swim ...
      Pingo.Lay
Lay, Lay ...
      Pingo.Fly
Fly, Fly ...
```

This is getting silly - we all know that Penguin's can't fly. This problem is simply resolved by overriding the `BirdStuff.Fly` method with `Penguin.Fly`. We can hide `BirdStuff.Fly` with a Private method in `Penguin` that does nothing. For example:

```
:Class Penguin: Animal
    :Include BirdStuff
    :Include FishStuff
    ∇ Fly ⍝ Override BirdStuff.Fly
    ∇
:EndClass ⍝ Penguin

      Pingo←⎕NEW Penguin
      Pingo.Fly
VALUE ERROR
      Pingo.Fly
     ^
```

or we can supersede it with a different Public method, as follows:

```
:Class Penguin: Animal
    :Include BirdStuff
    :Include FishStuff
    ∇ R←Fly ⍝ Override BirdStuff.Fly
      :Access Public Instance
      R←'Sadly, I cannot fly'
    ∇
:EndClass ⍝ Penguin

      Pingo←⎕NEW Penguin
      Pingo.Fly
Sadly, I cannot fly
```

Nested Classes

It is possible to define *Classes within Classes* (Nested Classes).

A Nested Class may be either `Private` or `Public`. This is specified by a :Access Statement, which must precede the definition of any Class contents. The default is `Private`.

A `Public` Nested Class is visible from outside its containing Class and may be used directly in its own right, whereas a `Private` Nested Class is not and may only be used by code inside the containing Class.

However, methods in the containing Class may return instances of Private Nested Classes and in that way expose them to the calling environment.

The GolfService Example Class illustrates the use of nested classes. GolfService was originally developed as a Web Service for Dyalog.Net and is one of the samples distributed in samples\asp.net\webservices. This version has been reconstructed as a stand-alone APL Class.

GolfService contains the following nested classes, all of which are `Private`.

GolfCourse	A Class that represents a Golf Course, having Fields `Code` and `Name`.
Slot	A Class that represents a tee-time or match, having Fields `Time` and `Players`. Up to 4 players may play together in a match.
Booking	A Class that represents a reservation for a particular tee-time at a particular golf course. This has Fields `OK`, `Course`, `TeeTime` and `Message`. The value of `TeeTime` is an Instance of a Slot Class.
StartingSheet	A Class that represents a day's starting-sheet at a particular golf course. It has Fields `OK`, `Course`, `Date`, `Slots`, `Message`. `Slots` is an array of Instances of Slot Class.

GolfService Example Class

```
:Class GolfService
:Using System

    :Field Private GOLFILE←'' ⍝ Name of Golf data file
    :Field Private GOLFID←0 ⍝ Tie number Golf data file

    :Class GolfCourse
        :Field Public Code←¯1
        :Field Public Name←''

        ∇ ctor args
          :Implements Constructor
          :Access Public Instance
          Code Name←args
          ⎕DF Name,'(',(⍕Code),')'
        ∇

    :EndClass

    :Class Slot
        :Field Public Time
        :Field Public Players

        ∇ ctor1 t
          :Implements Constructor
          :Access Public Instance
          Time←t
          Players←0⍴⊂''
        ∇
        ∇ ctor2 (t pl)
          :Implements Constructor
          :Access Public Instance
          Time Players←t pl
        ∇
        ∇ format
          :Implements Trigger Players
          ⎕DF⍕Time Players
        ∇
    :EndClass
```

```
:Class Booking
    :Field Public OK
    :Field Public Course
    :Field Public TeeTime
    :Field Public Message

    ∇ ctor args
      :Implements Constructor
      :Access Public Instance
      OK Course TeeTime Message←args
    ∇
    ∇ format
      :Implements Trigger OK,Message
      ⎕DF⍕Course TeeTime(⊃OK⌽Message'OK')
    ∇
:EndClass

:Class StartingSheet
    :Field Public OK
    :Field Public Course
    :Field Public Date
    :Field Public Slots←⎕NULL
    :Field Public Message

    ∇ ctor args
      :Implements Constructor
      :Access Public Instance
      OK Course Date←args
    ∇
    ∇ format
      :Implements Trigger OK,Message
      ⎕DF⍕2 1⍴(⍕Course Date)(↑⍕¨Slots)
    ∇
:EndClass

∇ ctor file
  :Implements Constructor
  :Access Public Instance
  GOLFILE←file
  ⎕FUNTIE(((↓⎕FNAMES)~' ')⍳⊂GOLFILE)⊃⎕FNUMS,0
  :Trap 22
      GOLFID←GOLFILE ⎕FTIE 0
  :Else
      InitFile
  :EndTrap
∇
```

```
∇ dtor
  :Implements Destructor
  ⎕FUNTIE GOLFID
∇

∇ InitFile;COURSECODES;COURSES;INDEX;I
  :Access Public
  :If GOLFID≠0
      GOLFILE ⎕FERASE GOLFID
  :EndIf
  GOLFID←GOLFILE ⎕FCREATE 0
  COURSECODES←1 2 3
  COURSES←'St Andrews' 'Hindhead' 'Basingstoke'
  INDEX←(⍴COURSES)⍴0
  COURSECODES COURSES INDEX ⎕FAPPEND GOLFID
  :For I :In ⍳⍴COURSES
      INDEX[I]←⍬ ⍬ ⎕FAPPEND 1
  :EndFor
  COURSECODES COURSES INDEX ⎕FREPLACE GOLFID 1
∇

∇ R←GetCourses;COURSECODES;COURSES;INDEX
  :Access Public
  COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
  R←{⎕NEW GolfCourse ⍵}¨↓⍉↑COURSECODES COURSES
∇
```

```
∇ R←GetStartingSheet ARGS;CODE;COURSE;DATE;COURSECODES
                         ;COURSES;INDEX;COURSEI;IDN
                         ;DATES;COMPS;IDATE;TEETIMES
                         ;GOLFERS;I;T
  :Access Public
  CODE DATE←ARGS
  COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
  COURSEI←COURSECODES⍳CODE
  COURSE←⎕NEW GolfCourse(CODE(COURSEI⊃COURSES,⊂''))
  R←⎕NEW StartingSheet(0 COURSE DATE)
  :If COURSEI>⍴COURSECODES
      R.Message←'Invalid course code'
      :Return
  :EndIf
  IDN←2 ⎕NQ'.' 'DateToIDN',DATE.(Year Month Day)
  DATES COMPS←⎕FREAD GOLFID,COURSEI⊃INDEX
  IDATE←DATES⍳IDN
  :If IDATE>⍴DATES
      R.Message←'No Starting Sheet available'
      :Return
  :EndIf
  TEETIMES GOLFERS←⎕FREAD GOLFID,IDATE⊃COMPS
  T←DateTime.New¨(⊂DATE.(Year Month Day)),¨↓[1]
                                  24 60 1⊤TEETIMES
  R.Slots←{⎕NEW Slot ⍵}¨T,∘⊂¨↓GOLFERS
  R.OK←1
∇
```

```
∇ R←MakeBooking ARGS;CODE;COURSE;SLOT;TEETIME
                    ;COURSECODES;COURSES;INDEX
                    ;COURSEI;IDN;DATES;COMPS;IDATE
                    ;TEETIMES;GOLFERS;OLD;COMP;HOURS
                    ;MINUTES;NEAREST;TIME;NAMES;FREE
                    ;FREETIMES;I;J;DIFF
  :Access Public
  ⍝ If GimmeNearest is 0, tries for specified time
  ⍝ If GimmeNearest is 1, gets nearest time
  CODE TEETIME NEAREST←3↑ARGS
  COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
  COURSEI←COURSECODES⍳CODE
  COURSE←⎕NEW GolfCourse(CODE(COURSEI⊃COURSES,⊂''))
  SLOT←⎕NEW Slot TEETIME
  R←⎕NEW Booking(0 COURSE SLOT'')
  :If COURSEI>⍴COURSECODES
      R.Message←'Invalid course code'
      :Return
  :EndIf
  :If TEETIME.Now>TEETIME
      R.Message←'Requested tee-time is in the past'
      :Return
  :EndIf
  :If TEETIME>TEETIME.Now.AddDays 30
      R.Message←'Requested tee-time is more than 30
                                       days from now'
      :Return
  :EndIf
  IDN←2 ⎕NQ'.' 'DateToIDN',TEETIME.(Year Month Day)
  DATES COMPS←⎕FREAD GOLFID,COURSEI⊃INDEX
  IDATE←DATES⍳IDN
  :If IDATE>⍴DATES
      TEETIMES←(24 60⊥7 0)+10×¯1+⍳1+8×6
      GOLFERS←((⍴TEETIMES),4)⍴⊂''llowed per tee time
      :If 0=OLD←⊃(DATES<2 ⎕NQ'.' 'DateToIDN',3↑⎕TS)/
                                                ⍳⍴DATES
          COMP←(TEETIMES GOLFERS)⎕FAPPEND GOLFID
          DATES,←IDN
          COMPS,←COMP
          (DATES COMPS)⎕FREPLACE GOLFID,COURSEI⊃INDEX
      :Else
          DATES[OLD]←IDN
          (TEETIMES GOLFERS)⎕FREPLACE GOLFID,
                                       COMP←OLD⊃COMPS
          DATES COMPS ⎕FREPLACE GOLFID,COURSEI⊃INDEX
      :EndIf
```

```
          :Else
          COMP←IDATE⊃COMPS
          TEETIMES GOLFERS←⎕FREAD GOLFID COMP
      :EndIf
      HOURS MINUTES←TEETIME.(Hour Minute)
      NAMES←(3↓ARGS)~θ''
      TIME←24 60⊥HOURS MINUTES
      TIME←10×⌊0.5+TIME÷10
      :If ~NEAREST
          I←TEETIMES⍳TIME
          :If I>⍴TEETIMES
          :OrIf (⍴NAMES)>⊃,/+/0=⍴¨GOLFERS[I;]
              R.Message←'Not available'
              :Return
          :EndIf
      :Else
          :If ~∨/FREE←(⍴NAMES)≤⊃,/+/0=⍴¨GOLFERS
              R.Message←'Not available'
              :Return
          :EndIf
          FREETIMES←(FREE×TEETIMES)+32767×~FREE
          DIFF←|FREETIMES-TIME
          I←DIFF⍳⌊/DIFF
      :EndIf
      J←(⊃,/0=⍴¨GOLFERS[I;])/⍳4
      GOLFERS[I;(⍴NAMES)↑J]←NAMES
      (TEETIMES GOLFERS)⎕FREPLACE GOLFID COMP
      TEETIME←DateTime.New TEETIME.(Year Month Day),
                                  3↑24 60⊤I⊃TEETIMES
      SLOT.Time←TEETIME
      SLOT.Players←(⊃,/0<⍴¨GOLFERS[I;])/GOLFERS[I;]
      R.(OK TeeTime)←1 SLOT
    ∇

:EndClass
```

The GolfService constructor takes the name of a file in which all the data is stored. This file is initialised by method `InitFile` if it doesn't already exist.

```
      G←⎕NEW GolfService 'F:\HELP11.0\GOLFDATA'
      G
#.[Instance of GolfService]
```

The GetCourses method returns an array of Instances of the internal (nested) Class GolfCourse. Notice how the display form of each Instance is established by the GolfCourse constructor, to obtain the output display shown below.

```
      G.GetCourses
 St Andrews(1)  Hindhead(2)  Basingstoke(3)
```

All of the dates and times employ instances of the .Net type System.DateTime, and the following statements just set up some temporary variables for convenience later.

```
      ⎕←Tomorrow←(⎕NEW DateTime(3↑⎕TS)).AddDays 1
31/03/2006 00:00:00
      ⎕←TomorrowAt7←Tomorrow.AddHours 7
31/03/2006 07:00:00
```

The MakeBooking method takes between 4 and 7 parameters viz.

the code for the golf course at which the reservation is required
the date and time of the reservation
a flag to indicate whether or not the nearest available time will do
and a list of up to 4 players who wish to book that time.

The result is an Instance of the internal Class Booking. Once again, `⎕DF` is used to make the default display of these Instances meaningful. In this case, the reservation is successful.

```
      G.MakeBooking 2 TomorrowAt7 1 'Pete' 'Tiger'
 Hindhead(2)   31/03/2006 07:00:00   Pete  Tiger    OK
```

Bob, Arnie and Jack also ask to play at 7:00 but are given the 7:10 tee-time instead (4-player restriction).

```
      G.MakeBooking 2 TomorrowAt7 1 'Bob' 'Arnie' 'Jack'
 Hindhead(2)   31/03/2006 07:10:00   Bob  Arnie  Jack    O
K
```

However, Pete and Tiger are joined at 7:00 by Dave and Al.

```
      G.MakeBooking 2 TomorrowAt7 1 'Dave' 'Al'
 Hindhead(2)   31/03/2006 07:00:00   Pete  Tiger  Dave  Al
    OK
```

Up to now, all bookings have been made with the tee-time flexibility flag set to 1. Inflexible Jim is only interested in playing at 7:00 …

```
      G.MakeBooking 2 TomorrowAt7 0 'Jim'
 Hindhead(2)  31/03/2006 07:00:00  Not available
```

… so his reservation fails (4-player restriction).

Finally the GetStartingSheet method is used to obtain an Instance of the internal Class StartingSheet for the given course and day.

```
      G.GetStartingSheet 2 Tomorrow
  Hindhead(2)  31/03/2006 00:00:00
  31/03/2006 07:00:00   Pete  Tiger  Dave  Al
  31/03/2006 07:10:00   Bob  Arnie  Jack
  31/03/2006 07:20:00
  ....
```

Namespace Scripts

A Namespace Script is a script that begins with a `:Namespace` statement and ends with a `:EndNamespace` statement. When a Namespace Script is fixed, it establishes an entire namespace that may contain other namespaces, functions, variables and classes.

The names of Classes defined within a Namespace Script which are parents, children, or siblings are visible both to one another and to code and expressions defined in the same script, regardless of the namespace hierarchy within it. Names of Classes which are nieces or nephews and their descendants are however not visible.

For example:

```
:Namespace a
    d←⎕NEW a1
    e←⎕NEW bb2

    :Class a1
        ∇ r←foo
          :Access Shared Public
          r←⎕NEW¨b1 b2
        ∇
    :EndClass ⍝ a1

    ∇ r←goo
      r←a1.foo
    ∇

    ∇ r←foo
      r←⎕NEW¨b1 b2
    ∇

    :Namespace b
        :Class b1
        :EndClass ⍝ b1
        :Class b2
            :Class bb2
            :EndClass ⍝ bb2
        :EndClass ⍝ b2
    :EndNamespace ⍝ b
:EndNamespace ⍝ a
```

```
      a.d
#.a.[a1]
      a.e
#.a.[bb2]
      a.foo
 #.a.[b1]  #.a.[b2]
```

Note that the names of Classes `b1` (`a.b.b1`) and `b2` (`a.b.b2`) are not visible from their “uncle” `a1` (`a.a1`).

