Programmer's Guide

Language Reference - Volume 1

Version 13.1

DYALOG APL

The tool of thought for expert programming

Contents

Chapter 1:

Introduction

Workspaces

APL expressions are evaluated within a workspace. The workspace may contain objects, namely operators, functions and variables defined by the user. APL expressions may include references to operators, functions and variables provided by APL. These objects do not reside in the workspace, but space is required for the actual process of evaluation to accommodate temporary data. During execution, APL records the state of execution through the STATE INDICATOR which is dynamically maintained until the process is complete. Space is also required to identify objects in the workspace in the SYMBOL TABLE. Maintenance of the symbol table is entirely dynamic. It grows and contracts according to the current workspace contents.

Workspaces may be explicitly saved with an identifying name. The workspace may subsequently be loaded, or objects may be selectively copied from a saved workspace into the current workspace.

Under UNIX, workspace names must be valid file names, but are otherwise unrestricted. See your UNIX documentation for details.

Under Windows, Dyalog APL workspaces are stored in files with the suffix ".DWS". However, they are referred to from within APL by only the first part of the file name which must conform to Windows file naming rules.

Namespaces

Namespace is a (class 9) object in Dyalog APL. Namespaces are analogous to nested workspaces.

```
'Flat' APL Workspace       Workspace with Namespaces
.OLD-------------------.    .NEW--------------------.
|                      |    | FOO MAT VEC           |
| DISPLAY              |    | .Util----------.      |
|                      |    | |DISPLAY       |      |
|   FOO MAT VEC        |    | |...           |      |
|                      |    | '--------------'      |
|   WsDoc_Init         |    | .WsDoc-------------.  |
|   WsDoc_Xref         |    | |Init .prt-..fmt--.|  |
|   WsDoc_Tree         |    | |     |Init||line ||  |
|   WsDoc_prt_init     |    | |Tree |    ||     ||  |
|   WsDoc_current_page |    | |Xref |page||     ||  |
|   ...                |    | |     '----''-----'|  |
|                      |    | '------------------'  |
'----------------------'    '-----------------------'
```

They provide the same sort of facility for workspaces as directories do for filesystems. The analogy might prove helpful:

Operation	MS-DOS	Namespace
Create	MKDIR"apl""Dyalog"	`)NS or ⎕NS`
Change	CD	`)CS or ⎕CS`
Relative name	DIR1\DIR2\FILE	`NS1.NS2.OBJ`
Absolute name	\DIR\FILE	`#.NS.OBJ`
Name separator	\	`.`
Top (root) object	\	`#`
Parent object	..	`##`

Namespaces bring a number of major benefits:

They provide static (as opposed to dynamic) local names. This means that a defined function can use local variables and functions which persist when it exits and which are available next time it is called.

Just as with the provision of directories in a filing system, namespaces allow us to organise the workspace in a tidy fashion. This helps to promote an object oriented programming style.

APL's traditional name-clash problem is ameliorated in several ways:

- Workspaces can be arranged so that there are many fewer names at each namespace level. This means that when copying objects from saved workspaces there is a much reduced chance of a clash with existing names.
- Utility functions in a saved workspace may be coded as a single namespace and therefore on being copied into the active workspace consume only a single name. This avoids the complexity and expense of a solution which is sometimes used in 'flat' workspaces, where such utilities dynamically fix local functions on each call.
- In flat APL, workspace administration functions such as `WSDOC` must share names with their subject namespace. This leads to techniques for trying to avoid name clashes such as using obscure name prefixes like `'⍙⍙L1'` This problem is now virtually eliminated because such a utility can operate exclusively in its own namespace.

The programming of GUI objects is considerably simplified.

- An object’s callback functions may be localised in the namespace of the object itself.
- Static variables used by callback functions to maintain information between calls may be localised within the object.

This means that the object need use only a single name in its namespace.

Arrays

A Dyalog APL data structure is called an array. An array is a rectangular arrangement of items, each of which may be a single number, a single character, a namespace reference (ref), another array, or the ⎕OR of an object. An array which is part of another array is also known as a subarray.

An array has two properties; structure and data type. Structure is identified by rank, shape, and depth.

Rank

An array may have 0 or more axes or dimensions. The number of axes of an array is known as its rank. Dyalog APL supports arrays with a maximum of 15 axes.

- An array with 0 axes (rank 0) is called a scalar.
- An array with 1 axis (rank 1) is called a vector.
- An array with 2 axes (rank 2) is called a matrix.
- An array with more than 2 axes is called a multi-dimensional array.

Shape

Each axis of an array may contain zero or more items. The number of items along each axis of an array is called its shape. The shape of an array is itself a vector. Its first item is the length of the first axis, its second item the length of the second axis, and so on. An array, whose length along one or more axes is zero, is called an empty array.

Depth

An array whose items are all simple scalars (i.e. single numbers, characters or refs) is called a simple array. If one or more items of an array is not a simple scalar (i.e. is another array, or a ⎕OR), the array is called a nested array. A nested array may contain items which are themselves nested arrays. The degree of nesting of an array is called its depth. A simple scalar has a depth of 0. A simple vector, matrix, or multi-dimensional array has depth 1. An array whose items are all depth 1 subarrays has depth 2; one whose items are all depth 2 subarrays has depth 3, and so forth.

Type

An array, whose elements are all numeric, is called a numeric array; its TYPE is numeric. A character array is one in which all items are characters. An array whose items contain both numeric and character elements is of MIXED type.

Numbers

Dyalog APL supports both real numbers and complex numbers.

Real Numbers

Numbers are entered or displayed using conventional decimal notation (e.g. 299792.458) or using a scaled form (e.g. 2.999792458E5).

On entry, a decimal point is optional if there is no fractional part. On output, a number with no fractional part (an integer) is displayed without a decimal point.

The scaled form consists of:

a. an integer or decimal number called the mantissa,
b. the letter `E` or `e`,
c. an integer called the scale, or exponent.

The scale specifies the power of 10 by which the mantissa is to be multiplied.

Example

```
      12 23.24 23.0 2.145E2
12 23.24 23 214.5
```

Negative numbers are preceded by the high minus (`¯`) symbol, not to be confused with the minus (`-`) function. In scaled form, both the mantissa and the scale may be negative.

Example

```
      ¯22 2.145E¯2 ¯10.25
¯22 0.02145 ¯10.25
```

Complex Numbers

Complex numbers use the J notation introduced in IBM APL2 and are written as `aJb` or `ajb` (without spaces) where the real and imaginary parts `a` and `b` are written as described above. The capital `J` is always used to display a value.

Examples

```
      2+¯1*.5
2J1
      .3j.5
0.3J0.5
       1.2E5J¯4E¯4
120000J¯0.0004
```

The empty vector (`⍳0`) may be represented by the numeric constant `⍬` called ZILDE.

Characters

Characters are entered within a pair of APL quotes. The surrounding APL quotes are not displayed on output. The APL quote character itself must be entered as a pair of APL quotes.

Examples

```
      'DYALOG APL'
DYALOG APL

      'I DON''T KNOW'
I DON'T KNOW

      '*'
*
```

Enclosed Elements

An array may be enclosed to form a scalar element through any of the following means:

- by the enclose function (⊂)
- by inclusion in vector notation
- as the result of certain functions when applied to arrays

Examples

```
      (⊂1 2 3),⊂'ABC'
1 2 3  ABC

       (1 2 3) 'ABC'
1 2 3  ABC

      ⍳2 3
1 1  1 2  1 3
2 1  2 2  2 3
```

Legal Names

APL objects may be given names. A name may be any sequence of characters, starting with an alphabetic character, selected from the following:

0123456789(but not as the 1st character in a name)

ABCDEFGHIJKLMNOPQRSTUVWXYZ_

abcdefghijklmnopqrstuvwxyz

ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝß

àáâãäåæçèéêëìíîïðñòóôõöøùúûüþ

∆⍙

A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲Q̲R̲S̲T̲U̲V̲W̲X̲Y̲Z̲

Note that using a standard Unicode font (rather than APL385 Unicode used in the table above), the last row above would appear as the circled alphabet, Ⓐ to ⓩ.