```
      a.goo
VALUE ERROR
foo[2] r←⎕NEW¨b1 b2
```

Notice that Classes in a Namepsace Script are fixed before other objects (hence the assignments to `d` and `e` are evaluated *after* Classes `a1` and `bb2` are fixed), although the order in which Classes themselves are defined is still important if they reference one another during initialisation.

Warning: If you introduce new objects of any type (functions, variables, or classes) into a namespace defined by a script by any other means than editing the script, then these objects will be lost the next time the script is edited and fixed. Also, if you modify a variable which is defined in a script, the script will not be updated.

Namespace Script Example

The DiaryStuff example illustrates the manner in which classes may be defined and used in a Namespace script.

DiaryStuff defines two Classes named `Diary` and `DiaryEntry`.

`Diary` contains a (private) Field named `entries`, which is simply a vector of instances of `DiaryEntry`. These are 2-element vectors containing a .NET DateTime object and a description.

The `entries` Field is initialised to an empty vector of `DiaryEntry` instances which causes the invocation of the default constructor `DiaryEntry.Make0` when `Diary` is fixed. See Empty Arrays of Instances for further explanation.

The `entries` Field is referenced through the `Entry` Property, which is defined as the Default Property. This allows individual entries to be referenced and changed using indexing on a `Diary` Instance.

Note that `DiaryEntry` is defined in the script first (before `Diary`) because it is referenced by the initialisation of the `Diaries.entries` Field

```
:Namespace DiaryStuff
:Using System

    :Class DiaryEntry
        :Field Public When
        :Field Public What
        ∇ Make(ymdhm wot)
          :Access Public
          :Implements Constructor
          When What←(⎕NEW DateTime(6↑5↑ymdhm))wot
          ⎕DF⍕When What
        ∇
        ∇ Make0
          :Access Public
          :Implements Constructor
          When What←⎕NULL''
        ∇
    :EndClass ⍝ DiaryEntry
```

```
        :Class Diary
            :Field Private entries←0⍴⎕NEW DiaryEntry
            ∇ R←Add(ymdhm wot)
              :Access Public
              R←⎕NEW DiaryEntry(ymdhm wot)
              entries,←R
            ∇
            ∇ R←DoingOn ymd;X
              :Access Public
              X←,(↑entries.When.(Year Month Day))∧.=3 1⍴3↑ymd
              R←X/entries
            ∇
            ∇ R←Remove ymdhm;X
              :Access Public
              :If R←∨/X←entries.When=⎕NEW DateTime(6↑5↑ymdhm)
                  entries←(~X)/entries
              :EndIf
            ∇
            :Property Numbered Default Entry
                ∇ R←Shape
                  R←⍴entries
                ∇
                ∇ R←Get arg
                  R←arg.Indexers⊃entries
                ∇
                ∇ Set arg
                  entries[arg.Indexers]←arg.NewValue
                ∇
            :EndProperty
        :EndClass ⍝ Diary

    :EndNamespace
```

Create a new instance of `Diary`.

```
      D←⎕NEW DiaryStuff.Diary
```

Add a new entry "meeting with John at 09:00 on April 30th"

```
      D.Add(2006 4 30 9 0)'Meeting with John'
 30/04/2006 09:00:00  Meeting with John
```

Add another diary entry "Dentist at 10:00 on April 30th".

```
      D.Add(2006 4 30 10 0)'Dentist'
 30/04/2006 10:00:00  Dentist
```

One of the benefits of the Namespace Script is that Classes defined within it (which are typically *related*) may be used *independently*, so we can create a stand-alone instance of `DiaryEntry`; "Doctor at 11:00"…

```
  Doc←⎕NEW DiaryStuff.DiaryEntry((2006 4 30 11 0)'Doctor')
      Doc
 30/04/2006 11:00:00  Doctor
```

… and then use it to replace the second Diary entry with indexing:

```
      D[2]←Doc
```

and just to confirm it is there …

```
      D[2]
 30/04/2006 11:00:00  Doctor
```

What am I doing on the 30th ?

```
      D.DoingOn 2006 4 30
  30/04/2006 09:00:00  Meeting with John    ...
  ... 30/04/2006 11:00:00  Doctor
```

Remove the 11:00 appointment …

```
      D.Remove 2006 4 30 11 0
1
```

and the complete Diary is …

```
      ⎕D
  30/04/2006 09:00:00  Meeting with John
```

Class Declaration Statements

This section summarises the various declaration statements that may be included in a Class or Namespace Script. For information on other declaration statements, as they apply to functions and methods, see Function Declaration Statements.

:Interface Statement

```
:Interface <interface name>
...
:EndInterface
```

An Interface is defined by a Script containing skeleton declarations of Properties and/or Methods. The script must begin with a `:Interface Statement` and end with a `:EndInterface Statement`.

An Interface may not contain Fields.

There is no need for the Properties and Methods defined in an Interface to contain `:Access` Statements as these will be overridden by the `:Access` declarations within the Classes that implement the Interface.

:Namespace Statement

```
:Namespace <namespace name>
...
:EndNamespace
```

A Namespace Script may be used to define an entire namespace containing other namespaces, functions, variables and Classes.

A Namespace script must begin with a `:Namespace` statement and end with a `:EndNamespace` statement.

Sub-namespaces, which may be nested, are defined by pairs of `:Namespace` and `:EndNamespace` statements within the Namespace script.

Classes are defined by pairs of `:Class` and `:EndClass` statements within the Namespace script, and these too may be nested.

The names of Classes defined within a Namespace Script are visible both to one another and to code and expressions defined in the same script, regardless of the namespace hierarchy within it.

A Namespace script is therefore particularly useful to group together Classes that refer to one another where the use of nested classes is inappropriate.

:Class Statement

```
:Class <class name><:base class name> <,interface name...>
:Include <namespace>
...
:EndClass
```

A class script begins with a `:Class` statement and ends with a `:EndClass` statement. The elements that comprise the `:Class` statement are as follows:

Element	Description
`class name`	Optionally, specifies the name of the Class, which must conform to the rules governing APL names.
`base class name`	Optionally specifies the name of a Class from which this Class is derived and whose members this Class inherits.
`interface name`	The names of one or more Interfaces which this Class supports.

A Class may import methods defined in separate plain Namespaces with one or more `:Include` statements. For further details, see Including Namespaces in Classes

Examples:

The following statements define a Class named `Penguin` that derives from (is based upon) a Class named `Animal` and which supports two Interfaces named `BirdBehaviour` and `FishBehaviour`.

```
:Class Penguin: Animal,BirdBehaviour,FishBehaviour
...
:EndClass
```

The following statements define a Class named `Penguin` that derives from (is based upon) a Class named `Animal` and includes methods defined in two separate Namespaces named `BirdStuff` and `FishStuff`.

```
:Class Penguin: Animal
:Include BirdStuff
:Include FishStuff
...
:EndClass
```

:Using Statement

```
:Using <NameSpace[,Assembly]>
```

This statement specifies a .NET namespace that is to be searched to resolve unqualified names of .NET types referenced by expressions in the Class.

Element	Description
`NameSpace`	Specifies a .NET namespace.
`Assembly`	Specifies the Assembly in which NameSpace is located. If the Assembly is defined in the *global assembly cache*, you need only specify its name. If not, you must specify a full or relative pathname.

If the Microsoft .Net Framework is installed, the System namespace in mscorlib.dll is automatically loaded when Dyalog APL starts. To access this namespace, it is not necessary to specify the name of the Assembly.

When the class is fixed, `⎕USING` is inherited from the surrounding space. Each `:Using` statement appends an element to `⎕USING`, with the exception of `:Using` with no argument:

If you omit `<Namespace>`, this is equivalent to clearing `⎕USING`, which means that no .NET namespaces will be searched (unless you follow this statement with additional `:Using` statements, each of which will append to `⎕USING`).

To set `⎕USING`, to a single empty character vector, which only allows references to fully qualified names of classes in `mscorlib.dll`, you must write:

```
      :Using  ,
```
(note the presence of the comma)

or
```
      :Using ,mscorlib.dll
```

(i.e. specify an empty namespace name followed by no assembly, or followed by the default assembly, which is always loaded.

:Attribute Statement

```
:Attribute <Name> [ConstructorArgs]
```

The :Attribute statement is used to attach .Net Attributes to a Class or a Method.

Attributes are descriptive tags that provide additional information about programming elements. Attributes are not used by Dyalog APL but other applications can refer to the extra information in attributes to determine how these items can be used. Attributes are saved with the *metadata* of Dyalog APL .NET assemblies.

Element	Description
`Name`	The name of a .Net attribute
`ConstructorArgs`	Optional arguments for the Attribute constructor

Example

The following Class has `SerializableAttribute` and `CLSCompliantAttribute` attributes attached to the Class as a whole, and `ObsoleteAttribute` attributes attached to Methods `foo` and `goo` within it.

```
:Class c1
:using System
    :attribute SerializableAttribute
    :attribute CLSCompliantAttribute 1

    ∇ foo(p1 p2)
      :Access public instance
      :Signature foo Object,Object
      :Attribute ObsoleteAttribute
    ∇

    ∇ goo(p1 p2)
      :Access public instance
      :Signature foo Object,Object
      :Attribute ObsoleteAttribute 'Don''t use this' 1

    ∇

:EndClass ⍝ c1
```

When this Class is exported as a .Net Class, the attributes are saved in its metadata. For example, Visual Studio will warn developers if they make use of a member which has the ObsoleteAttribute.

:Access Statement

```
:Access <Private|Public><Instance|Shared><Overridable>
                                        <Override>
:Access <WebMethod>
```

The :Access statement is used to specify characteristics for Classes, Properties and Methods.

Element	Description
Private|Public	Specifies whether or not the (nested) Class, Property or Method is accessible from outside the Class or an Instance of the Class. The default is `Private`.
Instance|Shared	For a Field, specifies if there is a separate value of the Field in each Instance of the Class, or if there is only a single value that is shared between all Instances. For a Property or Method, specifies whether the code associated with the Property or Method runs in the Class or Instance..
WebMethod	Applies only to a Method and specifies that the method is exported as a web method. This applies only to a Class that implements a Web Service.
Overridable	Applies only to an Instance Method and specifies that the Method may be overridden by a Method in a higher Class. See below.
Override	Applies only to an Instance Method and specifies that the Method overrides the corresponding Overridable Method defined in the Base Class. See below.

Overridable/Override

Normally, a Method defined in a higher Class replaces a Method of the same name that is defined in its Base Class, but only for calls made from above or within the higher Class itself (or an Instance of the higher Class). The base method remains available *in the Base Class* and is invoked by a reference to it *from within the Base Class*.

However, a Method declared as being `Overridable` is replaced in situ (i.e. within its own Class) by a Method of the same name in a higher Class if that Method is itself declared with the `Override` keyword. For further information, see Superseding Base Class Methods.

Nested Classes

The :Access statement is also used to control the visibility of one Class that is defined within another (a nested Class). A Nested Class may be either `Private` or `Public`. Note that the :Access Statement must precede the definition of any Class contents..

A `Public` Nested Class is visible from outside its containing Class and may be used directly in its own right, whereas a `Private` Nested Class is not and may only be used by code inside the containing Class.

However, methods in the containing Class may return instances of Private Nested Classes and in that way expose them to the calling environment.

WebMethod

Note that `:Access WebMethod` is equivalent to:

```
:Access Public
:Attribute System.Web.Services.WebMethodAttribute
```

:Field Statement

```
:Field <Private|Public> <Instance|Shared> <ReadOnly>...
       ... FieldName <← expr>
```

A `:Field` statement is a single statement whose elements are as follows:

Element	Description
`Private|Public`	Specifies whether or not the Field is accessible from outside the Class or an Instance of the Class. The default is `Private`.
`Instance|Shared`	Specifies if there is a separate value of the Field in each Instance of the Class, or if there is only a single value that is shared between all Instances.
`ReadOnly`	If specified, this keyword prevents the value in the Field from being changed after initialisation.
`FieldName`	Specifies the name of the Field (mandatory).
`← expr`	Specifies an initial value for the Field.

Examples:

The following statement defines a Field called `Name`. It is (by default), an Instance Field so every Instance of the Class has a separate value. It is a Public Field and so may be accessed (set or retrieved) from outside an Instance.

```
:Field Public Name
```

The following statement defines a Field called `Months`.

```
:Field Shared ReadOnly Months←12↑(⎕NEW DateTimeFormatInfo)
                                .AbbreviatedMonthNames
```

`Months` is a Shared Field so there is just a single value that is the same for every Instance of the Class. It is (by default), a Private Field and may only be referenced by code running in an Instance or in the Class itself. Furthermore, it is ReadOnly and may not be altered after initialisation. It's initial value is calculated by an expression that obtains the short month names that are appropriate for the current locale using the .Net Type DateTimeFormatInfo.

Note that Fields are initialised when a Class script is fixed by the editor or by `⎕FIX`. If the evaluation of `expr` causes an error (for example, a `VALUE ERROR`), an appropriate message will be displayed in the Status Window and `⎕FIX` will fail with a `DOMAIN ERROR`. Note that a ReadOnly Field may only be assigned a value by its `:Field` statement.

In the second example above, the expression will only succeed if `⎕USING` is set to the appropriate path, in this case System.Globalization.

:Property Section

A Property is defined by a `:Property ... :EndProperty` section in a Class Script. The syntax of the :Property Statement, and its optional `:Access` statement is as follows:

```
:Property <Simple|Numbered|Keyed> <Default> Name<,Name>...
:Access <Private|Public><Instance|Shared>
...
:EndProperty
```

Element	Description
`Name`	Specifies the name of the Property by which it is accessed. Additional Properties, sharing the same PropertyGet and/or PropertySet functions, and the same access behaviour may be specified by a comma-separated list of names.
`Simple|Numbered|Keyed`	Specifies the type of Property (see below). The default is `Simple`.
`Default`	Specifies that this Property acts as the default property for the Class when indexing is applied directly to an Instance of the Class.
`Private|Public`	Specifies whether or not the Property is accessible from outside the Class or an Instance of the Class. The default is `Private`.
`Instance|Shared`	Specifies if there is a separate value of the Property in each Instance of the Class, or if there is only a single value that is shared between all Instances.

A Simple Property is one whose value is accessed (by APL) in its entirety and re-assigned (by APL) in its entirety.

A Numbered Property behaves like an array (conceptually a vector) which is only ever *partially* accessed and set (one element at a time) via indices.

A Keyed Property is similar to a Numbered Property except that its elements are accessed via arbitrary keys instead of indices.

Numbered and Keyed Properties are designed to allow APL to perform selections and structural operations on the Property.

Within the body of a Property Section there may be:

- one or more `:Access` statements
- a single PropertyGet function.
- a single PropertySet function
- a single PropertyShape function

The three functions are identified by case-independent names `Get`, `Set` and `Shape`.

When a Class is fixed by the Editor or by `⎕FIX`, APL checks the validity of each Property section and the syntax of PropertyGet, PropertySet and PropertyShape functions within them. If anything is wrong, an error is generated and the Class is not fixed.

PropertyArguments Class

Where appropriate, APL supplies the PropertyGet and PropertySet functions with an argument that is an instance of the internal class `PropertyArguments`.

`PropertyArguments` has 2-3 read-only Fields which are as follows:

`Name`	The name of the property. This is useful when one function is handling several properties.
`NewValue`	Array containing the new value for the Property or for selected element(s) of the property as specified by `Indexers`.
`Indexers`	An array that identifies the element(s) of the Property that are to be referenced or assigned.

:PropertyGet Function Syntax

```
PropertyGet Syntax:     R←Get
                        R←Get ipa
```

The name of the PropertyGet function must be `Get`, but is case-independent. For example, `get`, `Get`, `gEt` and `GET` are all valid names for the PropertyGet function

The PropertyGet function must be result returning. For a Simple Property, it may be monadic or niladic. For a Numbered or Keyed Property it must be monadic.

For a Simple Property, the result `R` may be any array. For a Numbered Property, the result `R` must be scalar. For a Keyed Property, `R` must conform to the rank and shape specified by `ipa.Indexers`.

If monadic, `ipa` is an instance of the internal class PropertyArguments

In all cases, `ipa.Name` contains the name of the Property being referenced and `NewValue` is undefined (`VALUE ERROR`).

If the Property is *Simple*, `ipa.Indexers` is undefined (`VALUE ERROR`).

If the Property is *Numbered*, `ipa.Indexers` is an integer vector of the same length as the rank of the property (as implied by the result of the `Shape` function) that identifies a single element of the Property whose value is to be obtained. In this case, `R` must be scalar.

If the Property is Keyed, `ipa.Indexers` is a vector containing the arrays that were specified within the square brackets in the reference expression. Specifically, `ipa.Indexers` will contain one more elements than the number of semi-colon (;) separators.

PropertySet Function Syntax

PropertySet Syntax: `Set ipa`

The name of the PropertySet function must be `Set`, but is case-independent. For example, `set`, `Set`, `sEt` and `SET` are all valid names for the PropertySet function.

The PropertySet function must be monadic and may not return a result.

`ipa` is an instance of the internal class PropertyArguments.

In all cases, `ipa.Name` contains the name of the Property being referenced and `NewValue` contains the new value(s) for the element(s) of the Property being assigned.

If the Property is *Simple*, `ipa.Indexers` is undefined (`VALUE ERROR`).

If the Property is *Numbered*, `ipa.Indexers` is an integer vector of the same length as the rank of the property (as implied by the result of the `Shape` function) that identifies a single element of the Property whose value is to be set.

If the Property is Keyed, `ipa.Indexers` is a vector containing the arrays that were specified within the square brackets in the assignment expression. Specifically, `ipa.Indexers` will contain one fewer elements than, the number of semi-colon (;) separators.

PropertyShape Function Syntax