Examples

Legal	Illegal
THIS∆IS∆A⍙NAME	BAD NAME
X1233	3+21
SALES	S!H|PRICE
pjb_1	1_pjb

Specification of Variables

A variable is a named array. An undefined name or an existing variable may be assigned an array by specification with the left arrow (←).

Examples

```
      A←'CHIPS WITH EVERYTHING'
      A
CHIPS WITH EVERYTHING

      X Y←'ONE' 'TWO'
      X
ONE
      Y
TWO
```

Vector Notation

A series of two or more adjacent expressions results in a vector whose elements are the enclosed arrays resulting from each expression. This is known as VECTOR (or STRAND) NOTATION. Each expression in the series may consist of one of the following:

a. a single numeric value;
b. single character, within a pair of quotes;
c. more than one character, within a pair of quotes;
d. the name of a variable;
e. the evaluated input symbol ⎕;
f. the quote-quad symbol ⍞;
g. the name of a niladic, defined function yielding a result;
h. any other APL expression which yields a result, within parentheses.

Examples

```
      ⍴A←2 4 10
3
      ⍴TEXT←'ONE' 'TWO'
2
```

Numbers and characters may be mixed:

```
      ⍴X←'THE ANSWER IS ' 10
2
      X[1]
 THE ANSWER IS
      X[2] + 32
42
```

Blanks, quotes or parentheses must separate adjacent items in vector notation. Redundant blanks and parentheses are permitted. In this manual, the symbol pair '←→' indicates the phrase 'is equivalent to'.

```
      1  2  ←→ (1)(2) ←→ 1  (2)  ←→ (1)  2
      2'X'3 ←→ 2 'X' 3 ←→ (2) ('X') (3)
      1  (2+2) ←→ (1) ((2+2)) ←→ ((1))  (2+2)
```

Vector notation may be used to define an item in vector notation:

```
       ⍴X ← 1 (2 3 4) ('THIS' 'AND' 'THAT')
3
      X[2]
 2 3 4
      X[3]
  THIS  AND  THAT
```

Expressions within parentheses are evaluated to produce an item in the vector:

```
      Y ← (2+2) 'IS' 4
      Y
4  IS  4
```

The following identity holds:

```
      A  B  C  ←→ (⊂A), (⊂B), ⊂C
```

Structuring of Arrays

A class of primitive functions re-structures arrays in some way. Arrays may be input only in scalar or vector form. Structural functions may produce arrays with a higher rank. The Structural functions are reshape (⍴), ravel, laminate and catenate (,), reversal and rotation (⌽), transpose (⍉), mix and take (↑), split and drop (↓), and enclose (⊂). These functions are described in *Chapter 4*.

Examples

```
      2 2⍴1 2 3 4
1 2
3 4

      2 2 4⍴'ABCDEFGHIJKLMNOP'
ABCD
EFGH

IJKL
MNOP
         ↓2 4⍴'COWSHENS'
 COWS  HENS
```

Display of Arrays

Simple scalars and vectors are displayed in a single line beginning at the left margin. A number is separated from the next adjacent element by a single space. The number of significant digits to be printed is determined by the system variable `⎕PP` whose default value is 10. The fractional part of the number will be rounded in the last digit if it cannot be represented within the print precision. Trailing zeros after a decimal point and leading zeros will not be printed. An integer number will display without a decimal point.

Examples

```
      0.1 1.0 1.12
0.1 1 1.12

      'A' 2 'B' 'C'
A 2 BC

      ÷3 2 6
0.3333333333 0.5 0.1666666667
```

If a number cannot be fully represented in `⎕PP` significant digits, or if the number requires more than five leading zeros after the decimal point, the number is represented in scaled form. The mantissa will display up to `⎕PP` significant digits, but trailing zeros will not be displayed.

Examples

```
      ⎕PP←3

      123 1234 12345 0.12345 0.00012345 0.00000012345
123 1.23E3 1.23E4 0.123 0.000123 1.23E¯7
```

Simple matrices are displayed in rectangular form, with one line per matrix row. All elements in a given column are displayed in the same format, but the format and width for each column is determined independently of other columns. A column is treated as numeric if it contains any numeric elements. The width of a numeric column is determined such that the decimal points (if any) are aligned; that the `E` characters for scaled formats are aligned, with trailing zeros added to the mantissae if necessary, and that integer forms are right-adjusted one place to the left of the decimal point column (if any). Numeric columns are right-justified; a column which contains no numeric elements is left-justified. Numeric columns are separated from their neighbours by a single column of blanks.

Examples

```
      2 4⍴'HANDFIST'
HAND
FIST

      1 2 3 ∘.× 6 2 5
 6 2  5
12 4 10
18 6 15

      2 3⍴2 4 6.1 8 10.24 12
2  4     6.1
8 10.24 12

      2 4⍴4 'A' 'B' 5 ¯0.000000003 'C' 'D' 123.56
 4E0  AB   5
¯3E¯9 CD 123.56
```

In the display of non-simple arrays, each element is displayed within a rectangle such that the rows and columns of the array are aligned. Simple items within the array are displayed as above. For non-simple items, this rule is applied recursively, with one space added on each side of the enclosed element for each level of nesting.

Examples

```
      ⍳3
1 2 3

      ⊂⍳3
 1 2 3

      ⊂⊂⍳3
  1 2 3

      ('ONE' 1) ('TWO' 2) ('THREE' 3) ('FOUR' 4)
  ONE  1   TWO  2   THREE  3   FOUR  4

      2 4⍴'ONE' 1 'TWO' 2 'THREE' 3 'FOUR' 4
 ONE    1  TWO   2
 THREE  3  FOUR  4
```

Multi-dimensional arrays are displayed in rectangular planes. Planes are separated by one blank line, and hyper-planes of higher dimensions are separated by increasing numbers of blank lines. In all other respects, multi-dimensional arrays are displayed in the same manner as matrices.

Examples

```
      2 3 4⍴⍳24
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

      3 1 1 3⍴'THEREDFOX'
THE


RED


FOX
```

The power of this form of display is made apparent when formatting informal reports.

Examples

```
      +AREAS←'West' 'Central' 'East'
 West  Central  East

      +PRODUCTS←'Biscuits' 'Cakes' 'Buns' 'Rolls'
 Biscuits  Cakes  Buns  Rolls

      SALES←50 5.25 75 250 20.15 900 500
      SALES,←80.98 650 1000 90.03 1200
      +SALES←4 3⍴SALES
  50  5.25   75
 250 20.15  900
 500 80.98  650
1000 90.03 1200

      ' ' PRODUCTS ⍪., AREAS SALES
            West  Central  East
 Biscuits     50     5.25    75
 Cakes       250    20.15   900
 Buns        500    80.98   650
 Rolls      1000    90.03  1200
```

If the display of an array is wider than the page width, as set by the system variable ⎕PW, it will be folded at or before ⎕PW and the folded portions indented six spaces. The display of a simple numeric or mixed array may be folded at a width less than ⎕PW so that individual numbers are not split across a page boundary.

Example

```
      ⎕PW←40

      ?3 20⍴100
54 22  5 68 68 94 39 52 84  4  6 53 68
85 53 10 66 42 71 92 77 27  5 74 33 64
66  8 64 89 28 44 77 48 24 28 36 17 49
       1  39  7 42 69 49 94
      76 100 37 25 99 73 76
      90  91  7 91 51 52 32
```

The Display Function

The DISPLAY function is implemented as a user command]display distributed with Dyalog APL and may be used to illustrate the structure of an array.]display is monadic. Its result is a character matrix containing a pictorial representation of its argument.]display is used throughout this manual to illustrate examples. An array is illustrated with a series of boxes bordering each sub-array. Characters embedded in the border indicate rank and type information. The top and left borders contain symbols that indicate its rank. A symbol in the lower border indicates type. The symbols are defined as follows:

→	Vector.
↓	Matrix or higher rank array.
⊖	Empty along last axis.
⌽	Empty along other than last axis.
∊	Nested array.
~	Numeric data.
-	Character data.
+	Mixed character and numeric data.
∇	⎕OR object.
#	array of refs.

```
      ]display 'ABC' (1 4⍴1 2 3 4)
.→-----------------.
| .→--.  .→------. |
| |ABC|  ↓1 2 3 4| |
| '---'  '~-------' |
'∊-----------------'
```

Prototypes and Fill Items

Every array has an associated *prototype* which is derived from the array's first item.

If the first item is a number, the prototype is 0. Otherwise, if the first item is a character, the prototype is `' '` (space). Otherwise, if the first item is a (ref to) an instance of a Class, the prototype is a ref to that Class.

Otherwise (in the nested case, when the first item is other than a simple scalar), the prototype is defined recursively as the prototype of each of the array's first item.

Examples:

Array	Prototype
`1 2 3.4`	`0`
`2 3 5⍴'hello'`	`' '`
`99 'b' 66`	`0`
`(1 2)(3 4 5)`	`0 0`
`((1 2)3)(4 5 6)`	`(0 0)0`
`'hello' 'world'`	`'     '`
`⎕NEW MyClass`	`MyClass`
`(88(⎕NEW MyClass)'X')7`	`0 MyClass ' '`

Fill Items

Fill items for an overtake operation, are derived from the argument's prototype. For each `0` or `' '` in the prototype, there is a corresponding `0` or `' '` in the fill item and for each class reference in the prototype, there is a ref to a (newly constructed and distinct) instance of that class that is initialised by the niladic (default) constructor for that class, if defined.

Examples:

```
      4↑1 2
1 2 0 0
      4↑'ab'
ab
      4↑(1 2)(3 4 5)
 1 2  3 4 5  0 0  0 0
      2↑⎕NEW MyClass
 #.[Instance of MyClass]  #.[Instance of MyClass]
```

In the last example, two distinct instances are constructed (the first by `⎕NEW` and the second by the overtake).

Fill items are used in a number of operations including:

- First (`⊃` or `↑`) of an empty array
- Fill-elements for overtake
- For use with the Each operator on an empty array

Expressions

An expression is a sequence of one or more syntactic tokens which may be symbols or constants or names representing arrays (variables) or functions. An expression which produces an array is called an ARRAY EXPRESSION. An expression which produces a function is called a FUNCTION EXPRESSION. Some expressions do not produce a result.

An expression may be enclosed within parentheses.

Evaluation of an expression proceeds from right to left, unless modified by parentheses. If an entire expression results in an array that is not assigned to a name, then that array value is displayed. (Some system functions and defined functions return an array result only if the result is assigned to a name or if the result is the argument of a function or operator.)

Examples

```
      X←2×3-1

      2×3-1
4
      (2×3)-1
5
```

Either blanks or parentheses are required to separate constants, the names of variables, and the names of defined functions which are adjacent. Excessive blanks or sets of parentheses are redundant, but permitted. If `F` is a function, then:

```
      F 2↔ F(2) ↔ (F)2 ↔ (F) (2) ↔ F  (2) ↔ F ((2))
```

Blanks or parentheses are not needed to separate primitive functions from names or constants, but they are permitted:

```
      -2 ↔ (-)(2) ↔ (-) 2
```

Blanks or parentheses are not needed to separate operators from primitive functions, names or constants. They are permitted with the single exception that a dyadic operator must have its right argument available when encountered. The following syntactical forms are accepted:

```
      (+.×) ↔ (+).× ↔ +.(×)
```

The use of parentheses in the following examples is not accepted:

```
      +(.)×  or    (+.)×
```

Functions

A function is an operation which is performed on zero, one or two array arguments and may produce an array result. Three forms are permitted:

- NILADIC defined for no arguments
- MONADIC defined for a right but not a left argument
- DYADIC defined for a left and a right argument

The number of arguments is referred to as its VALENCE.

The name of a non-niladic function is AMBIVALENT; that is, it potentially represents both a monadic and a dyadic function, though it might not be defined for both. The usage in an expression is determined by syntactical context. If the usage is not defined an error results.

Functions have long SCOPE on the right; that is, the right argument of the function is the result of the entire expression to its right which must be an array. A dyadic function has short scope on the left; that is, the left argument of the function is the array immediately to its left. Left scope may be extended by enclosing an expression in parentheses whence the result must be an array.

For some functions, the explicit result is suppressed if it would otherwise be displayed on completion of evaluation of the expression. This applies on assignment to a variable name. It applies for certain system functions, and may also apply for defined functions.

Examples

```
      10×5-2×4
¯30
      2×4
8
      5-8
¯3
      10×¯3
¯30
      (10×5)-2×4
42
```

Defined Functions

Functions may be defined with the system function ⎕FX, or with the function editor. A function consists of a HEADER which identifies the syntax of the function, and a BODY in which one or more APL statements are specified.

The header syntax identifies the function name, its (optional) result and its (optional) arguments. If a function is ambivalent, it is defined with two arguments but with the left argument within braces ({}). If an ambivalent function is called monadically, the left argument has no value inside the function. If the explicit result is to be suppressed for display purposes, the result is shown within braces. A function need not produce an explicit result. Refer to *Chapter 2* for further details.

Example

```
     ∇ R←{A} FOO B
[1]    R←⊃'MONADIC' 'DYADIC'[⎕IO+0≠⎕NC'A']
[2]  ∇

      FOO 1
MONADIC

      'X' FOO 'Y'
DYADIC
```

Functions may also be created by using assignment (←).

Function Assignment & Display

The result of a function-expression may be given a name. This is known as FUNCTION ASSIGNMENT (see also "Dynamic Functions & Operators" on page 112). If the result of a function-expression is not given a name, its value is displayed. This is termed FUNCTION DISPLAY.

Examples

```
      PLUS←+
      PLUS
+
      SUM←+/
      SUM
+/
```

Function expressions may include defined functions and operators. These are displayed as a ∇ followed by their name.

Example

```
      ∇ R←MEAN X     ⍝ Arithmetic mean
[1]     R←(+/X)÷⍴X
      ∇


      MEAN
 ∇MEAN

      AVERAGE←MEAN
      AVERAGE
 ∇MEAN

      AVG←MEAN∘,
      AVG
 ∇MEAN ∘,
```

Operators

An operator is an operation on one or two operands which produces a function called a DERIVED FUNCTION. An operand may be a function or an array. Operators are not ambivalent. They require either one or two operands as applicable to the particular operator. However, the derived function may be ambivalent. The derived function need not return a result. Operators have higher precedence than functions. Operators have long scope on the left. That is, the left operand is the longest function or array expression on its left. The left operand may be terminated by:

1. the end of the expression
2. the right-most of two consecutive functions
3. a function with an array to its left
4. an array with a function to its left

an array or function to the right of a monadic operator.

A dyadic operator has short scope on the right. That is, the right operand of an operator is the single function or array on its right. Right scope may be extended by enclosing an expression in parentheses.

Examples

```
      ρ¨X←'WILLIAM' 'MARY' 'BELLE'
 7  4  5

      ρ∘ρ¨X
 1  1  1

      (ρ∘ρ)¨X
 1  1  1

      ⎕∘←∘⎕VR¨'PLUS' 'MINUS'
     ∇ R←A PLUS B
[1]    R←A+B
     ∇
     ∇ R←A MINUS B
[1]    R←A-B
     ∇

      PLUS/1 2 3 4
10
```

Defined Operators

Operators may be defined with the system function ⎕FX, or with the function editor. A defined operator consists of a HEADER which identifies the syntax of the operator, and a BODY in which one or more APL statements are specified.