```
PropertyShape Syntax:    R←Shape
                         R←Shape ipa
```

The name of the PropertyShape function must be `Shape`, but is case-independent. For example, `shape`, `Shape`, `sHape` and `SHAPE` are all valid names for the PropertyShape function.

A PropertyShape function is only called if the Property is a Numbered Property.

The PropertyShape function must be niladic or monadic and must return a result.

If monadic, `ipa` is an instance of the internal class PropertyArguments. `ipa.Name` contains the name of the Property being referenced and `NewValue` and `Indexers` are undefined (`VALUE ERROR`).

The result `R` must be an integer vector or scalar that specifies the `rank` of the Property. Each element of `R` specifies the length of the corresponding dimension of the Property. Otherwise, the reference or assignment to the Property will fail with `DOMAIN ERROR`.

Note that the result `R` is used by APL to check that the number of indices corresponds to the rank of the Property and that the indices are within the bounds of its dimensions. If not, the reference or assignment to the Property will fail with `RANK ERROR` or `LENGTH ERROR`..

CHAPTER 3

Using Classes with the Dyalog GUI and .Net

Using the Dyalog GUI

With the introduction of Classes in Version 11.0, you may manipulate Dyalog GUI objects as Instances of built-in (GUI) Classes. This approach supplements (but does not replace) the use of ⎕WC, ⎕WS and so forth.

To create a GUI object using ⎕NEW, the Class is given as the GUI Object name and the Constructor Argument as a vector of (Property Name / Property Value) pairs. For example, to create a Form:

```
      F1←⎕NEW 'Form' (⊂'Caption' 'Hello World')
```

Notice however that only perfectly formed name/value pairs are accepted. The highly flexible syntax for specifying Properties by position and omitting levels of enclosure, that is supported by ⎕WC and ⎕WS, is not provided with ⎕NEW.

Naturally, you may reference and assign Properties in the same way as for objects created using ⎕WC:

```
      F1.Size
50 50
      F1.Size←20 30
```

Callbacks to regular defined finctions in the Root or in another space, work in the same way too. If function FOO merely displays its argument:

```
     ∇ FOO M
[1]    ⎕←M
     ∇

      F1.onMouseUp←'#.FOO'
#.[Form]  MouseUp  78.57142639 44.62540...
```

Note that the first item in the event message is a ref to the Instance of the Form.

To create a control such as a Button, it is only necessary to run ⎕NEW inside a ref to the appropriate parent object. For example:

```
   B1←F1.⎕NEW 'Button' (('Caption' '&OK')('Size' (10 10)))
```

As illustrated in this example, it is not necessary to assign the resulting Button Instance to a name *inside* the Form (F1 in this case). However, it is a good idea to do so so that refs to Instances of controls are expunged when the parent object is expunged. In the example above, expunging F1 will not remove the Form from the screen because B1 still exists as a ref to the Button. So, the following is safer:

```
  F1.B1←F1.⎕NEW'Button'(('Caption' '&OK')('Size' (10 10)))
```

Or pehaps better still,

```
  F1.(B1←⎕NEW 'Button'(('Caption' '&OK')('Size' (10 10))))
```

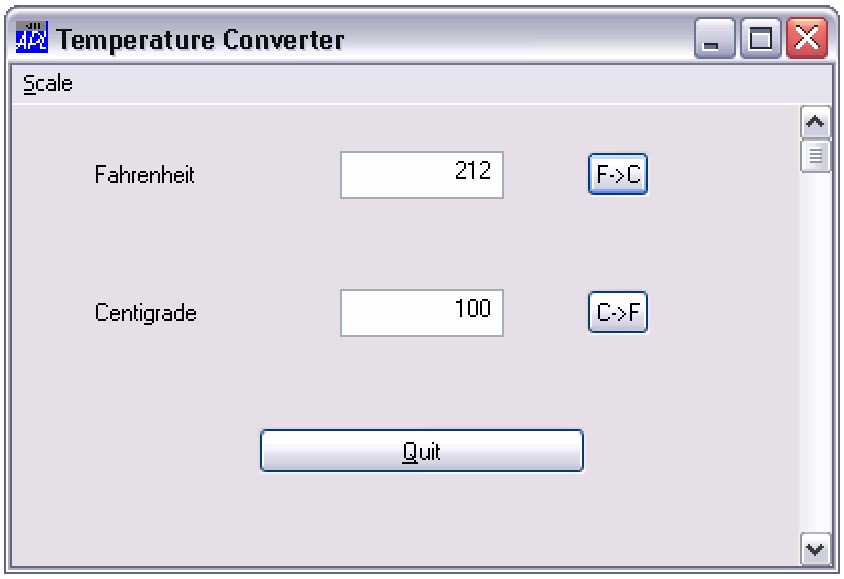
Temperature Converter Example

The following function illustrates this approach using the Temperature Converter example descibed previously.

```
     ∇ TempConv;TITLE;TEMP
       TITLE←'Temperature Converter'
       TEMP←⎕NEW'Form'(('Caption'TITLE)('Posn'(10 10))
                       ('Size'(30 40)))

       TEMP.(MB←⎕NEW⊂'MenuBar')
       TEMP.MB.(M←⎕NEW'Menu'(,⊂'Caption' '&Scale'))
       TEMP.MB.M.(C←⎕NEW'MenuItem'
                 (('Caption' '&Centigrade')('Checked' 1)))
       TEMP.MB.M.(F←⎕NEW'MenuItem'
                 (,⊂('Caption' '&Fahrenheit')))

       TEMP.(LF←⎕NEW'Label'(('Caption' 'Fahrenheit')
                            ('Posn'(10 10))))
       TEMP.(F←⎕NEW'Edit'(('Posn'(10 40))('Size'(⍬ 20))
                          ('FieldType' 'Numeric')))

       TEMP.(LC←⎕NEW'Label'(('Caption' 'Centigrade')
                            ('Posn'(40 10))))
       TEMP.(C←⎕NEW'Edit'(('Posn'(40 40))('Size'(⍬ 20))
                          ('FieldType' 'Numeric')))

       TEMP.(F2C←⎕NEW'Button'(('Caption' 'F->C')
                              ('Posn'(10 70))('Default' 1))
)
       TEMP.(C2F←⎕NEW'Button'(('Caption' 'C->F')
                               ('Posn'(40 70))))
       TEMP.(Q←⎕NEW'Button'(('Caption' '&Quit')
                            ('Posn'(70 30))('Size'(⍬ 40))
                            ('Cancel' 1)))

       TEMP.(S←⎕NEW'Scroll'(⊂('Range' 101)))

       TEMP.MB.M.C.onSelect←'SET_C'
       TEMP.MB.M.F.onSelect←'SET_F'
       TEMP.F2C.onSelect←'f2c'
       TEMP.F.onGotFocus←'SET_DEF'
       TEMP.C2F.onSelect←'c2f'
       TEMP.C.onGotFocus←'SET_DEF'
       TEMP.onClose←'QUIT'
       TEMP.Q.onSelect←'QUIT'
       TEMP.S.onScroll←'c2f_scroll'

       ⎕DQ'TEMP'
     ∇
```

For brevity, only a couple of the callback functions are shown here.

```
     ∇ f2c
[1]    TEMP.C.Value←(TEMP.F.Value-32)×5÷9
     ∇

     ∇ c2f_scroll MSG
[1]   ⍝ Callback for Centigrade input via scrollbar
[2]    TEMP.C.Value←101-4⊃MSG
[3]    c2f
     ∇
```



Writing Classes based on the Dyalog GUI

You may create user-defined Classes based upon Dyalog GUI objects as illustrated by the Temperature Converter Class which is listed overleaf.

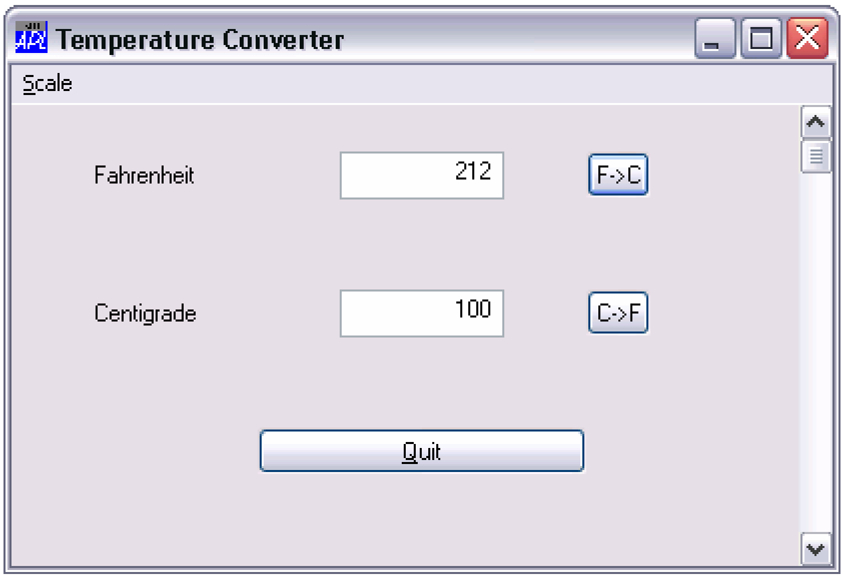
Temperature Converter Class

To base a Class on a Dyalog GUI object, you specify the *name* of the object as its Base Class. For example, the Temperature Converter is based upon a Form:

```
      :Class Temp: 'Form'
```

Being based upon a top-level GUI object, the Temperature Converter may be used as follows:

```
      T1←⎕NEW Temp(⊂'Posn'(68 50))
```



Temperature Converter Example

```
:Class Temp: 'Form'
    ∇ Make pv;TITLE
      :Access Public
      TITLE←'Temperature Converter'
      :Implements Constructor :Base (⊂'Caption' TITLE),pv,
                                     ⊂('Size' (30 40))
      MB←⎕NEW⊂'MenuBar'
      MB.(M←⎕NEW'Menu'(,⊂'Caption' '&Scale'))
      MB.M.(C←⎕NEW'MenuItem'(('Caption' '&Centigrade')
                              ('Checked' 1)))
      MB.M.(F←⎕NEW'MenuItem'(,⊂('Caption' '&Fahrenheit')))
      LF←⎕NEW'Label'(('Caption' 'Fahrenheit')
                     ('Posn'(10 10)))
      F←⎕NEW'Edit'(('Posn'(10 40))('Size'(⍬ 20))
                   ('FieldType' 'Numeric'))
      LC←⎕NEW'Label'(('Caption' 'Centigrade')
                     ('Posn'(40 10)))
      C←⎕NEW'Edit'(('Posn'(40 40))('Size'(⍬ 20))
                   ('FieldType' 'Numeric'))
      F2C←⎕NEW'Button'(('Caption' 'F->C')('Posn'(10 70))
                       ('Default' 1))
      C2F←⎕NEW'Button'(('Caption' 'C->F')('Posn'(40 70)))
      Q←⎕NEW'Button'(('Caption' '&Quit')('Posn'(70 30))
                     ('Size'(⍬ 40))('Cancel' 1))
      S←⎕NEW'Scroll'(⊂('Range' 101))
      MB.M.C.onSelect←'SET_C'
      MB.M.F.onSelect←'SET_F'
      F2C.onSelect←'f2c'
      F.onGotFocus←'SET_DEF'
      C2F.onSelect←'c2f'
      C.onGotFocus←'SET_DEF'
      onClose←'QUIT'
      Q.onSelect←'QUIT'
      S.onScroll←'c2f_scroll'
    ∇

    ∇ f2c
      C.Value←(F.Value-32)×5÷9
    ∇
    ∇ c2f
      F.Value←32+C.Value×9÷5
    ∇
    ∇ c2f_scroll MSG
      ⍝ Callback for Centigrade input via scrollbar
      C.Value←101-4⊃MSG
      c2f
    ∇
```

```
        ∇ f2c_scroll Msg
          ⍝ Callback for Fahrenheit input via scrollbar
          F.Value←213-4⊃Msg
          f2c
        ∇
        ∇ Quit
          Close
        ∇
        ∇ SET_DEF MSG
          (⊃MSG).Default←1
        ∇
        ∇ SET_C
          ⍝ Sets the scrollbar to work in Centigrade
          S.Range←101
          S.onScroll←'c2f_scroll'
          MB.M.C.Checked←1
          MB.M.F.Checked←0
        ∇
        ∇ SET_F
          ⍝ Sets the scrollbar to work in Fahrenheit
          S.Range←213
          S.onScroll←'f2c_scroll'
          MB.M.F.Checked←1
          MB.M.C.Checked←0
        ∇
:EndClass ⍝ Temp
```

Notice that the `:Implements Constructor` statement of its Constructor `Make`:

```
:Implements Constructor :Base (⊂'Caption' TITLE),pv,
                              ⊂('Size' (30 40))
```

passes on the application-specific property list (`pv`) given as its argument, but (in this case) specifies Caption and Size as well. The order in which the properties are specified in the `:Base` call ensures that the former will act as a default (and be overriden by an application-specific Caption requested in `pv`), whereas the specied Size of`(30 40)` will override whatever value of Size is requested by the host application in `pv`.

Other Instances can co-exist with the first:

```
        T2←⎕NEW Temp(('Caption' 'My Application')
                     ('Posn'(10 10))
```

Dual Class Example

The Dual Class example is based upon the example used to illustrate how you may build an ActiveX Control in Dyalog APL (see Chapter 13), but re-engineered as a internal Dyalog APL Class. The full listing of the Dual Class script is provided overleaf.

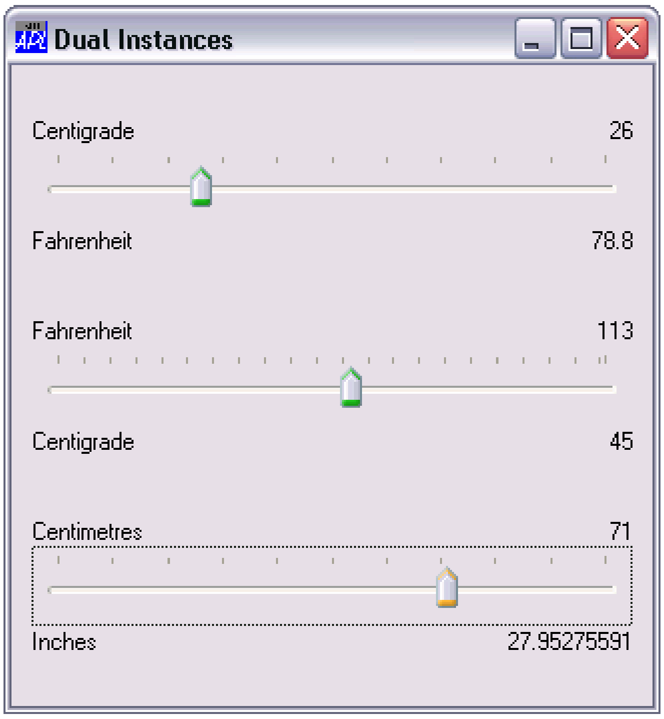
This version of Dual is based upon a SubForm:

```
      :Class Dual: 'SubForm'
```

The Dual Control requires a GUI parent but several Instances can co-exist, quite independently, in the same parent.

For example, function `RUN` creates a Form and 3 Instances of Dual; one to convert Centigrade to Fahrenheit, one to convert Fahrenheit to Centigrade, and the third to convert centimetres to inches.