A defined operator may have one or two operands; and its derived function may have one or two arguments, and may or may not produce a result. The header syntax defines the operator name, its operand(s), the argument(s) to its derived function, and the result (if any) of its derived function. The names of the operator and its operand(s) are separated from the name(s) of the argument(s) to its derived function by parentheses.

Example

```
      ∇ R←A(F AND G)B
[1]     R←(A F B)(A G B)
      ∇
```

The above example shows a dyadic operator called AND with two operands (F and G). The operator produces a derived function which takes two arguments (A and B), and produces a result (R).

```
      12 +AND÷ 4
16 3
```

Operands passed to an operator may be either functions or arrays.

```
      12 (3 AND 5) 4
12 3 4  12 5 4

      12 (× AND 5) 4
48  12 5 4
```

Complex Numbers

A complex number is a number consisting of a real and an imaginary part which is usually written in the form ***a+ bi***, where ***a*** and ***b*** are real numbers, and ***i*** is the standard imaginary unit with the property $i^2 = -1$.

Dyalog APL adopts the J notation introduced in IBM APL2 to represent the value of a complex number which is written as `aJb` or `ajb` (without spaces). The former representation (with a capital `J`) is always used to display a value.

Notation

```
      2+¯1*.5
2J1

      .3j.5
0.3J0.5

      1.2E5J¯4E¯4
120000J¯0.0004
```

Arithmetic

The arithmetic primitive functions handle complex numbers in the appropriate way.

```
      2j3+.3j.5 ⍝ (a+bi)+(c+di) = (a+c)+(b+d)i
2.3J3.5

      2j3-.3j5  ⍝ (a+bi)-(c+di) = (a-c)+(b-d)i
1.7J¯2

      2j3×.3j.5 ⍝ (a+bi)(c+di)= ac+bci+adi+bdi²
                ⍝             = (ac-bd)+(bc+ad)i
¯0.9J1.9
```

The absolute value, or magnitude of a complex number is naturally obtained using the Magnitude function

```
      |3j4
5
```

Monadic `+` of a complex number ***(a+bi)*** returns its conjugate ***(a-bi)*** ...

```
      +3j4
3J¯4
```

... which when multiplied by the complex number itself, produces the square of its magnitude.

```
       3j4×3j¯4
25
```

Furthermore, adding a complex number and its conjugate produces a real number:

```
       3j4+3j¯4
6
```

The famous Euler's Identity $e^{i\pi} + 1 = 0$ may be expressed as follows:

```
      1+*○0j1 ⍝ Euler Identity
0
```

Different Result for Power

IFrom Version 13.0 onwards , the implementation of `X*Y` (Power) gives a different answer for negative real `X` than in all previous Versions of Dyalog APL. This change is however in accordance with the ISO/EEC 13751 Standard for Extended APL.

In Version 13.0 onwards, the result is the principal value; whereas in previous Versions the result is a negative or positive real number or `DOMAIN ERROR`. The following examples illustrate this point:

```
      ¯8 * 1 2 ÷ 3          ⍝ Version 12.1
¯2 4
      ¯8 * 1 2 ÷ 3          ⍝ Version 13.0
1J1.732050808 ¯2J3.464101615

      * (1 2 ÷ 3) × ⍟ ¯8    ⍝ Version 13.0
1J1.732050808 ¯2J3.464101615
```

Circular functions

The basic set of circular functions X○Y cater for complex values in Y, while the following extended functions provide specific features for complex arguments. Note that a and b are the real and imaginary parts of Y respectively and θ is the phase of Y..

(-X) ○ Y	X	X ○ Y
-8○Y	8	(-1+Y*2)*0.5
Y	9	a
+Y	10	|Y
Y×0J1	11	b
*Y×0J1	12	θ

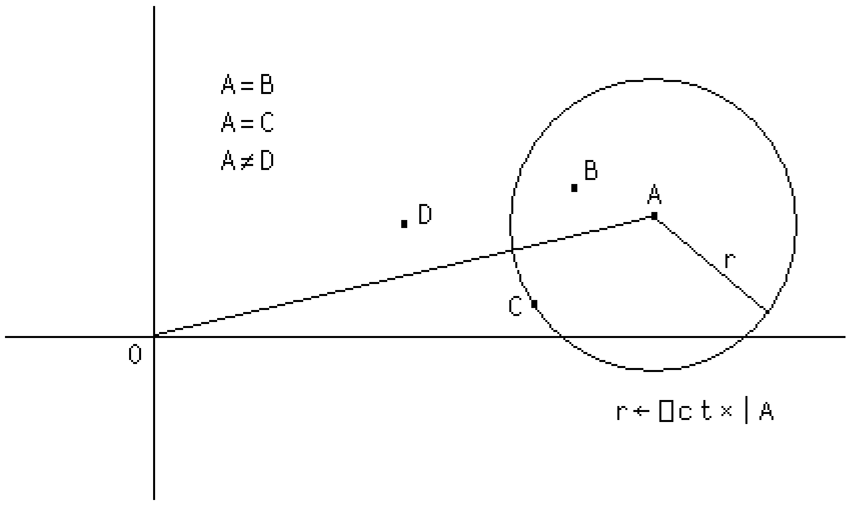
Note that 9○Y and 11○Y return the real and imaginary parts of Y respectively:

```
      9 11○3.5J¯1.2
3.5 ¯1.2

      9 11∘.○3.5J¯1.2 2J3 3J4
 3.5 2 3
¯1.2 3 4
```

Comparison

In comparing two complex numbers X and Y, X=Y is 1 if the magnitude of X-Y does not exceed ⎕CT times the larger of the magnitudes of X and Y; geometrically, X=Y if the number smaller in magnitude lies on or within a circle centred on the one with larger magnitude, having radius ⎕CT times the larger magnitude.



As with real values, complex values sufficiently close to Boolean or integral values are accepted by functions which require Boolean or integral values. For example:

```
      2j1e¯14 ⍴ 12
12 12
      0 ⍱ 1j1e¯15
0
```

Note that Dyalog APL always stores complex numbers as a pair of 64-bit binary floating-point numbers, regardless of the setting of ⎕FR. Comparisons between complex numbers and decimal floating-point numbers will require conversion of the decimal number to binary to allow the comparison. When ⎕FR=1287, comparisons are always subject to ⎕DCT, not ⎕CT - regardless of the data type used to represent a number.

This only really comes into play when determining whether the imaginary part of a complex number is so small that it can be considered to be on the real plane. However, Dyalog recommends that you do not mix the use of complex and decimal numbers in the same component of an application.

128 Bit Decimal Floating-Point Support

Introduction

The original IEE-754 64-bit binary floating point (FP) data type (also known as type number 645), that is used internally by Dyalog APL to represent floating-point values, does not have sufficient precision for certain financial computations – typically involving large currency amounts. The binary representation also causes errors to accumulate even when all values involved in a calculation are "exact" (rounded) decimal numbers, since many decimal numbers cannot be accurately represented regardless of the precision used to hold them. To reduce this problem, Dyalog APL includes support for the 128-bit decimal data type described by IEEE-754-2008 as an alternative representation for floating-point values.

System Variable: ⎕FR

Computations using 128-bit decimal numbers require twice as much space for storage, and run more than an order of magnitude more slowly on platforms which do not provide hardware support for the type. At this time, hardware support is only available from IBM (Power chips starting with the "P6", and recent "z" series mainframes). Even with hardware support, a slowdown of a factor of 4 can be expected. For this reason, Dyalog allows users to decide whether they need the higher-precision decimal representation, or prefer to stay with the faster and smaller binary representation.