```
      ∇ RUN;F;D1PROPS;D2PROPS;D3PROPS
[1]
[2]     F←⎕NEW'Form'(('Caption' 'Dual Instances')
                     ('Coord' 'Pixel')('Size'(320 320)))
[3]
[4]     D1PROPS←('Caption1' 'Centigrade')
                 ('Caption2' 'Fahrenheit')
[5]     D1PROPS,←('Intercept' 32)('Gradient'(9÷5))
                  ('Value1' 0)('Range'(0 100))
[6]     F.D1←F.⎕NEW Dual(('Coord' 'Pixel')('Posn'(10 10))
                        ('Size'(100 300)),D1PROPS)
[7]
[8]     D2PROPS←('Caption1' 'Fahrenheit')
                 ('Caption2' 'Centigrade')
[9]     D2PROPS,←('Intercept'(-32×5÷9))('Gradient'(5÷9))
                  ('Value1' 0)('Range'(0 212))
[10]    F.D2←F.⎕NEW Dual(('Coord' 'Pixel')('Posn'(110 10))
                        ('Size'(100 300)),D2PROPS)
[11]
[12]    D3PROPS←('Caption1' 'Centimetres')
                 ('Caption2' 'Inches')
[13]    D3PROPS,←('Intercept' 0)('Gradient'(÷2.54))
                  ('Value1' 0)('Range'(0 100))
[14]    F.D3←F.⎕NEW Dual(('Coord' 'Pixel')('Posn'(210 10))
                        ('Size'(100 300)),D3PROPS)
[15]
[16]    ⎕DQ'F'
      ∇
```



Dual's Constructor `Make` first splits its constructor arguments into those that apply to the Dual Class itself, and those that apply to the SubForm. Its `:Implements Constructor` statement then passes these on to the Base Constructor, together with an appropriate setting for EdgeStyle.

```
:Implements Constructor :Base BaseArgs,
                            ⊂'EdgeStyle' 'Dialog'
```

Dual Class Example

```
:Class Dual: 'SubForm'
    :Include GUITools
    :Field Private _Caption1←''
    :Field Private _Caption2←''
    :Field Private _Value1←0
    :Field Private _Value2←0
    :Field Private _Range←0
    :Field Private _Intercept←0
    :Field Private _Gradient←1
    :Field Private _Height←40

    ∇ Create args;H;W;POS;SH;CH;Y1;Y2;BaseArgs;MyArgs;
                Defaults
      :Access Public
      MyArgs BaseArgs←SplitNV args
      :Implements Constructor :Base BaseArgs,
                                   ⊂'EdgeStyle' 'Dialog'
      ExecNV_ MyArgs ⍝ Set Flds named _PropertyName MyArgs
      Coord←'Pixel'
      H W←Size
      POS←2↑⌊0.5×0⌈(H-_Height)
      CH←⊃GetTextSize'W'
      'Slider'⎕WC'TrackBar'POS('Size'_Height W)
      Slider.(Limits AutoConf)←_Range 0
      Slider.(TickSpacing TickAlign)←10 'Top'
      Slider.onThumbDrag←'ChangeValue'
      Slider.onScroll←'ChangeValue'
      Y1←POS[1]-CH+1
      Y2←POS[1]+_Height+1
      'Caption1_'⎕WC'Text'_Caption1(Y1,0)('AutoConf' 0)
      'Caption2_'⎕WC'Text'_Caption2(Y2,0)('AutoConf' 0)
      'Value1_'⎕WC'Text'(⍕_Value1)(Y1,W)('HAlign' 2)
                        ('AutoConf' 0)
      CalcValue2
      'Value2_'⎕WC'Text'(⍕_Value2)(Y2,W)('HAlign' 2)
              ('AutoConf' 0)
      onConfigure←'Configure'
    ∇

    :Property Caption1, Caption2
    :Access Public
        ∇ R←Get arg
          R←(arg.Name,'_')⎕WG'Text'
        ∇
        ∇ Set arg
          (arg.Name,'_')⎕WS'Text'arg.NewValue
        ∇
    :EndProperty
    :Property Value1
    :Access Public
        ∇ R←Get
          R←_Value1
        ∇
        ∇ Set arg
          ⎕NQ'Slider' 'Scroll' 0 arg.NewValue
        ∇
    :EndProperty

    :Property Intercept, Gradient, Range, Height, Value2
    :Access Public
        ∇ R←Get arg
          R←⍎'_',arg.Name
        ∇
    :EndProperty

    ∇ CalcValue2
      _Value2←_Intercept+_Gradient×_Value1
    ∇

    ∇ ChangeValue MSG
      ⍝ Callback for ThumbDrag and Scroll
      _Value1←⊃¯1↑MSG
      CalcValue2
      Value1_.Text←⍕_Value1
      Value2_.Text←⍕_Value2
    ∇

    ∇ Configure MSG;H;W;POS;CH;Y1;Y2
      2 ⎕NQ MSG
      H W←Size
      POS←2↑⌊0.5×(H-_Height)
      CH←⊃GetTextSize'W'
      Slider.(Posn Size)←POS(_Height W)
      Y1←POS[1]-CH+1
      Y2←POS[1]+_Height+1
      Caption1_.Points←1 2⍴Y1,0
      Caption2_.Points←1 2⍴Y2,0
      Value1_.Points←1 2⍴Y1,W
      Value1_.Points←1 2⍴Y2,W
    ∇

:EndClass ⍝ Dual
```

Writing Classes based on .Net Types

In Version 11.0 you may create user-defined Classes based upon .Net Classes. This feature supercedes the NetType object through which this was achieved in previous Versions of Dyalog APL.

To base a Class on a .Net Type, you simply specify the .Net Type *as* its Base Class. However, you must also specify the .Net search path with one or more `:Using` statements.

For example, the following Class called `APLGreg` derives from the .Net Type GregorianCalendar which is located in the System.Globalization .Net namespace.

```
:Class APLGreg: GregorianCalendar
:Using System.Globalization
...
:EndClass
```

Exporting the Class

Unlike other Classes, a Class that derives from a .Net Type must be **exported to a .Net Assembly** before it can be used. It must be turned into a fully fledged .Net Type before you can access it through .Net.

You may either export your Class (or Classes) to a named Assembly file (DLL) on disk, or you may take advantage of the new Version 11.0 feature and export it to memory. This is done using the *Export to Memory* menu item on the Session File menu. Export to Memory builds a temporary in-memory .Net Assembly and is intended to speed the development cycle. Note however that in order to use the Class in a live application, it will be necessary to save it to file.

Using the Class

If you have exported the Class to a Microsoft .Net Assembly (dll), you must specify the correct .Net search path to locate the file using `⎕USING`.

However, if you have exported the Class to memory (using *Export to Memory*), it is not necessary to set `⎕USING`.

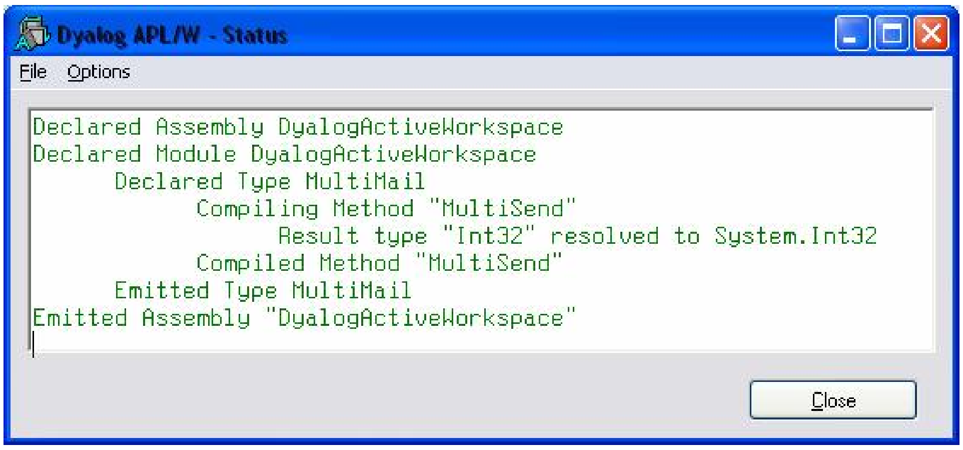
Example of a Class based on a .Net Type

The following example illustrates an APL Class that is based upon the .Net Type MailMessage.

```
:Class MultiMail : MailMessage
:Using System
:Using System.Web.Mail,system.web.dll
⍝ Adds "MultiSend" method to System.Web.Mail.MailMessage
⍝ "MultiSend" sends one message to each To address (separa
ted by ";")
⍝ So that each recipient sees self as sole addressee

    ⎕ML←0

    ∇ r←MultiSend;to;all;t
      :Access Public
      :Signature Int32←MultiSend
      r←⍴to←1↓¨(';'=to)⊂to←';',all←To
      :For t :In to
          To←t
          SmtpMail.Send ⎕THIS
      :EndFor
      To←all
    ∇

:EndClass
```

The `MultiMail` Class adds a `MultiSend` method to the basic System.Web.Mail.MailMessage Type. `MultiSend` sends a separate message to each of the recipients in the semi-colon separated address list. The result is the number of recipients.

To use the Class, you must first export it as a .Net Assembly (in this case, using the Export to Memory menu item on the Session File menu).



```
Declared Assembly DyalogActiveWorkspace
Declared Module DyalogActiveWorkspace
      Declared Type MultiMail
            Compiling Method "MultiSend"
                  Result type "Int32" resolved to System.Int32
            Compiled Method "MultiSend"
      Emitted Type MultiMail
Emitted Assembly "DyalogActiveWorkspace"
```

Note that when using a Class that has been exported to memory, it is not necessary to set ⎕USING.

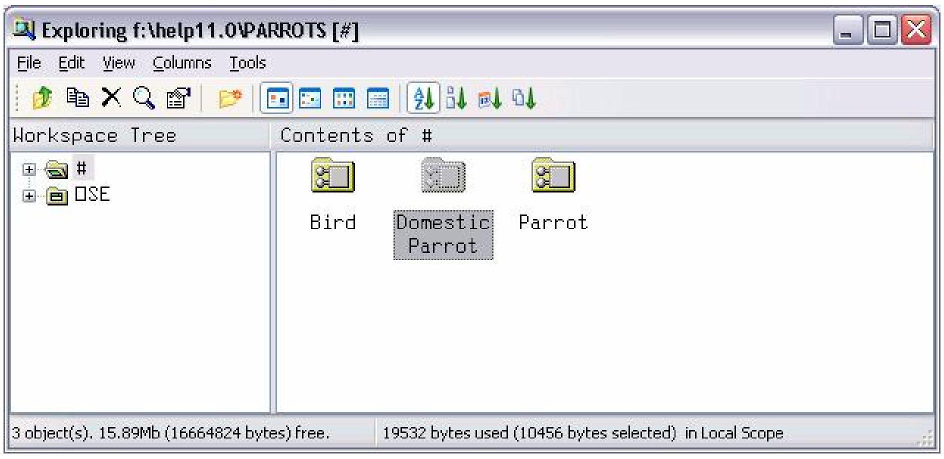
```
      z←⎕NEW MultiMail
      z.To←'mkrom@dyalog.com;mkrom@insight.dk'
      z.Subject←'hello'
      z.From←'mkrom@dyalog.com'
      z.Body←'????'
      z.MultiSend
2
```

The following points are noteworthy:

1. The `:Signature` statement in function `MultiSend` defines the signature of the method. It specifies that MultiSend takes no argument and returns a result of Type Int32 (which will be resolved to System.Int32 via `⎕USING`)
2. Of the two `:Using` statements, the first one is required to resolve the reference to Int32 to System.Int32.
3. The second `:Using` statement is required to resolve the references to MailMessage (in the `:Class` statement) and SmtpMail (in `MultiSend`).

Browsing Classes

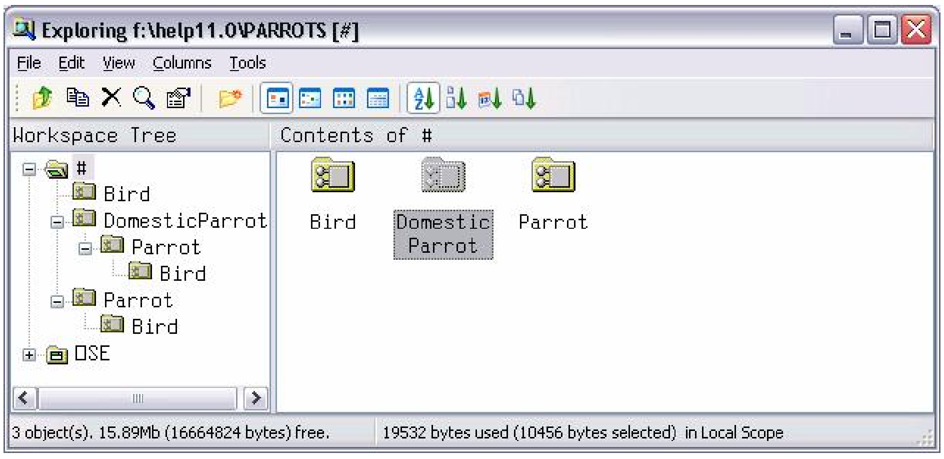
Classes are represented by icons. The picture below shows 3 classes: `Bird`, `Parrot` and `DomesticParrot`.



Exploring f:\help11.0\PARROTS [#]

3 object(s). 15.89Mb (16664824 bytes) free. 19532 bytes used (10456 bytes selected) in Local Scope

If you open Class nodes in the left-hand pane, the Explorer shows the Class hierarchy. In this example, `DomesticParrot` is based upon `Parrot` which in turn is based upon `Bird`.

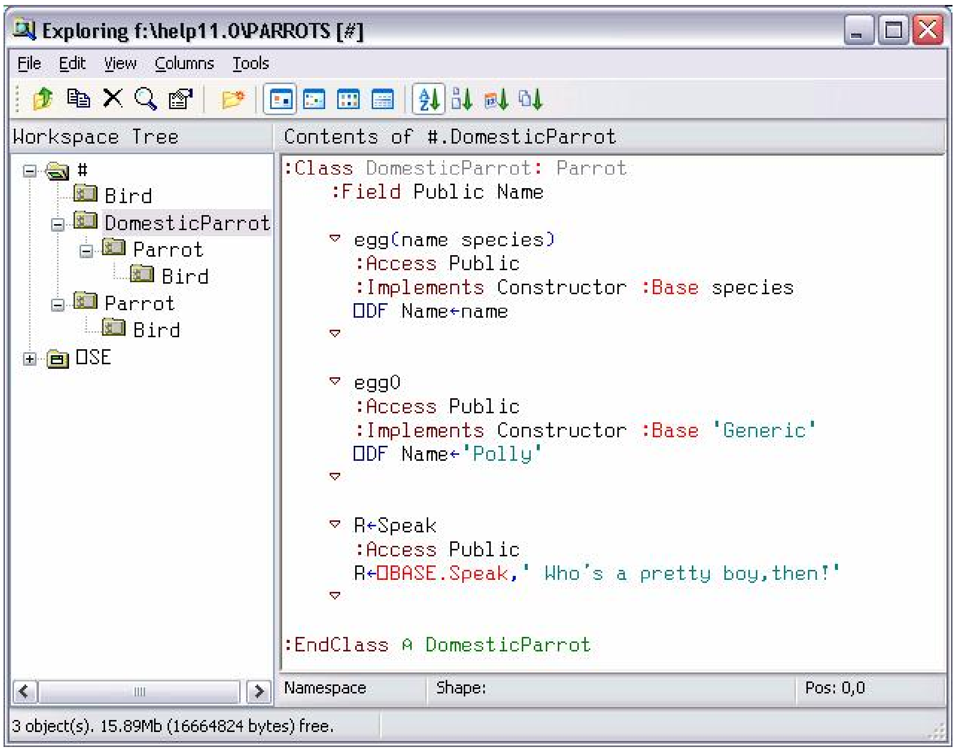


Exploring f:\help11.0\PARROTS [#]

3 object(s). 15.89Mb (16664824 bytes) free. 19532 bytes used (10456 bytes selected) in Local Scope

Browsing Class Scripts

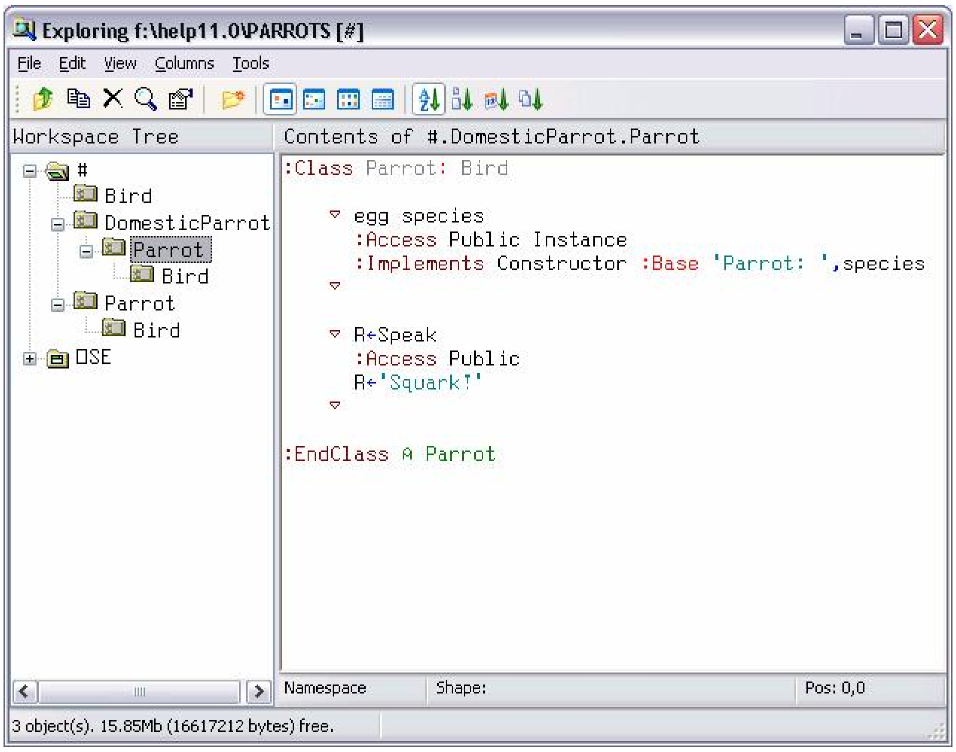
Selecting `DomesticParrot` in the left-hand pane brings up its Class Script in the right-hand pane.



Exploring f:\help11.0\PARROTS [#]
File Edit View Columns Tools
Workspace Tree
#
Bird
DomesticParrot
Parrot
Bird
Parrot
Bird
⎕SE
Contents of #.DomesticParrot
Namespace
Shape:
Pos: 0,0
3 object(s). 15.89Mb (16664824 bytes) free.