A new system variable ⎕FR (for Floating-point Representation) can be set to the value 645 (the installed default) to indicate 64-bit binary FP, or 1287 for 128-bit decimal FP. The default value of ⎕FR is configurable.

Simply put, the value of ⎕FR decides the type of the result of any floating-point calculation that APL performs. In other words, when entered into the session:

```
      ⎕FR = ⎕DR 1.234 ⍝ Type of a floating-point constant
      ⎕FR = ⎕DR 3÷4   ⍝ Type of any floating-point result
```

`⎕FR` has workspace scope, and may be localised. If so, like most other system variables, it inherits its initial value from the global environment.

However: Although `⎕FR` *can* vary, the system is not designed to allow “seamless” modification during the running of an application and the dynamic alteration of `⎕FR` is not recommended. Strange effects may occur. For example, the type of a constant contained in a line of code (in a function or class), will depend on the value of `⎕FR` *when the function is fixed*. Similarly, a constant typed into a line in the Session is evaluated using the value of `⎕FR` that pertained **before** the line is executed. Thus, it would be possible for the first line of code above to return 0, if it is in the body of a function. If the function was edited and while suspended and execution is resumed, the result would become 1. Also note:

```
      ⎕FR←1287
      x←1÷3

      ⎕FR←645
      x=1÷3
1
```

The decimal number has 17 more 3s. Using the tolerance which applies to binary floats (type 645), the numbers are equal. However, the “reverse” experiment yields 0, as tolerance is much narrower in the 128-bit universe:

```
      ⎕FR←645
      x←1÷3

      ⎕FR←1287
      x=1÷3
0
```

Since `⎕FR` can vary, it will be possible for a single workspace to contain floating-point values of both types (existing variables are not converted when `⎕FR` is changed). For example, an array that has just been brought into the workspace from external storage may have a different type from `⎕FR` in the current namespace. Conversion (if necessary) will only take place when a *new* floating-point array is generated as the result of “a calculation”. The result of a computation returning a floating-point result will *not* depend on the type of the arrays involved in the expression: `⎕FR` at the time when a computation is performed decides the result type, alone.

Structural functions generally do NOT change the type, for example:

```
      ⎕FR←1287
      x←1.1 2.2 3.3

      ⎕FR←645
      ⎕dr x
1287
      ⎕dr 2↑x
1287
```

128-bit decimal numbers not only have greater precision (roughly 34 decimal digits); they also have significantly larger range- from `¯1E6145` to `1E6145`. Loss of precision is accepted on conversion from 645 to 1287, but the magnitude of a number may make the conversion impossible, in which case a `DOMAIN ERROR` is issued:

```
      ⎕FR←1287
      x←1E1000

      ⎕FR←645
      x+0
DOMAIN ERROR
```

WARNING: The use of COMPLEX numbers when `⎕FR` is 1287 is not recommended, because:

- any 128-bit decimal array into which a complex number is inserted or appended will be forced in its entirety into complex representation, potentially losing precision
- all comparisons are done using `⎕DCT` when `⎕FR` is 1287, and this is equivalent to 0 for complex numbers.

Conversion between Decimal and Binary

Conversion of data from Binary to Decimal is logically equivalent to formatting, and the reverse conversion is equivalent to evaluating input. These operations are performed according to the same rules that are used when formatting (and evaluating) numbers with `⎕PP` set to 17 (guaranteeing that the decimal value can be converted back to the same binary bit pattern). Because the precision of decimal floating-point numbers is much higher, there will always be a large number of potential decimal values which map to the same binary number: As with formatting, the rule is that the SHORTEST decimal number which maps to a particular binary value will be used as its decimal representation.

Data in component files will be stored without conversion, and only converted when a computation happens. It should be stored in decimal form if it will repeatedly be used by application code in which ⎕FR has the value 1287. Even in applications which use decimal floating point everywhere, reading old component files containing arrays of type 645, or receiving data via ⎕NA, the .Net interface or other external sources, will allow binary floating-point values to enter the system and require conversion.

⎕DCT - Decimal Comparison Tolerance

When ⎕FR has the value 1287, the system variable ⎕DCT will be used to specify comparison tolerance. The default value of ⎕DCT is 1E¯28, and the maximum value is 2.3283064365386962890625E¯10 (the value is chosen to avoid fuzzy comparison of 32-bit integers).

Passing floating-point values using ⎕NA

⎕NA supports the data type "D" to represent the Densely Packed Decimal (DPD) form of 128-bit decimal numbers, as specified by the IEEE-754 2008 standard. Dyalog has decided to use DPD, which is the format used by IBM for hardware support, on ALL platforms, although "Binary Integer Decimal" (BID) is the format that Intel libraries use to implement software libraries to do decimal arithmetic. Experiments have shown that the performance of 128-bit DPD and BID libraries are very similar on Intel platforms. In order to avoid the added complication of having two internal representations, Dyalog has elected to go with the hardware format, which is expected to be adopted by future hardware implementations.

The support libraries for writing AP's and DLL's include new functions to extract the contents of a value of type D as a string or double-precision binary "float" – and convert data to D format.

Decimal Floats and Microsoft.NET

The Microsoft.NET framework contains a type named System.Decimal, which implements decimal floating-point numbers. However, it uses a different internal format from that defined by IEEE-754 2008.

Dyalog APL includes a Microsoft.NET class (called Dyalog.Dec128), which will perform arithmetic on data represented using the "Binary Integer Decimal" format. All computations performed by the Dyalog.Dec128 class will produce exactly the same results as if the computation was performed in APL. A "DCT" property allows setting the comparison tolerance to be used in comparisons, Ceiling/Floor, etc).

The Dyalog class is modelled closely after the existing System.Decimal type, providing the same methods (Add, Ceiling, Compare, CompareTo, Divide, Equals, Finalize, Floor, FromOACurrency, GetBits, GetHashCode, GetType, GetTypeCode, MemberwiseClone, Multiply, Negate, Parse, Remainder, Round, Subtract, To*, Truncate, TryParse) and operators (Addition, Decrement, Division, Equality, Explicit, GreaterThan, GreaterThanOrEqual, Implicit, Increment, Inequality, LessThan, LessThanOrEqual, Modulus, Multiply, Subtraction, UnaryNegation, UnaryPlus).

The “bridge” between Dyalog and .NET is able to cast floating-point numbers to or from System.Double, System.Decimal and Dyalog.Dec128 (and perform all other reasonable casts to integer types etc). Casting a Dyalog.Dec128 to or from strings will perform a “lossless” conversion.

The .Net type System.Int64 will now always be cast to a 128-bit decimal number when entering Dyalog APL, regardless of the setting of ⎕FR. So long as no 64-bit arithmetic is performed on such a value, it will remain a 128-bit number and can be passed back to .Net without loss.

Namespace Syntax

Names within namespaces may be referenced *explicitly* or *implicitly*. An *explicit* reference requires that you identify the object by its full or relative pathname using a `'.'` syntax; for example:

```
      X.NUMB ← 88
```

sets the variable `NUMB` in namespace `X` to 88.

```
      88 UTIL.FOO 99
```

calls dyadic function `FOO` in namespace `UTIL` with left and right arguments of 88 and 99 respectively. The interpreter can distinguish between this use of `'.'` and its use as the inner product operator, because the leftmost name: `UTIL` is a (class 9) namespace, rather than a (class 3) function.

The general namespace reference syntax is:

```
  SPACE . SPACE . (...) EXPR
```

Where `SPACE` is an *expression* which resolves to a namespace reference, and `EXPR` is any APL expression to be resolved in the resulting namespace.

There are two special space names:

`#` is the top level or 'Root' namespace.

`##` is the parent or space containing the current namespace.

`⎕SE` is a system namespace which is preserved across workspace load and clear.