```
:Class DomesticParrot: Parrot
    :Field Public Name

    ∇ egg(name species)
      :Access Public
      :Implements Constructor :Base species
      ⎕DF Name←name
    ∇

    ∇ egg0
      :Access Public
      :Implements Constructor :Base 'Generic'
      ⎕DF Name←'Polly'
    ∇

    ∇ R←Speak
      :Access Public
      R←⎕BASE.Speak,' Who's a pretty boy,then!'
    ∇

:EndClass ⍝ DomesticParrot
```

… and selecting `Parrot` in the left-hand pane brings up the Class Script for `Parrot`.

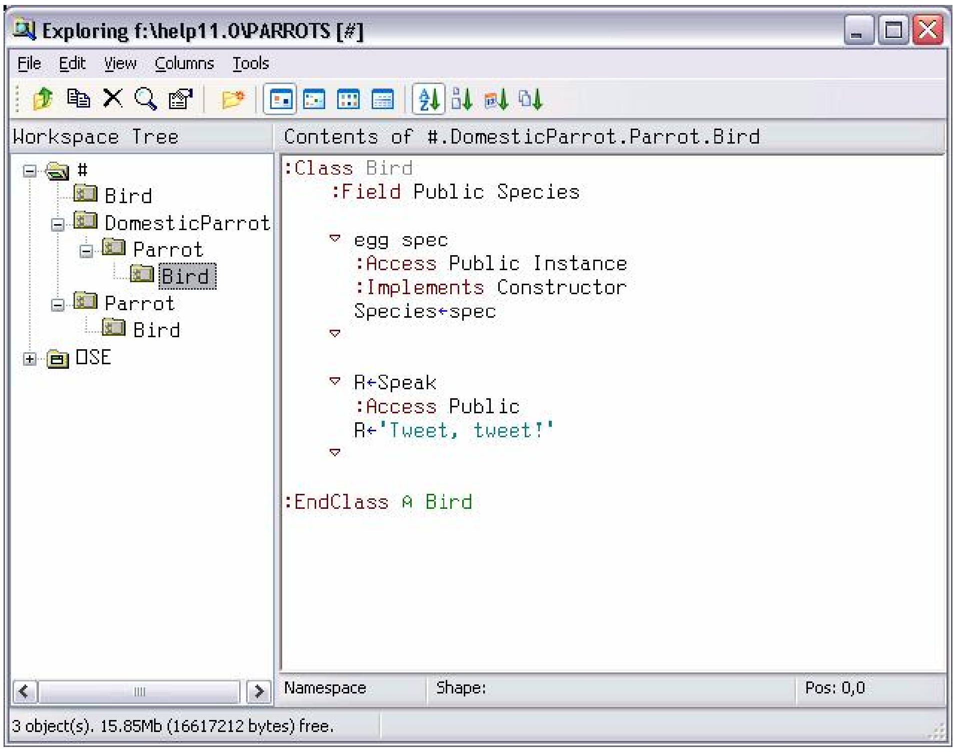


Contents of #.DomesticParrot.Parrot

```
:Class Parrot: Bird

    ∇ egg species
      :Access Public Instance
      :Implements Constructor :Base 'Parrot: ',species
    ∇

    ∇ R←Speak
      :Access Public
      R←'Squark!'
    ∇

:EndClass ⍝ Parrot
```

3 object(s). 15.85Mb (16617212 bytes) free.

… and finally, selecting `Bird` in the left-hand pane brings up the Class Script for `Bird`.


Exploring f:\help11.0\PARROTS [#]
File
Edit
View
Columns
Tools
Workspace Tree
#
Bird
DomesticParrot
Parrot
Bird
Parrot
Bird
⎕SE
Contents of #.DomesticParrot.Parrot.Bird
Namespace
Shape:
Pos: 0,0
3 object(s). 15.85Mb (16617212 bytes) free.

```
:Class Bird
    :Field Public Species

    ∇ egg spec
      :Access Public Instance
      :Implements Constructor
      Species←spec
    ∇

    ∇ R←Speak
      :Access Public
      R←'Tweet, tweet!'
    ∇

:EndClass ⍝ Bird
```

CHAPTER 4

Language Enhancements

New and Improved Primitive Functions & Operators

New Primitive Functions & Operators

⌷	Index
⌷[ ]	Index with Axes
⍣	Power Operator

Improved Primitive Functions & Operators

∧	And, Lowest Common Multiple
∨	Or, Greatest Common Divisor

And, Lowest Common Multiple: *R←X∧Y*

Case 1: *X* and *Y* are boolean

.*R* is boolean is determined as follows:

X	*Y*	*R*
0	0	0
0	1	0
1	0	0
1	1	1

Note that the ASCII caret (^) will also be interpreted as an APL And (∧).

Example

```
      0 1 0 1 ∧ 0 0 1 1
0 0 0 1
```

Case 2: *X* and *Y* are numeric (non-boolean)

R is the lowest common multiple of *X* and *Y*.

Example

```
      15 1 2 7 ∧ 35 1 4 0
105 1 4 0
```

Or, Greatest Common Divisor: $R \leftarrow X \vee Y$

Case 1: *X* and *Y* are boolean

R is boolean and is determined as follows:

X	*Y*	*R*
0	0	0
0	1	1
1	0	1
1	1	1

Example

```
      0 0 1 1 ∨ 0 1 0 1
0 1 1 1
```

Case 2: *X* and *Y* are numeric (non-boolean)

R is the Greatest Common Divisor of *X* and *Y*.

Example

```
      15 1 2 7 ∨ 35 1 4 0
5 1 2 7
```

Index: R←{X}⌷Y

Dyadic case

X must be a scalar or vector of depth `≤2` of integers each `≥⎕IO`. *Y* may be any array. In general, the result *R* is similar to that obtained by square-bracket indexing in that:

```
      (I J ... ⌷ Y) ≡ Y[I;J;...]
```

The length of left argument *X* must be equal to the rank of right argument *Y*.

Note that in common with square-bracket indexing, items of the left argument *X* may be of any rank and that the shape of the result is the concatenation of the shapes of the items of the left argument:

```
      (⍴X⌷Y) ≡ ↑,/⍴¨X
```

Index is sometimes referred to as *squad indexing*.

Note that index may be used with selective specification.
`⎕IO` is an implicit argument of index.

Examples

```
      ⎕IO←1

      VEC←111 222 333 444
      3⌷VEC
333
      (⊂4 3)⌷VEC
444 333
      (⊂2 3⍴3 1 4 1 2 3)⌷VEC
333 111 444
111 222 333

            ⎕←MAT←10⊥¨⍳3 4
11 12 13 14
21 22 23 24
31 32 33 34
      2 1⌷MAT
21
      3(2 1)⌷MAT
32 31
      (2 3)1⌷MAT
21 31
      (2 3)(,1)⌷MAT
21
31
```

```
      ⍴(2 1⍴1)(3 4⍴2)⌷MAT
2 1 3 4
      ⍴⍬ ⍬⌷MAT
0 0
      (3(2 1)⌷MAT)←0 ⋄ MAT     ⍝ Selective assignment.
11 12 13 14
21 22 23 24
 0  0 33 34
```

Monadic case

If `Y` is an array, `Y` is returned.

If `Y` is a ref to an instance of a Class with a Default property, all elements of the Default property are returned. For example, if `Item` is the default property of `MyClass`, and `imc` is an Instance of `MyClass`, then by definition:

```
      imc.Items≡⌷imc
```

Version 11.0 issues a `NONCE ERROR` if the Default Property is Keyed, because in this case APL has no way to determine the list of all the elements. A future version will probably introduce a way for a class to define an ordered "key set" for a Keyed property, at which point monadic squad will be extended to return the corresponding elements.

Note that the *values* of the index set are obtained or assigned by calls to the corresponding PropertyGet and PropertySet functions. Furthermore, if there is a sequence of primitive functions to the left of the Index function, that operate on the index set itself (functions such as dyadic `⍴`, `↑`, `↓`, `⊃`) as opposed to functions that operate on the *values* of the index set (functions such as `+`, `⌈`, `⌊`, `⍴¨`), calls to the PropertyGet and PropertySet functions are deferred until the required index set has been completely determined. The full set of functions that cause deferral of calls to the PropertyGet and PropertySet functions is the same as the set of functions that applies to selective specification.

If for example, `CompFile` is an Instance of a Class with a Default Numbered Property, the expression:

```
      1↑⌽⌷CompFile
```

would only call the PropertyGet function (for `CompFile`) once, to get the value of the last element.

Note that similarly, the expression

```
      10000⍴⌷CompFile
```

would call the PropertyGet function 10000 times, on repeated indices if CompFile has less than 10000 elements. The deferral of access function calls is intended to be an optimisation, but can have the opposite effect. You can avoid unnecessary repetitive calls by assigning the result of ⌷ to a temporary variable.

Index with Axes: R←{X}⌷[K]Y

X must be a scalar or vector of depth ≤2, of integers each ≥⎕IO. Y may be any array. K is a simple scalar of vector specifying axes of Y. The length of K must be the same as the length of X:

```
      (⍴,X) ≡ ⍴,K
```

In general, the result R is similar to that obtained by square-bracket indexing with elided subscripts. Items of K distribute items of X along the axes of Y. For example:

```
      I J ⌷[1 3] Y  ←→  Y[I;;J]
```

Note that index with axis may be used with selective specification. ⎕IO is an implicit argument of index with axis..

Examples

```
      ⎕IO←1

      ⎕←CUBE←10⊥¨⍳2 3 4
111 112 113 114
121 122 123 124
131 132 133 134

211 212 213 214
221 222 223 224
231 232 233 234

      2⌷[1]CUBE
211 212 213 214
221 222 223 224
231 232 233 234

      2⌷[3]CUBE
112 122 132
212 222 232
```

```
      CUBE[;;2] ≡ 2⌷[3]CUBE
1
      (1 3)4⌷[2 3]CUBE
114 134
214 234

      CUBE[;1 3;4] ≡ (1 3)4⌷[2 3]CUBE
1
      (2(1 3)⌷[1 3]CUBE)←0 ⋄ CUBE  ⍝ Selective assignment.
111 112 113 114
121 122 123 124
131 132 133 134

  0 212   0 214
  0 222   0 224
  0 232   0 234
```

Power Operator: {R}←{X}(f⍣g)Y

If right operand *g* is a numeric integer scalar, power applies its left operand function `f` cumulatively *g* times to its argument. In particular, *g* may be boolean 0 or 1 for conditional function application.

If right operand *g* is a scalar-boolean-returning dyadic *function*, then left operand function *f* is applied repeatedly **until** `(Y g f Y)` or until a strong interrupt occurs. In particular, if *g* is `=` or `≡`, the result is sometimes termed a *fixpoint* of *f*.

If a left argument *X* is present, it is bound as left argument to left operand function *f*:

```
X (f ⍣ g) Y → (X∘f ⍣ g) Y
```

A *negative* right operand *g* applies the *inverse* of the operand function *f*, `(|g)` times. In this case, *f* may be a primitive function or an expression of primitive functions combined with primitive operators:

`∘`	compose
`¨`	each
`∘.`	outer product
`⍨`	commute
`[]`	axis
`\`	scan
`⍣`	power

Defined, dynamic and some primitive functions do not have an inverse. In this case, a negative argument *g* generates `DOMAIN ERROR`.

Examples

```
      (,∘⊂∘,⍣(1=≡,vec))vec      ⍝ ravel-enclose if simple.

      a b c←1 0 1{(⊂⍣⍺)⍵}¨abc   ⍝ enclose first and last.

      cap←{(⍺⍺⍣⍺)⍵}             ⍝ conditional application.

      a b c←1 0 1⊂cap¨abc       ⍝ enclose first and last.

      succ←1∘+                  ⍝ successor function.

      (succ⍣4)10                ⍝ fourth successor of 10.
14
      (succ⍣¯3)10               ⍝ third predecessor of 10.
7
      1+∘÷⍣=1                   ⍝ fixpoint: golden mean.
1.618033989

      f←(32∘+)∘(×∘1.8)          ⍝ Fahrenheit from Celsius.
      f 0 100
32 212

      c←f⍣¯1                    ⍝ c is Inverse of f.
      c 32 212                  ⍝ Celsius from Fahrenheit.
0 100

      invs←{(⍺⍺⍣¯1)⍵}           ⍝ inverse operator.

      +\invs 1 3 6 10           ⍝ scan inverse.
1 2 3 4

      2∘⊥invs 9                 ⍝ decode inverse.
1 0 0 1

      dual←{⍵⍵⍣¯1 ⍺⍺ ⍵⍵ ⍵}       ⍝ dual operator.

      mean←{(+/⍵)÷⍴⍵}           ⍝ mean function.

      mean dual⍟ 1 2 3 4 5      ⍝ geometric mean.
2.605171085

      +/dual÷ 1 2 3 4 5         ⍝ parallel resistance.
0.4379562044

      mean dual(×⍨)1 2 3 4 5    ⍝ root-mean-square.
3.31662479

      ⍉dual↑ 'hello' 'world'    ⍝ vector transpose.
 hw  eo  lr  ll  od
```

New and Improved System Functions & Commands

New System Functions & Commands

⎕BASE	Base Class
⎕CLASS	Class
⎕DF	Display Form
⎕FIX	Fix Script
⎕INSTANCES	Instances
⎕NEW	New Instance
⎕SRC	Source
⎕THIS	This Space

Improved System Functions & Commands

⎕ED	Edit Object
⎕NC	Name Class
⎕NL	Name List
⎕PP	Print Precision
⎕WX	Window Expose
)ED	Edit Object

Base Class: R←⎕BASE.Y

⎕BASE is used to access the base class implementation of the name specified by Y.

Y must be the name of a Public member (Method, Field or Property) that is provided by the Base Class of the current Class or Instance.

⎕BASE is typically used to call a method in the Base Class which has been *superseded* by a Method in the current Class.

Note that ⎕BASE.Y is *special syntax* and any direct reference to ⎕BASE on its own or in any other context, is meaningless and causes SYNTAX ERROR.

In the following example, Class DomesticParrot derives from Class Parrot and supersedes its Speak method. DomesticParrot.Speak calls the Speak method in its Base Class Parrot, via ⎕BASE.

```
:Class Parrot: Bird
    ∇ R←Speak
      :Access Public
      R←'Squark!'
    ∇
:EndClass ⍝ Parrot

:Class DomesticParrot: Parrot
    ∇ R←Speak
      :Access Public
      R←⎕BASE.Speak,' Who's a pretty boy,then!'
    ∇
:EndClass ⍝ DomesticParrot

      Maccaw←⎕NEW Parrot
      Maccaw.Speak
Squark!

      Polly←⎕NEW DomesticParrot
      Polly.Speak
Squark! Who's a pretty boy,then!
```

Class: R←{X}⎕CLASS Y

Monadic Case

Monadic `⎕CLASS` returns a list of references to Classes and Interfaces that specifies the class hierarchy for the Class or Instance specified by `Y`.

`Y` must be a reference to a Class or to an Instance of a Class.

`R` is a vector or vectors whose items represent nodes in the Class hierarchy of `Y`. Each item of `R` is a vector whose first item is a Class reference and whose subsequent items (if any) are references to the Interfaces supported by that Class.

Example 1

This example illustrates a simple inheritance tree or Class hierarchy. There are 3 Classes, namely:

```
Animal
      Bird (derived from Animal)
            Parrot (derived from Bird)
```

```
:Class Animal
...
:EndClass ⍝ Animal

:Class Bird: Animal
...
:EndClass ⍝ Bird

:Class Parrot: Bird
...
:EndClass ⍝ Parrot

      ⎕CLASS Eeyore←⎕NEW Animal
  #.Animal
      ⎕CLASS Robin←⎕NEW Bird
  #.Bird    #.Animal
      ⎕CLASS Polly←⎕NEW Parrot
  #.Parrot    #.Bird    #.Animal

     ⎕CLASS¨ Parrot Animal
   #.Parrot    #.Bird    #.Animal       #.Animal
```

Example 2

The Penguin Class example (see page 76) illustrates the use of Interfaces.
In this case, the `Penguin` Class derives from `Animal` (as above) but additionally supports the `BirdBehaviour` and `FishBehaviour` Interfaces, thereby inheriting members from both.

```
      Pingo←⎕NEW Penguin
      ⎕CLASS Pingo
 #.Penguin  #.FishBehaviour  #.BirdBehaviour    #.Animal
```

Dyadic Case

If `X` is specified, `Y` must be a reference to an Instance of a Class and `X` is a reference to an Interface that is supported by Instance `Y` or to a Class upon which Instance `Y` is based.

In this case, `R` is a reference to the implementation of Interface `X` by Instance `Y`, or to the implementation of (Base) Class `X` by Instance `Y`,and is used as a *cast* in order to access members of `Y` that correspond to members of Interface of (Base) Class `X`.

Example 1:

Once again, the Penguin Class example (see page 76) is used to illustrate the use of Interfaces.

```
      Pingo←⎕NEW Penguin
      ⎕CLASS Pingo
 #.Penguin  #.FishBehaviour  #.BirdBehaviour    #.Animal

      (FishBehaviour ⎕CLASS Pingo).Swim
I can dive and swim like a fish
      (BirdBehaviour ⎕CLASS Pingo).Fly
Although I am a bird, I cannot fly
      (BirdBehaviour ⎕CLASS Pingo).Lay
I lay one egg every year
      (BirdBehaviour ⎕CLASS Pingo).Sing
Croak, Croak!
```

Example 2:

This example illustrates the use of dyadic `⎕CLASS` to cast an Instance to a lower Class and thereby access a member in the lower Class that has been superseded by another Class higher in the tree.

```
      Polly←⎕NEW DomesticParrot
      Polly.Speak
Squark! Who's a pretty boy,then!
```

Note that the `Speak` method invoked above is the `Speak` method defined by Class `DomesticParrot`, which supersedes the `Speak` methods of sub-classes `Parrot` and `Bird`.

You may use a cast to access the (superseded) `Speak` method in the sub-classes `Parrot` and `Bird`.

```
      (Parrot ⎕CLASS Polly).Speak
Squark!
      (Bird ⎕CLASS Polly).Speak
Tweet, tweet!
```

Display Form: R←⎕DF Y

⎕DF sets the *Display Form* of a namespace, a GUI object, a Class, or an Instance of a Class.

Y must be a simple character array that specifies the display form of a namespace. If defined, this array will be returned by the *format* functions and ⎕FMT instead of the default for the object in question. This also applies to the string that is displayed when the name is referenced but not assigned (the *default display*).

The result R is the previous value of the Display Form which initially is ⎕NULL.

```
      'F'⎕WC'Form'
      ⍕F
#.F
      ⍴⍕F
3
      ⎕FMT F
#.F
      ⍴⎕FMT F
1 3
      F ⍝ default display uses ⍕
#.F

       F.⎕DF 'Pete's Form'
       ⍕F
Pete's Form
       ⍴⍕F
11
       ⎕FMT F
Pete's Form
       ⍴⎕FMT F
1 11
```

Notice that ⎕DF will accept any character array, but ⎕FMT always returns a matrix.

```
      F.⎕DF 2 2 5⍴⎕A
      F
ABCDE
FGHIJ

KLMNO
PQRST
      ⍴⍕F
2 2 5
```

```
      ⍴⎕←⎕FMT F
ABCDE
FGHIJ

KLMNO
PQRST
5 5
```

Note that ⎕DF defines the Display Form statically, rather than dynamically.

```
      'F'⎕WC'Form' 'This is the Caption'
      F
#.F
      F.(⎕DF Caption)⍝ make current caption the display fo
rm
      F
This is the Caption
      F.Caption←'New Caption' ⍝ changing caption does not
change the display form
      F
This is the Caption
```

You may use the Constructor function to assign the Display Form to an Instance of a Class. For example:

```
:Class MyClass
    ∇ Make arg
      :Access Public
      :Implements Constructor
      ⎕DF arg
    ∇
:EndClass ⍝ MyClass

      PD←⎕NEW MyClass 'Pete'
      PD
Pete
```

It is possible to set the Display Form for the Root and for ⎕SE

```
      )CLEAR
clear ws
      #
#
      ⎕DF ⎕WSID
      #
CLEAR WS

      ⎕SE
⎕SE
      ⎕SE.⎕DF 'Session'
      ⎕SE
Session
```

Note that ⎕DF applies directly to the object in question and is not automatically applied in a hierarchical fashion.

```
      'X'⎕NS ''
      X
#.X

      'Y'X.⎕NS ''
      X.Y
#.X.Y
      X.⎕DF 'This is X'
      X
This is X

      X.Y
#.X.Y
```

Edit Object: {R}←{X}⎕ED Y

`⎕ED` invokes the Editor. `Y` is a simple character vector, a simple character matrix, or a vector of character vectors, containing the name(s) of objects to be edited. The optional left argument `X` is a character scalar or character vector with as many elements as there are names in `Y`. Each element of `X` specifies the type of the corresponding (new) object named in `Y`, where :

∇	function/operator
→	simple character vector
∊	vector of character vectors
−	character matrix
⊛	Namespace script
○	Class script
∘	Interface

If an object named in `Y` already exists, the corresponding type specification in `X` is ignored.

If `⎕ED` is called from the Session, it opens Edit windows for the object(s) named in `Y` and returns a null result. The cursor is positioned in the first of the Edit windows opened by `⎕ED`, but may be moved to the Session or to any other window which is currently open. The effect is almost identical to using `)ED`.

If `⎕ED` is called from a defined function or operator, its behaviour is different. On asynchronous terminals, and in Dyalog APL for DOS/386, the Edit windows are automatically displayed in "full-screen" mode (ZOOMED). In all implementations, the user is restricted to those windows named in `Y`. The user may not skip to the Session even though the Session may be visible

`⎕ED` terminates and returns a result ONLY when the user explicitly closes all the windows for the named objects. In this case the result contains the names of any objects which have been changed, and has the same structure as `Y`.

Fix Script: R←{X}⎕FIX Y

⎕FIX fixes a Class from the script specified by Y.

Y must be a vector of character vectors (scalars) that contains a well-formed Class script. If so, R is a reference to the new Class fixed by ⎕FIX.

The Class specified by Y may be named or unnamed.

If specified, X must be a numeric scalar numeric. If X is omitted or non-zero, and the Class script Y specifies a name (for the Class), ⎕FIX establishes that Class in the workspace.

If X is 0 or the Class specified by Y is unnamed, the Class is not established *per se*, although it will exist for as long as a reference to it exists.

In the first example, the Class specified by Y is *named* (MyClass) but the result of ⎕FIX is discarded. The end-result is that MyClass is established in the workspace as a Class.

```
      ⎕←⎕FIX ':Class MyClass' ':EndClass'
#.MyClass
```

In the second example, the Class specified by Y is *named* (MyClass) and the result of ⎕FIX is assigned to a different name (MYREF). The end-result is that a Class named MyClass is established in the workspace, and MYREF is a reference to it.

```
      MYREF←⎕FIX ':Class MyClass' ':EndClass'
      )CLASSES
MyClass MYREF
      ⎕NC'MyClass' 'MYREF'
9.4 9.4
      MYREF
#.MyClass
```

In the third example, the left-argument of 0 causes the named Class MyClass to be visible only via the reference to it (MYREF). It is there, but hidden.

```
      MYREF←0 ⎕FIX ':Class MyClass' ':EndClass'
      )CLASSES
MYREF
      MYREF
#.MyClass
```

The final example illustrates the use of un-named Classes.

```
      src←':Class' '∇Make n'
      src,←'Access Public' 'Implements Constructor'
      src,←'⎕DF n' '∇' ':EndClass'
      MYREF←⎕FIX src
      )CLASSES
MYREF
      MYINST←⎕NEW MYREF'Pete'
      MYINST
Pete
```

Instances: R←⎕INSTANCES Y

`⎕INSTANCES` returns a list all the current instances of the Class specified by `Y`.

`Y` must be a reference to a Class.
`R` is a vector of references to all existing Instances of Class `Y`.

Examples

This example illustrates a simple inheritance tree or Class hierarchy. There are 3 Classes, namely:

```
Animal
      Bird (derived from Animal)
            Parrot (derived from Bird)
```

```
:Class Animal
...
:EndClass ⍝ Animal

:Class Bird: Animal
...
:EndClass ⍝ Bird

:Class Parrot: Bird
...
:EndClass ⍝ Parrot

      Eeyore←⎕NEW Animal
      Robin←⎕NEW Bird
      Polly←⎕NEW Parrot

      ⎕INSTANCES Parrot
 #.[Parrot]
      ⎕INSTANCES Bird
 #.[Bird]  #.[Parrot]
      ⎕INSTANCES Animal
 #.[Animal]  #.[Bird]  #.[Parrot]

      Eeyore.⎕DF 'eeyore'
      Robin.⎕DF 'robin'
      Polly.⎕DF 'polly'
      ⎕INSTANCES Parrot
polly
      ⎕INSTANCES Bird
robin  polly
      ⎕INSTANCES Animal
eeyore  robin  polly
```

Name Classification: R←⎕NC Y

`Y` must be a simple character scalar, vector, matrix ,or vector of vectors that specifies a list of names. `R` is a simple numeric vector containing one element per name in `Y`.

Each element of `R` is the name class of the active referent to the object named in `Y`.

If `Y` is *simple*, a name class may be:

Name Class	**Description**
¯1	invalid name
0	unused name
1	Label
2	Variable
3	Function
4	Operator
9	Object (GUI, namespace, COM, .Net)

If `Y` is nested, a more precise analysis of name class is obtained whereby different types of functions (primitive, traditional defined functions, D-fns) are identified by a decimal extension. For example, defined functions have name class 3.1, D-fns have name class 3.2, and so forth. The complete set of name classification is as follows:

	Array (2)	**Functions (3)**	**Operators (4)**	**Namespaces (9)**
n.1	Variable	Traditional	Traditional	Created by `⎕NS`
n.2	Field	D-fns	D-ops	Instance
n.3	Property	Derived Primitive		
n.4				Class
n.5		N/A		Interface
n.6	External Shared	External		External Class
n.7				External Interface

In addition, values in `R` are negative to identify names of methods, properties and events that are inherited through the *class hierarchy* of the current class or instance.

Variable (Name-Class 2.1)

Conventional APL arrays have name-class 2.1.

```
      NUM←88
      CHAR←'Hello World'

      ⎕NC ↑'NUM' 'CHAR'
2 2

      ⎕NC 'NUM' 'CHAR'
2.1 2.1

      'MYSPACE'⎕NS ''
      MYSPACE.VAR←10
      MYSPACE.⎕NC'VAR'
2
      MYSPACE.⎕NC⊂'VAR'
2.1
```

Field (Name-Class 2.2)

Fields defined by APL Classes have name-class 2.2.

```
:Class nctest
    :Field Public pubFld
    :Field pvtFld

    ∇ r←NC x
      :Access Public
      r←⎕NC x
    ∇
...
:EndClass ⍝ nctest

      ncinst←⎕NEW nctest
```

The name-class of a Field, whether Public or Private, *viewed from a Method that is executing within the Instance Space,* is 2.2.

```
      ncinst.NC'pubFld' 'pvtFld'
2.2 2.2
```

Note that an internal Method sees both Public and Private Fields in the Class Instance. However, when viewed from *outside* the instance, only public fields are visible

```
      ⎕NC 'ncinst.pubFld' 'ncinst.pvtFld'
¯2.2 0
```

In this case, the name-class is negative to indicate that the name has been exposed by the class hierarchy, rather than existing in the associated namespace which APL has created to contain the instance. The same result is returned if `⎕NC` is executed inside this space:

```
      ncinst.⎕NC'pubFld' 'pvtFld'
¯2.2 0
```

Note that the names of Fields are reported as being *unused* if the argument to `⎕NC` is simple.