Examples

```
      WSDOC.PAGE.NO +← 1      ⍝ Increment WSDOC page count

      #.⎕NL 2                    ⍝ Variables in root space

      UTIL.⎕FX 'Z←DUP A' 'Z←A A'    ⍝ Fix remote function

      ##.⎕ED'FOO'          ⍝ Edit function in parent space

      ⎕SE.RECORD ← PERS.RECORD    ⍝ Copy from PERS to ⎕SE

      UTIL.(⎕EX ⎕NL 2)        ⍝ Expunge variables in UTIL


      (⊃⎕SE #).(⍎⊃↓⎕NL 9).(⎕NL 2)     ⍝ Vars in first ⎕SE
                                      ⍝ namespace.

      UTIL.⍎STRING         ⍝ Execute STRING in UTIL space
```

You may also reference a function or operator in a namespace *implicitly* using the mechanism provided by `⎕EXPORT` (See *Language Reference*) and `⎕PATH`. If you reference a name that is undefined in the current space, the system searches for it in the list of exported names defined for the namespaces specified by `⎕PATH`. See.*Language Reference* for further details.

Notice that the expression to the right of a dot may be arbitrarily complex and will be executed within the namespace or ref to the left of the dot.

```
      X.(C←A×B)
      X.C
10 12 14
16 18 20
      NS1.C
10 12 14
16 18 20
```

Summary

Apart from its use as a decimal separator (`3.14`), '`.`' is interpreted by looking at the type or *class* of the expression to its left:

Template	Interpretation	Example
`∘.`	Outer product	`2 3 ∘.× 4 5`
`function.`	Inner product	`2 3 +.× 4 5`
`ref.`	Namespace reference	`2 3 x.foo 4 5`
`array.`	Reference array expansion	`(x y).⎕nc⊂'foo'`

Namespace Reference Evaluation

When the interpreter encounters a namespace reference, it:

1. Switches to the namespace.
2. Evaluates the name.
3. Switches back to the original namespace.

If for example, in the following, the current namespace is `#.W`, the interpreter evaluates the line:

```
      A ← X.Y.DUP MAT
```

in the following way:

1. Evaluate array `MAT` in current namespace `W` to produce argument for function.
2. Switch to namespace `X.Y` within `W`.
3. Evaluate function `DUP` in namespace `W.X.Y` with argument.
4. Switch back to namespace `W`.
5. Assign variable `A` in namespace `W`.

Namespaces and Localisation

The rules for name resolution have been generalised for namespaces.

In flat APL, the interpreter searches the state indicator to resolve names referenced by a defined function or operator. If the name does not appear in the state indicator, then the workspace-global name is assumed.

With namespaces, a defined function or operator is evaluated in its 'home' namespace. When a name is referenced, the interpreter searches only those lines of the state indicator which belong to the home namespace. If the name does not appear in any of these lines, the home namespace-global value is assumed.

For example, if `#.FN1` calls `XX.FN2` calls `#.FN3` calls `XX.FN4`, then:

`FN1`:
: is evaluated in `#`
: can see its own dynamic local names
: can see global names in `#`

`FN2`:
: is evaluated in `XX`
: can see its own dynamic local names
: can see global names in `XX`

`FN3`:
: is evaluated in `#`
: can see its own dynamic local names
: can see dynamic local names in `FN1`
: can see global names in `#`

`FN4`:
: is evaluated in `XX`
: can see its own dynamic local names
: can see dynamic local names in `FN2`
: can see global names in `XX`

Namespace References

A *namespace reference*, or *ref* for short, is a unique data type that is distinct from and in addition to *number* and *character*.

Any expression may result in a ref, but the simplest one is the namespace itself:

```
      )NS NS1           ⍝ Make a namespace called NS1
      NS1.A←1           ⍝ and populate it with variables A
      NS1.B←2 3⍴⍳6      ⍝ and B

      NS1               ⍝ expression results in a ref
#.NS1
```

You may assign a ref; for example:

```
      X←NS1
      X
#.NS1
```

In this case, the display of `X` informs you that `X` refers to the named namespace `#.NS1`.

You may also supply a ref as an argument to a defined or dynamic function:

```
     ∇ FOO ARG
[1]    ARG
     ∇

      FOO NS1
#.NS1
```

The name class of a *ref* is 9.

```
      ⎕NC 'X'
9
```

You may use a ref to a namespace anywhere that you would use the namespace itself. For example:

```
      X.A
1
      X.B
1 2 3
4 5 6
```

Notice that refs are references to namespaces, so that if you make a copy, it is the reference that is copied, not the namespace itself. This is sometimes referred to as a shallow as opposed to a deep copy. It means that if you change a ref, you actually change the namespace that it refers to.

```
      X.A+←1
      X.A
2
      NS1.A
2
```

Similarly, a ref passed to a defined function is call-by-reference, so that modifications to the content or properties of the argument namespace using the passed reference persist after the function exits. For example:

```
     ∇ FOO nsref
[1]    nsref.B+←nsref.A
     ∇

      FOO NS1
      NS1.B
3 4 5
6 7 8
      FOO X
      NS1.B
5 6  7
8 9 10
```

Notice that the expression to the right of a dot may be arbitrarily complex and will be executed within the namespace or ref to the left of the dot.

```
      X.(C←A×B)
      X.C
10 12 14
16 18 20
      NS1.C
10 12 14
16 18 20
```

Unnamed Namespaces

The monadic form of ⎕NS makes a new (and unique) unnamed namespace and returns a ref to it.

One use of unnamed namespaces is to represent hierarchical data structures; for example, a simple employee database:

The first record is represented by JOHN which is a ref to an *unnamed* namespace:

```
      JOHN←⎕NS ''
      JOHN
#.[Namespace]

      JOHN.FirstName←'John'
      JOHN.FirstName
John

      JOHN.LastName←'Smith'
      JOHN.Age←50
```

Data variables for the second record, PAUL, can be established using strand, or vector, assignment:

```
      PAUL←⎕NS ''
      PAUL.(FirstName LastName Age←'Paul' 'Brown' 44)
```

The function SHOW can be used to display the data in each record (the function is split into 2 lines only to fit on the printed page). Notice that its argument is a ref.

```
     ∇ R←SHOW PERSON
[1]    R←PERSON.FirstName,' ',PERSON.LastName
[2]    R, ←' is ',⍕PERSON.Age
     ∇

      SHOW JOHN
John Smith is 50

      SHOW PAUL
Paul Brown is 44
```

An alternative version of the function illustrates the use of the `:With :EndWith` control structure to execute an expression, or block of expressions, within a namespace:

```
     ∇ R←SHOW1 PERSON
[1]    :With PERSON
[2]        R←FirstName,' ',LastName,' is ',(⍕Age)
[3]    :EndWith
     ∇

      SHOW1 JOHN
John Smith is 50
```

In this case, as only a single expression is involved, it can be expressed more simply using parentheses.

```
     ∇ R←SHOW2 PERSON
[1]    R←PERSON.(FirstName,' ',LastName,' is ',(⍕Age))
     ∇
      SHOW2 PAUL
Paul Brown is 44
```

Dynamic functions also accept refs as arguments:

```
      SHOW3←{
        ⍵.(FirstName,' ',LastName,' is ',⍕Age)
      }

      SHOW3 JOHN
John Smith is 50
```

Arrays of Namespace References

You may construct arrays of refs using strand notation, catenate (`,`) and reshape (`⍴`).

```
      EMP←JOHN PAUL
      ⍴EMP
```

2

```
      EMP
 #.[Namespace]  #.[Namespace]
```

Like any other array, an array of refs has name class 2:

```
      ⎕NC 'EMP'
2
```

Expressions such as indexing and pick return refs that may in turn be used as follows:

```
      EMP[1].FirstName
John
      (2⊃EMP).Age
44
```

The each (`¨`) operator may be used to apply a function to an array of refs:

```
      SHOW¨EMP
 John Smith is 50  Paul Brown is 44
```

An *array* of namespace references (refs) to the left of a '`.`' is expanded according to the following rule, where `x` and `y` are refs, and `exp` is an arbitrary expression:

```
      (x y).exp → (x.exp)(y.exp)
```

If `exp` evaluates to a function, the items of its argument array(s) are *distributed* to each referenced function. In the dyadic case, there is a 3-way distribution among: left argument, referenced functions and right argument.