```
      ncinst.⎕NC 2 6⍴'pubFldpvtFld'
0 0
```

Property (Name-Class 2.3)

Properties defined by APL Classes have name-class 2.3.

```
:Class nctest
    :Field pvtFld←99

    :Property pubProp
    :Access Public
        ∇ r←get
          r←pvtFld
        ∇
    :EndProperty

    :Property pvtProp
        ∇ r←get
          r←pvtFld
        ∇
    :EndProperty

    ∇ r←NC x
      :Access Public
      r←⎕NC x
    ∇
...
:EndClass ⍝ nctest

      ncinst←⎕NEW nctest
```

The name-class of a Property, whether Public or Private, *viewed from a Method that is executing within the Instance Space,* is 2.3.

```
      ncinst.NC'pubProp' 'pvtProp'
2.3 2.3
```

Note that an internal Method sees both Public and Private Properties in the Class Instance. However, when viewed from *outside* the instance, only Public Properties are visible

```
      ⎕NC 'ncinst.pubProp' 'ncinst.pvtProp'
¯2.3 0
```

In this case, the name-class is negative to indicate that the name has been exposed by the class hierarchy, rather than existing in the associated namespace which APL has created to contain the instance. The same result is returned if ⎕NC is executed inside this space:

```
      ncinst.⎕NC 'pubProp' 'pvtProp'
¯2.3 0
```

Note that the names of Properties are reported as being *unused* if the argument to ⎕NC is simple.

```
      ncinst.⎕NC 2 6⍴'pubProppvtProp'
0 0
```

External Properties (Name-Class 2.6)

Properties exposed by external objects (.Net and COM and the APL GUI) have name-class ¯2.6.

```
      ⎕USING←'System'
      dt←⎕NEW DateTime (2006 1 1)
      dt.⎕NC 'Day' 'Month' 'Year'
¯2.6 ¯2.6 ¯2.6

      'ex' ⎕WC 'OLEClient' 'Excel.Application'
      ex.⎕NC 'Caption' 'Version' 'Visible'
¯2.6 ¯2.6 ¯2.6

     'f'⎕WC'Form'
      f.⎕NC'Caption' 'Size'
¯2.6 ¯2.6
```

Note that the names of such Properties are reported as being *unused* if the argument to ⎕NC is simple.

```
      f.⎕NC 2 7⍴'CaptionSize   '
0 0
```

Defined Functions (Name-Class 3.1)

Traditional APL defined functions have name-class 3.1.

```
      ∇ R←AVG X
[1]     R←(+/X)÷⍴X
      ∇
       AVG ⍳100
50.5

       ⎕NC'AVG'
3
       ⎕NC⊂'AVG'
3.1

       'MYSPACE'⎕NS 'AVG'
        MYSPACE.AVG ⍳100
50.5

       MYSPACE.⎕NC'AVG'
3
       ⎕NC⊂'MYSPACE.AVG'
3.1
```

Note that a function that is simply cloned from a defined function by assignment retains its name-class.

```
       MEAN←AVG
       ⎕NC'AVG' 'MEAN'
3.1 3.1
```

Whereas, the name of a function that amalgamates a defined function with any other functions has the name-class of a Derived Function, i.e. 3.3.

```
       VMEAN←AVG∘,
       ⎕NC'AVG' 'VMEAN'
3.1 3.3
```

D-Fns (Name-Class 3.2)

D-Fns (Dynamic Functions) have name-class 3.2

```
      Avg←{(+/⍵)÷⍴⍵}

      ⎕NC'Avg'
3
      ⎕NC⊂'Avg'
3.2
```

Derived Functions (Name-Class 3.3)

Derived Functions and functions created by naming a Primitive function have name-class 3.3.

```
      PLUS←+
      SUM←+/
      CUM←PLUS\
      ⎕NC'PLUS' 'SUM' 'CUM'
3.3 3.3 3.3
      ⎕NC 3 4⍴'PLUSSUM CUM '
3 3 3
```

Note that a function that is simply cloned from a defined function by assignment retains its name-class. Whereas, the name of a function that amalgamates a defined function with any other functions has the name-class of a Derived Function, i.e. 3.3.

```
     ∇ R←AVG X
[1]    R←(+/X)÷⍴X
     ∇

      MEAN←AVG
      VMEAN←AVG∘,
      ⎕NC'AVG' 'MEAN' 'VMEAN'
3.1 3.1 3.3
```

External Functions (Name-Class 3.6)

Methods exposed by the Dyalog APL GUI and COM and .NET objects have name-class ¯3.6. Methods exposed by External Functions created using ⎕NA and ⎕SH all have name-class 3.6.

```
      'F'⎕WC'Form'

      F.⎕NC'GetTextSize' 'GetFocus'
¯3.6 ¯3.6

      'EX'⎕WC'OLEClient' 'Excel.Application'
      EX.⎕NC 'Wait' 'Save' 'Quit'
¯3.6 ¯3.6 ¯3.6

      ⎕USING←'System'
      dt←⎕NEW DateTime (2006 1 1)
      dt.⎕NC 'AddDays' 'AddHours'
¯3.6 ¯3.6

       'beep'⎕NA'user32|MessageBeep i'

      ⎕NC'beep'
3
      ⎕NC⊂'beep'
3.6
      'xutils'⎕SH''
      )FNS
avx     box     dbr     getenv  hex     ltom    ltov    mt
ol      ss      vtol
      ⎕NC'hex' 'ss'
3.6 3.6
```

Operators (Name-Class 4.1)

Traditional Defined Operators have name-class 4.1.

```
      ∇FILTER∇
     ∇ VEC←(P FILTER)VEC   ⍝ Select from VEC those elts ..
[1]    VEC←(P¨VEC)/VEC     ⍝ for which BOOL fn P is true.
     ∇

      ⎕NC'FILTER'
4
      ⎕NC⊂'FILTER'
4.1
```

D-Ops (Name-Class 4.2

D-Ops (Dynamic Operators) have name-class 4.2.

```
pred←{⎕IO ⎕ML←1 3    ⍝ Partitioned reduction.
     ⊃⍺⍺/¨(⍺/⍳⍴⍺)⊂⍵
     }

      2 3 3 2 +pred ⍳10
3 12 21 19

      ⎕NC'pred'
4
      ⎕NC⊂'pred'
4.2
```

External Events (Name-Class 8.6)

Events exposed by Dyalog APL GUI objects, COM and .NET objects have name-class ¯8.6.

```
      f←⎕NEW'Form'('Caption' 'Dyalog GUI Form')

      f.⎕NC'Close' 'Configure' 'MouseDown'
¯8.6 ¯8.6 ¯8.6

      xl←⎕NEW'OLEClient'(⊂'ClassName' 'Excel.Application')
      xl.⎕NL -8
 NewWorkbook  SheetActivate  SheetBeforeDoubleClick ...

      xl.⎕nc 'SheetActivate' 'SheetCalculate'
¯8.6 ¯8.6
```

```
      ⎕USING←'System.Windows.Forms,system.windows.forms.dll'
      ⎕NC,⊂'Form'
9.6
      Form.⎕NL -8
 Activated  BackgroundImageChanged  BackColorChanged ...
```

Namespaces (Name-Class 9.1)

Plain namespaces created using ⎕NS have name-class 9.1.

```
      'MYSPACE' ⎕NS ''
      ⎕NC'MYSPACE'
9
      ⎕NC⊂'MYSPACE'
9.1
```

Note however that a namespace created by cloning, where the right argument to ⎕NS is a ⎕OR of a namespace, retains the name-class of the original space.

```
      'CopyMYSPACE'⎕NS ⎕OR 'MYSPACE'
      'CopyF'⎕NS ⎕OR 'F'⎕WC'Form'

      ⎕NC'MYSPACE' 'F'
9.1 9.2
      ⎕NC'CopyMYSPACE' 'CopyF'
9.1 9.2
```

The Name-Class of .Net namespaces (visible through ⎕USING) is also 9.1

```
      ⎕USING←''
      ⎕NC 'System' 'System.IO'
9.1 9.1
```

Instances (Name-Class 9.2)

Instances of Classes created using ⎕NEW, and GUI objects created using ⎕WC all have name-class 9.2.

```
      MyInst←⎕NEW MyClass
      ⎕NC'MyInst'
9
      ⎕NC⊂'MyInst'
9.2
      UrInst←⎕NEW ⎕FIX ':Class'  ':EndClass'
      ⎕NC 'MyInst' 'UrInst'
9.2 9.2
```

```
      'F'⎕WC 'Form'
      'F.B' ⎕WC 'Button'
      ⎕NC 2 3⍴'F  F.B'
9 9
      ⎕NC'F' 'F.B'
9.2 9.2
      F.⎕NC'B'
9
      F.⎕NC⊂,'B'
9.2
```

Instances of COM Objects whether created using `⎕WC` or `⎕NEW` also have name-class 9.2.

```
      xl←⎕NEW'OLEClient'(⊂'ClassName' 'Excel.Application')
      'XL'⎕WC'OLEClient' 'Excel.Application'
      ⎕nc'xl' 'XL'
9.2 9.2
```

The same is true of Instances of .Net Classes (Types) whether created using `⎕NEW` or `.New`.

```
      ⎕USING←'System'
      dt←⎕NEW DateTime (3↑⎕TS)
      DT←DateTime.New 3↑⎕TS
      ⎕NC 'dt' 'DT'
9.2 9.2
```

Note that if you remove the GUI component of a GUI object, using the Detach method, it reverts to a plain namespace.

```
      F.Detach
      ⎕NC⊂,'F'
9.1
```

Correspondingly, if you attach a GUI component to a plain namespace using the monadic form of `⎕WC`, it morphs into a GUI object

```
      F.⎕WC 'PropertySheet'
      ⎕NC⊂,'F'
9.2
```

Classes (Name-Class 9.4)

Classes created using the editor or `⎕FIX` have name-class 9.4.

```
      )ED ○MyClass

:Class MyClass
    ∇ r←NC x
      :Access Public Shared
      r←⎕NC x
    ∇
:EndClass ⍝ MyClass

      ⎕NC 'MyClass'
9
      ⎕NC⊂'MyClass'
9.4

      ⎕FIX ':Class UrClass'  ':EndClass'
      ⎕NC 'MyClass' 'UrClass'
9.4 9.4
```

Note that the name of the Class is visible to a Public Method in that Class, or an Instance of that Class.

```
      MyClass.NC'MyClass'
9
      MyClass.NC⊂'MyClass'
9.4
```

Interfaces (Name-Class 9.5)

Interfaces, defined by `:Interface ... :EndInterface` clauses, have name-class 9.5.

```
:Interface IGolfClub
:Property Club
    ∇ r←get
    ∇
    ∇ set
    ∇
:EndProperty

∇ Shank←Swing Params
∇

:EndInterface ⍝ IGolfClub

      ⎕NC 'IGolfClub'
9
      ⎕NC ⊂'IGolfClub'
9.5
```

External Classes (Name-Class 9.6)

External Classes (Types) .exposed by .Net have name-class 9.6.

```
      ⎕USING←'System' 'System.IO'

      ⎕NC 'DateTime' 'File' 'DirectoryInfo'
9.6 9.6 9.6
```

Note that referencing a .Net class (type) with `⎕NC`, fixes the name of that class in the workspace and obviates the need for APL to repeat the task of searching for and loading the class when the name is next used.

External Interfaces (Name-Class 9.7)

External Interfaces exposed by .Net have name-class 9.7.

```
      ⎕USING←'System.Web.UI,system.web.dll'

      ⎕NC 'IPostBackDataHandler' 'IPostBackEventHandler'
9.7 9.7
```

Note that referencing a .Net Interface with `⎕NC`, fixes the name of that Interface in the workspace and obviates the need for APL to repeat the task of searching for and loading the Interface when the name is next used.

New Instance: R←⎕NEW Y

`⎕NEW` creates a new instance of the Class or .Net Type specified by `Y`.

`Y` must be a 1- or 2-item scalar or vector. The first item is a reference to a Class or to a .Net Type, or a character vector containing the name of a Dyalog GUI object. The second item, if specified, contains the argument to be supplied to the Class or Type *Constructor*.

The result `R` is a reference to a new instance of Class or Type `Y`.

Class Example

```
:Class Animal
    ∇ Name nm
      :Access Public
      :Implements Constructor
      ⎕DF nm
    ∇
:EndClass ⍝ Animal

      Donkey←⎕NEW Animal 'Eeyore'
      Donkey
Eeyore
```

If `⎕NEW` is called with just a Class reference (i.e. without parameters for the Constructor), the default constructor will be called. A default constructor is defined by a *niladic* function with the `:Implements Constructor` attribute. For example, the `Animal` Class may be redefined as:

```
:Class Animal
    ∇ NoName
      :Access Public
      :Implements Constructor
      ⎕DF 'Noname'
    ∇
    ∇ Name nm
      :Access Public
      :Implements Constructor
      ⎕DF nm
    ∇
:EndClass ⍝ Animal

      Horse←⎕NEW Animal
      Horse
Noname
```

.Net Examples

```
      ⎕USING←'System' 'System.Web.Mail,System.Web.dll'
      dt←⎕NEW DateTime (2006 1 1)
      msg←⎕NEW MailMessage
      ⎕NC 'dt' 'msg' 'DateTime' 'MailMessage'
9.2 9.2 9.6 9.6
```

Note that **.Net Types** are accessed as follows.

If the name specified by the first item of `Y` would otherwise generate a `VALUE ERROR`, and `⎕USING` has been set, APL attempts to load the Type specified by `Y` from the .Net assemblies (DLLs) specified in `⎕USING.` If successful, the name specified by `Y` is entered into the SYMBOL TABLE with a name-class of `9.6`. Subsequent references to that symbol (in this case `DateTime`) are resolved directly and do not involve any assembly searching.

```
      F←⎕NEW ⊂'Form'
      F←⎕NEW'Form'(('Caption' 'Hello')('Posn' (10 10)))

      ⎕NEW'Form'(('Caption' 'Hello')('Posn' (10 10)))
#.[Form]
```

Name List: R←{X}⎕NL Y

`Y` must be a simple numeric scalar or vector containing one or more of the values for name-class See also the system function `⎕NC`.

`X` is optional. If present, it must be a simple character scalar or vector. `R` is a list of the names of active objects whose name-class is included in `Y` in standard sorted order.

If any element of `Y` is negative, `R` is a vector of character vectors. Otherwise, `R` is simple character matrix.

Furthermore, if `⎕NL` is being evaluated inside the namespace associated with a Class or an Instance of a Class, and any element of `Y` is negative, `R` includes the Public names exposed by the Base Class (if any) and all other Classes in the Class hierarchy.

If `X` is supplied, `R` contains only those names which begin with any character of `X`. Standard sorted order is the collation order of `⎕AV`.

If an element of *Y* is an integer, the names of all of the corresponding sub-name-classes are included in *R*. For example, if *Y* contains the value 2, the names of all variables (name-class 2.1), fields (2.2), properties (2.3) and external or shared variables (2.6) are obtained. Otherwise, only the names of members of the corresponding sub-name-class are obtained.

Examples:

```
      ⎕NL 2 3
A
FAST
FIND
FOO
V

      'AV' ⎕NL 2 3
A
V

      ⎕NL ¯9
 Animal  Bird  BirdBehaviour  Coin  Cylinder  DomesticParr
ot  Eeyore  FishBehaviour  Nickel  Parrot  Penguin  Polly
 Robin
      ⎕NL ¯9.3 ⍝ Instances
 Eeyore  Nickel  Polly  Robin
      ⎕NL ¯9.4 ⍝ Classes
 Animal  Bird  Coin  Cylinder  DomesticParrot  Parrot  Pen
guin
      ⎕NL ¯9.5 ⍝ Interfaces
 BirdBehaviour  FishBehaviour
```

`⎕NL` can also be used to explore Dyalog GUI Objects, .Net types and COM objects.

Dyalog GUI Objects

`⎕NL` may be used to obtain lists of the Methods, Properties and Events provided by Dyalog APL GUI Objects.