Monadic function `f`:

```
      (x y).f d e → (x.f d)(y.f e)
```

Dyadic function `g`:

```
      a b (x y).g  d e → (a x.g d)(b y.g e)
```

An array of refs to the left of an assignment arrow is expanded thus:

```
      (x y).a←c d   →  (x.a←c)(y.a←d)
```

Note that the array of refs can be of any rank. In the limiting case of a simple scalar array, the *array* construct: `refs.exp` is identical to the *scalar* construct: `ref.exp`.

Note that the expression to the right of the '`.`' *pervades* a nested array of refs to its left:

```
      ((u v)(x y)).exp → ((u.exp)(v.exp))((x.exp)(y.exp))
```

Note also that with *successive* expansions `(u v).(x y z).` ..., the final number of 'leaf' terms is the *product* of the number of refs at each level.

Examples:

```
      JOHN.Children←⎕NS¨'' ''
      ⍴JOHN.Children
2
      JOHN.Children[1].FirstName←'Andy'
      JOHN.Children[1].Age←23

      JOHN.Children[2].FirstName←'Katherine'
      JOHN.Children[2].Age←19

      PAUL.Children←⎕NS¨'' ''
      PAUL.Children[1].(FirstName Age←'Tom' 25)
      PAUL.Children[2].(FirstName Age←'Jamie' 22)

      ⍴EMP
2
      (⊃EMP).Children.(FirstName Age)
 Andy  23   Katherine  19

      ]display (2⊃EMP).Children.(FirstName Age)
.→----------------------------.
| .→---------. .→-----------. |
| | .→--.    | | .→----.    | |
| | |Tom| 25 | | |Jamie| 22 | |
| | '---'    | | '-----'    | |
| '∊---------' '∊-----------' |
'∊----------------------------'

      EMP.Children ⍝ Is an array of refs
 #.[Namespace]  #.[Namespace]   #.[Namespace]  ...

      EMP.Children.(FirstName Age)
  Andy  23   Katherine  19    Tom  25   Jamie  22
```

Distributed Assignment

Assignment pervades nested strands of names to the left of the arrow. The conformability rules are the same as for scalar (pervasive) dyadic primitive functions such as '+'. The mechanism can be viewed as a way of naming the parts of a *structure*.

Examples:

```
      EMP.(FirstName Age)
 JOHN  43   PAUL  44

      EMP.(FirstName Age)←('Jonathan' 21)('Pauline' 22)

      EMP.(FirstName Age)
 Johnathan  21   Pauline  22

⍝ Distributed assignment is pervasive
      JOHN.Children.(FirstName Age)
 Andy  23   Katherine  19

      JOHN.Children.(FirstName Age)←('Andrew' 21)('Kate'
9)

      JOHN.Children.(FirstName Age)
 Andrew  21   Kate  9
```

More Examples:

```
      ((a b)(c d))←(1 2)(3 4)   ⍝ a←1 ⋄ b←2 ⋄ c←3 ⋄ d←4

      ((⎕io ⎕ml)vec)←0 ⎕av      ⍝ ⎕io←0 ⋄ ⎕ml←0 ⋄ vec←⎕av

      (i (j k))+←1 2            ⍝ i+←1 ⋄ j+←2 ⋄ k+←2

⍝ Naming of parts:

      ((first last) sex (street city state))←n⊃pvec

⍝ Distributed assignment in :For loop:

      :For (i j)(k l) :In array

⍝ Ref array expansion:

      (x y).(first last)←('John' 'Doe')('Joe' 'Blow')
      (f1 f2).(b1 b2).Caption←⊂'OK' 'Cancel'
```

```
⍝ Structure rearrangement:
      rotate1←{       ⍝ Simple binary tree rotation.
           (a b c)d e←⍵
           a b(c d e)
      }
      rotate3←{       ⍝ Compound binary tree rotation.
           (a b(c d e))f g←⍵
           (a b c)d(e f g)
      }
```

Distributed Functions

Namespace ref array expansion syntax applies to functions too.

```
      JOHN.PLOT←{↑⍵⍴¨'⎕'}
      JOHN.PLOT ⍳10
⎕
⎕⎕
⎕⎕⎕
⎕⎕⎕⎕
⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕

      PAUL.PLOT←{(⍵,¨1)⍴¨'⎕'}
      PAUL.PLOT ⍳10
 ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
    ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
       ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
          ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
             ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
                ⎕  ⎕  ⎕  ⎕  ⎕
                   ⎕  ⎕  ⎕  ⎕
                      ⎕  ⎕  ⎕
                         ⎕  ⎕
                            ⎕

      EMP.PLOT⊂⍳10   ⍝ (temporary vector of functions)
 ⎕            ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕              ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕                ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕⎕                  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕⎕⎕                    ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕⎕⎕⎕                      ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕⎕⎕⎕⎕                        ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕⎕⎕⎕⎕⎕                          ⎕  ⎕  ⎕
 ⎕⎕⎕⎕⎕⎕⎕⎕⎕                            ⎕  ⎕
 ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕                              ⎕
```

```
      (x y).⎕NL 2 3                ⍝ x:vars, y:fns
 varx  funy

      (x y).⎕NL⊂2 3                ⍝ x&y: vars&fns
 funx  funy
 varx  vary

      (x y).(⎕NL¨)⊂2 3             ⍝ x&y: separate
vars&fns
  varx  funx    vary  funy

      'v'(x y).⎕NL 2 3             ⍝ x:v-vars, y:v-fns
 varx

      'vf'(x y).⎕NL 2 3            ⍝ x:v-vars, y:f-fns
  varx  funy
                                   ⍝ x:v-vars&fns,
      'vf'(x y).⎕NL⊂2 3            ⍝ y:f-vars&fns
  varx  funy

      x.⎕NL 2 3                    ⍝ depth 0 ref
funx
varx

      (x y).⎕NL⊂2 3                ⍝ depth 1 refs
 funx  funy
 varx  vary

      ((u v)(x y)).⎕NL⊂⊂2 3        ⍝ depth 2 refs
  funu  funv    funx  funy
  varu  varv    varx  vary

      (1 2)3 4(w(x y)z).+1 2(3 4) ⍝ argument
distribution.
 2 3  5 5  7 8
```

Namespaces and Operators

A function passed as operand to a primitive or defined operator, carries its namespace context with it. This means that if subsequently, the function operand is applied to an argument, it executes in its home namespace, irrespective of the namespace from which the operator was invoked or defined.

Examples

```
      VAR←99                       ⍝ #.VAR

      )NS X
#.X
      X.VAR←77                     ⍝ X.VAR
      X.⎕FX'Z←FN R' 'Z←R,VAR'

      )NS Y
#.Y
      Y.VAR←88                     ⍝ Y.VAR
      Y.⎕FX'Z←(F OP)R' 'Z←F R'

      X.FN¨⍳3
 1 77  2 77  3 77

      X.FN 'VAR:'
 VAR: 77

      X.FN Y.OP 'VAR:'
 VAR: 77
      ⍎ Y.OP'VAR'
99
```

Threads

Overview

Dyalog APL supports multithreading - the ability to run more than one APL expression at the same time.

This unique capability allows you to perform background processing, such as printing, database retrieval, database update, calculations, and so forth while at the same time perform other interactive tasks.

Multithreading may be used to improve throughput and system responsiveness.

A *thread* is a strand of execution in the APL workspace.