```
     'F' ⎕WC 'Form'
      F.⎕NL -2 ⍝ Properties
 Accelerator  AcceptFiles  Active  AlphaBlend  AutoConf  B
order  BCol  Caption ...

      F.⎕NL -3 ⍝ Methods
 Animate  ChooseFont  Detach  GetFocus  GetTextSize  ShowS
IP  Wait

      F.⎕NL -8 ⍝ Events
Close  Create  DragDrop  Configure  ContextMenu  DropFiles
  DropObjects  Expose  Help ...
```

.Net Classes (Types)

⎕NL can be used to explore .Net types.

When a reference is made to an undefined name, and ⎕USING is set, APL attempts to load the Type from the appropriate .Net Assemblies. If successful, the name is entered into the symbol table with name-class 9.6.

```
      ⎕USING←'System'
      DateTime
(System.DateTime)
      ⎕NL -9
 DateTime
      ⎕NC,⊂'DateTime'
9.6
```

The names of the Properties and Methods of a .Net Type may then be obtained using ⎕NL.

```
      DateTime.⎕NL -2 ⍝ Properties
 MaxValue  MinValue  Now  Today  UtcNow

      DateTime.⎕NL -3 ⍝ Methods
 get_Now  get_Today  get_UtcNow  op_Addition  op_Equality
 ...
```

In fact it is not necessary to make a separate reference first, because the expression Type.⎕NL (where Type is a .Net Type) is itself a reference to Type. So, (with ⎕USING still set to 'System'):

```
      Array.⎕NL -3
 BinarySearch  Clear  Copy  CreateInstance  IndexOf  LastI
ndexOf  Reverse  Sort

      ⎕NL -9
 Array  DateTime
```

Another use for ⎕NL is to examine .Net *enumerations*. For example:

```
      ⎕USING←'System.Windows.Forms,system.windows.forms.dl
l'

      FormBorderStyle.⎕NL -2
Fixed3D  FixedDialog  FixedSingle  FixedToolWindow  None
Sizable  SizableToolWindow

      FormBorderStyle.FixedDialog.value__
3

      FormBorderStyle.({⍵,[1.5]⍎¨⍵,¨⊂'.value__'}⎕NL -2)
 Fixed3D            2
 FixedDialog        3
 FixedSingle        1
 FixedToolWindow    5
 None               0
 Sizable            4
 SizableToolWindow  6
```

COM Objects

Once a reference to a COM object has been obtained, ⎕NL may be used to obtain lists of its Methods, Properties and Events.

```
      xl←⎕NEW'OLEClient'(⊂'ClassName' 'Excel.Application')

      xl.⎕NL -2 ⍝ Properties
 _Default  ActiveCell  ActiveChart  ActiveDialog  ActiveMe
nuBar  ActivePrinter  ActiveSheet  ActiveWindow ...

      xl.⎕NL -3 ⍝ Methods
 _Evaluate  _FindFile  _Run2  _Wait  _WSFunction  Activate
MicrosoftApp  AddChartAutoFormat  AddCustomList  Browse  C
alculate ...

      ⎕NL -9
 xl
```

Source: R←⎕SRC Y

`⎕SRC` returns the script that defines the Class `Y`.

`Y` must be a reference to a Class.
`R` is a vector of character vectors containing the script that was used to define Class `Y`.

```
      )ED ○MyClass

:Class MyClass
     ∇ Make Name
       :Implements Constructor
       ⎕DF Name
     ∇
:EndClass ⍝ MyClass

      Z←⎕SRC MyClass
      ⍴Z
6
      ⍴¨Z
 14  15  29  14  5  19
      6 1⍴Z
 :Class MyClass
     ∇ Make Name
       :Implements Constructor
       ⎕DF Name
     ∇
 :EndClass ⍝ MyClass
```

This Space: R←⎕THIS

⎕THIS returns a reference to the current namespace, i.e. to the space in which it is referenced.

If NC9 is a reference to any object whose name-class is 9, then:

```
      NC9≡NC9.⎕THIS
1
```

Examples

```
      ⎕THIS
#
      'X'⎕NS ''
      X.⎕THIS
#.X
     'F'⎕WC'Form'
     'F.B'⎕WC'Button'
      F.B.⎕THIS
#.F.B

      Polly←⎕NEW Parrot
      Polly.⎕THIS
#.[Parrot]
```

An Instance may use ⎕THIS to obtain a reference to its own Class:

```
    Polly.(⊃⊃⎕CLASS ⎕THIS)
#.Parrot
```

or a function (such as a Constructor or Destructor) may identify or enumerate all other Instances of the same Class:

```
      Polly.(⍴⎕INSTANCES⊃⊃⎕CLASS ⎕THIS)
1
```

Window Expose: ⎕WX

⎕WX is a system variable that determines:

a) whether or not the names of properties, methods and events provided by a Dyalog APL GUI object are exposed.
b) certain aspects of behaviour of .Net and COM objects. See External Object behaviour.

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

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

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

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

Note that it is the value of ⎕WX in the object itself, rather than the value of ⎕WX in the calling environment, that determines its behaviour.

The value of ⎕WX in a clear workspace is defined by the default_wx parameter (see User Guide) which itself defaults to 3.

⎕WX has namespace scope and may be localised in a function header. This allows you to create a utility namespace or utility function in which the exposure of objects is known and determined, regardless of its global value in the workspace.

List Classes:)CLASSES

This command lists the names of APL Classes in the active workspace.

Example:

```
      )CLEAR
clear ws
      )ED ○MyClass

:Class MyClass
    ∇ Make Name
      :Implements Constructor
      ⎕DF Name
    ∇
:EndClass ⍝ MyClass

      )CLASSES
MyClass
      )COPY OO YourClass
.\OO saved Sun Jan 29 18:32:03 2006
      )CLASSES
MyClass YourClass
      ⎕NC 'MyClass' 'YourClass'
9.4 9.4
```

Edit Object:)ED nms

`)ED` invokes the Dyalog APL editor and opens an Edit window for each of the objects specified in `nms`.

If a name specifies a new symbol it is taken to be a function/operator. However, if a name is localised in a suspended function/operator but is otherwise undefined, it is assumed to be a vector of character vectors.

The type of a new object may be specified explicitly by preceding its name with an appropriate symbol as follows :

∇	function/operator
→	simple character vector
∊	vector of character vectors
-	character matrix
⍟	Namespace script
○	Class script
∘	Interface

The first object named becomes the top window on the stack. See *User Guide* for details. `)ED` ignores names which specify GUI objects.

Examples

```
      )ED MYFUNCTION

      )ED ∇FOO -MAT ∊VECVEC
```

Function Declaration Statements

Certain statements that are used to identify the characteristics of a function in some way. These statements are not executable statements and may appear anywhere in the body of the function.

Access Statement :Access

```
:Access <Private|Public><Instance|Shared>
:Access <WebMethod>
```

The :Access statement is used to specify characteristics for functions that represent Methods in classes (see chapter 3). It is also applicable to Classes and Properties.

Element	Description
`Private|Public`	Specifies whether or not the method is accessible from outside the Class or an Instance of the Class. The default is `Private`.
`Instance|Shared`	Specifies whether the method runs in the Class or Instance. The default is `Instance`.
`WebMethod`	Specifies that the method is exported as a web method. This applies only to a Class that implements a Web Service.
`Overridable`	Applies only to an Instance Method and specifies that the Method may be overridden by a Method in a higher Class. See below.
`Override`	Applies only to an Instance Method and specifies that the Method overrides the corresponding Overridable Method defined in the Base Class. See below

Overridable/Override

Normally, a Method defined in a higher Class replaces a Method of the same name that is defined in its Base Class, but only for calls made from above or within the higher Class itself (or an Instance of the higher Class). The base method remains available *in the Base Class* and is invoked by a reference to it *from within the Base Class*.

However, a Method declared as being `Overridable` is replaced in situ (i.e. within its own Class) by a Method of the same name in a higher Class if that Method is itself declared with the `Override` keyword. For further information, see Superseding Base Class Methods.

WebMethod

Note that `:Access WebMethod` is equivalent to:

```
:Access Public
:Attribute System.Web.Services.WebMethodAttribute
```

Attribute Statement :Attribute

```
:Attribute <Name> [ConstructorArgs]
```

The :Attribute statement is used to attach .Net Attributes to a Method (or Class).

Attributes are descriptive tags that provide additional information about programming elements. Attributes are not used by Dyalog APL but other applications can refer to the extra information in attributes to determine how these items can be used. Attributes are saved with the *metadata* of Dyalog APL .NET assemblies.

Element	Description
`Name`	The name of a .Net attribute
`ConstructorArgs`	Optional arguments for the Attribute constructor

Examples

```
      :Attribute ObsoleteAttribute
      :Attribute ObsoleteAttribute 'Don''t use' 1
```

Implements Statement :Implements

```
:Implements Constructor <[:Base expr]>
:Implements Destructor
:Implements Method <InterfaceName.MethodName>
:Implements Trigger <name1><,name2,name3,...
```

The :Implements statement identifies the function to be one of the following special types.

Element	Description
`Constructor`	Specifies that the function is a class constructor.
`:Base expr`	Specifies that the Base Constructor be called with the result of the expression `expr` as its argument.
`Destructor`	Specifies that the method is a Class Destructor.
`Method`	Specifies that the function implements the Method `MethodName` whose syntax is specified by Interface `InterfaceName`.
`Trigger`	Identifies the function as a Trigger Function which is activated by changes to variables `name1`, `name2`, etc. (see Triggers).

Signature Statement :Signature

```
:Signature <rslttype←><name><arg1type arg1name>,...
```

This statement identifies the name and signature by which a function is exported as a method to be called from outside Dyalog APL. Several :Signature statements may be specified to allow the method to be called with different arguments and/or to specify a different result type.

Element	Description
`rslttype`	Specifies the data type for the result of the method
`name`	Specifies the name of the exported method.
`argntype`	Specifies the data type of the nth parameter
`argnname`	Specifies the name of the nth parameter

Argument and result data types are identified by the names of .Net Types which are defined in the .Net Assemblies specified by ⎕USING or by a :USING statement.

Examples

In the following examples, it is assumed that the .Net Search Path (defined by :Using or ⎕USING includes 'System'.

The following statement specifies that the function is exported as a method named Format which takes a single parameter of type System.Object named Array. The data type of the result of the method is an array (vector) of type System.String.

```
:Signature String[]←Format Object Array
```

The next statement specifes that the function is exported as a method named Catenate whose result is of type System.Object and which takes 3 parameters. The first parameter is of type System.Double and is named Dimension. The second is of type System.Object and is named Arg1. The third is of type System.Object and is named Arg2.

```
:Signature Object←Catenate Double Dimension,...
                          ...Object Arg1, Object Arg2
```

The next statement specifes that the function is exported as a method named IndexGen whose result is an array of type System.Int32 and which takes 2 parameters. The first parameter is of type System.Int32 and is named N. The second is of type System.Int32 and is named Origin.

```
:Signature Int32[]←IndexGen Int32 N, Int32 Origin
```

The next block of ststements specifies that the function is exported as a method named Mix. The method has 4 different signatures; i.e. it may be called with 4 different parameter/result combinations.

```
:Signature Int32[,]←Mix Double Dimension, ...
      ...Int32[] Vec1, Int32[] Vec2
:Signature Int32[,]←Mix Double Dimension,...
      ... Int32[] Vec1, Int32[] Vec2, Int32 Vec3
:Signature Double[,]←Mix Double Dimension, ...
      ... Double[] Vec1, Double[] Vec2
:Signature Double[,]←Mix Double Dimension, ...
      ... Double[] Vec1, Double[] Vec2, Double[] Vec
3
```

Triggers

Triggers provide the ability to have a function called automatically whenever a variable or a Field is assigned. Triggers are actioned by all forms of assignment (←), but only by assignment.

Triggers are designed to allow a class to perform some action when a field is modified – without having to turn the field into a property and use the property setter function to achieve this. Avoiding the use of a property allows the full use of the APL language to manipulate data in a field, without having to copy field data in and out of the class through get and set functions.

Triggers *can* also be applied to variables outside a class, and there will be situations where this is very useful. However, dynamically attaching and detaching a trigger from a variable is a little tricky at present.

The function that is called when a variable or Field changes is referred to as the *Trigger Function*. The name of a variable or Field which has an associated Trigger Function is termed a *Trigger*.

A function is declared as aTrigger function by including the statement:

```
      :Implements Trigger Name1,Name2,Name3, ...
```

where `Name1`, `Name2` etc are the Triggers.

When a Trigger function is invoked, it is passed an Instance of the internal Class `TriggerArguments`. This Class has 3 Fields:

Member	**Description**
`Name`	Name of the Trigger whose change in value has caused the Trigger Function to be invoked.
`NewValue`	The newly assigned value of the Trigger
`OldValue`	The previous value of the Trigger. If the Trigger was not previously defined, a reference to this Field causes a `VALUE ERROR`.

A Trigger Function is called *as soon as possible* after the value of a Trigger was assigned; typically by the end of the currently executing line of APL code. The precise timing is not guaranteed and may not be consistent because internal workspace management operations can occur at any time.

If the value of a Trigger is changed more than once by a line of code, the Trigger Function will be called at least once, but the number of times is not guaranteed.

A Trigger Function is not called when the Trigger is expunged.

Expunging a Trigger disconnects the name from the Trigger Function and the Trigger Function will not be invoked when the Trigger is reassigned. The connection may be re-established by re-fixing the Trigger Function.

A Trigger may have only a single Trigger Function. If the Trigger is named in more than one Trigger Function, the Trigger Function that was last fixed will apply.

In general, it is inadvisable for a Trigger function to modify its own Trigger, as this will potentially cause the Trigger to be invoked repeatedly and forever.

To associate a Trigger function with a *local* name, it is necessary to dynamically fix the Trigger function in the function in which the Trigger is localised; for example:

```
     ∇ TRIG arg
       :Implements Trigger A
       ...

     ∇ TEST;A
       ⎕FX ⎕OR'TRIG'
       A←10
       ...
```

Example

The following function displays information when the value of variables A or B changes.

```
     ∇ TRIG arg
       :Implements Trigger A,B
       arg.Name'is now 'arg.NewValue
       :Trap 6 ⍝ VALUE ERROR
           arg.Name'was    'arg.OldValue
       :Else
           arg.Name' was     [undefined]'
       :EndTrap
     ∇
```

Note that on the very first assignment to `A`, when the variable was previously undefined, `arg.OldValue` is a `VALUE ERROR`.

```
      A←10
A  is now   10
A   was     [undefined]

      A+←10
A  is now   20
A  was      10

      A←'Hello World'
A  is now   Hello World
A  was      20

      A[1]←⊂2 3⍴⍳6
A  is now    1 2 3 ello World
             4 5 6
A  was      Hello World

      B←⌽¨A
B  is now    3 2 1 ello World
             6 5 4
B   was     [undefined]

      A←⎕NEW MyClass
A  is now   #.[Instance of MyClass]
A  was       1 2 3 ello World
             4 5 6

      'F'⎕WC'Form'
      A←F
A  is now   #.F
A  was      #.[Instance of MyClass]
```

Note that Trigger functions are actioned only by assignment, so changing `A` to a Form using `⎕WC` does not invoke `TRIG`.

```
      'A'⎕WC'FORM' ⍝ Note that Trigger Function is not inv
oked
```

However, the connection (between `A` and `TRIG`) remains and the Trigger Function will be invoked if and when the Trigger is re-assigned.

```
      A←99
A  is now   99
A  was      #.A
```

Symbolic Index

Alphabetic Index