A thread is created by calling a function *asynchronously*, using the new primitive operator 'spawn': *&* or by the asynchronous invocation of a callback function.

With a traditional APL *synchronous* function call, execution of the calling environment is paused, *pendent* on the return of the called function. With an *asynchronous* call, both calling environment and called function proceed to execute concurrently.

An asynchronous function call is said to start a new *thread* of execution. Each thread has a unique *thread number*, with which, for example, its presence can be monitored or its execution terminated.

Any thread can spawn any number of sub-threads, subject only to workspace availability. This implies a hierarchy in which a thread is said to be a *child thread* of its *parent thread*. The *base thread* at the root of this hierarchy has thread number 0.

With multithreading, APL's stack or state indicator can be viewed as a branching tree in which the path from the base to each leaf is a thread.

When a parent thread terminates, any of its children which are still running, become the children of (are ‘adopted’ by) the parent’s parent.

Thread numbers are allocated sequentially from 0 to 2147483647. At this point, the sequence ‘wraps around’ and numbers are allocated from 0 again avoiding any still in use. The sequence is reinitialised when a `)RESET` command is issued, or the active workspace is cleared, or a new workspace is loaded. A workspace may not be saved with threads other than the base thread: 0, running.

Multi-Threading language elements.

The following language elements are provided to support threads.

- Primitive operator, spawn: `&`.
- System functions: `⎕TID`, `⎕TCNUMS`, `⎕TNUMS`, `⎕TKILL`, `⎕TSYNC`.
- An extension to the GUI Event syntax to allow asynchronous callbacks.
- A control structure: `:Hold`.
- System commands: `)HOLDS`, `)TID`.
- Extended `)SI` and `)SINL` display.

Running CallBack Functions as Threads

A callback function is associated with a particular event via the Event property of the object concerned. A callback function is executed by `⎕DQ` when the event occurs, or by `⎕NQ`.

If you append the character `&` to the name of the callback function in the `Event` specification, the callback function will be executed asynchronously as a thread when the event occurs. If not, it is executed synchronously as before.

For example, the event specification:

```
      ⎕WS'Event' 'Select' 'DoIt&'
```

tells `⎕DQ` to execute the callback function `DoIt` *asynchronously as a thread* when a Select event occurs on the object.

Thread Switching

Programming with threads requires care.

The interpreter may switch between running threads at the following points:

- Between any two lines of a defined (or dynamic) function or operator.
- While waiting for a `⎕DL` to complete.
- While waiting for a `⎕FHOLD` to complete.
- While awaiting input from:
  - `⎕DQ`
  - `⎕SR`
  - `⎕ED`
- The session prompt or `⍞:` or `⎕.`
- While awaiting the completion of an external operation:
  - A call on an external (AP) function.
  - A call on a `⎕NA` (DLL) function
  - A call on an OLE function.
  - A call on a .Net function.

At any of these points, the interpreter might execute code in other threads. If such threads change the global environment; for example by changing the value of, or expunging a name; then the changes will appear to have happened while the thread in question passes through the switch point. It is the task of the application programmer to organise and contain such behaviour!

You can prevent threads from interacting in critical sections of code by using the `:Hold` control structure.

High Priority Callback Functions

Note that the interpreter cannot perform thread-switching during the execution of a *high-priority callback*. This is a callback function that is invoked by a *high-priority* event which demands that the interpreter must return a result to Windows before it may process any other event. Such high-priority events include Configure, Exit-Windows, DateTimeChange, DockStart, DockCancel, DropDown. It is therefore not permitted to use a `:Hold` control structure in a high-priority callback function.

Name Scope

APL's name scope rules apply whether a function call is synchronous or asynchronous. For example when a defined function is called, names in the calling environment are visible, unless explicitly shadowed in the function header.

Just as with a synchronous call, a function called asynchronously has its own local environment, but can communicate with its parent and 'sibling' functions via local names in the parent.

This point is important. It means that siblings can run in parallel without danger of local name clashes. For example, a GUI application can accommodate multiple concurrent instances of its callback functions.

However, with an asynchronous call, as the calling function continues to execute, both child *and parent functions* may modify values in the calling environment. Both functions see such changes immediately they occur.

If a parent function terminates while any of its children are still running, those children will thenceforward 'see' local names in the environment that called the parent function. In cases where a child function relies on its parent's environment (the setting of a local value of `⎕IO` for example), this would be undesirable, and the parent function would normally execute a `⎕TSYNC` in order to wait for its children to complete before itself exiting.

If, on the other hand, after launching an asynchronous child, the parent function calls a *new* function (either synchronously or asynchronously); names in the new function are beyond the purview of the original child. In other words, a function can only ever see its calling stack decrease in size – never increase. This is in order that the parent may call new defined functions without affecting the environment of its asynchronous children.

Using Threads

Put most simply, multithreading allows you to *appear to* run more than one APL function at the same time, just as Windows (or UNIX) *appears to* run more than one application at the same time. In both cases this is something of an illusion, although it does nothing to detract from its usefulness.

Dyalog APL implements an internal timesharing mechanism whereby it shares processing between threads. Although the mechanics are somewhat different, APL multithreading is rather similar to the multitasking provided by Windows. If you are running more than one application, Windows switches from one to another, allocating each one a certain *time slice* before switching. At any point in time, only one application is actually running; the others are paused, waiting.

If you execute more than one Dyalog APL thread, only one thread is actually running; the others are paused. Each APL thread has its own State Indicator, or SI stack. When APL switches from one thread to another, it saves the current stack (with all its local variables and function calls), restores the new one, and then continues processing.

Stack Considerations

When you start a thread, it begins with the SI stack of the calling function and sees all of the local variables defined in all the functions down the stack. However, unless the calling function specifically waits for the new thread to terminate (see *Language Reference*), the calling functions will (bit by bit, in their turn) continue to execute. The new thread's view of its calling environment may then change. Consider the following example:

Suppose that you had the following functions: `RUN[3]` calls `INIT` which in turn calls `GETDATA` but as 3 separate threads with 3 different arguments:

```
     ∇ RUN;A;B
[1]    A←1
[2]    B←'Hello World'
[3]    INIT
[4]    CALC
[5]    REPORT
     ∇
     ∇ INIT;C;D
[1]    C←D←0
[2]    GETDATA&'Sales'
[3]    GETDATA&'Costs'
[4]    GETDATA&'Expenses'
     ∇
```

When each `GETDATA` thread starts, it immediately *sees* (via `⎕SI`) that it was called by `INIT` which was in turn called by `RUN`, and it *sees* local variables `A`, `B`, `C` and `D`. However, once `INIT[4]` has been executed, `INIT` terminates, and execution of the root thread continues by calling `CALC`. From then on, each `GETDATA` thread no longer sees `INIT` (it thinks that it was called directly from `RUN`) nor can it see the local variables `C` and `D` that `INIT` had defined. However, it *does* continue to see the locals `A` and `B` defined by `RUN`, until `RUN` itself terminates.

Note that if `CALC` were also to define locals `A` and `B`, the `GETDATA` threads would still see the values defined by `RUN` and not those defined by `CALC`. However, if `CALC` were to modify `A` and `B` (as globals) without localising them, the `GETDATA` threads would see the modified values of these variables, whatever they happened to be at the time.

Globals and the Order of Execution

It is important to recognise that any reference or assignment to a global or semi-global object (including GUI objects) is **inherently dangerous** (i.e. a source of programming error) if more than one thread is running. Worse still, programming errors of this sort may not become apparent during testing because they are dependent upon random timing differences. Consider the following example:

```
     ∇ BUG;SEMI_GLOBAL
[1]    SEMI_GLOBAL←0
[2]    FOO& 1
[3]    GOO& 1
     ∇

     ∇ FOO
[1]    :If SEMI_GLOBAL=0
[2]        DO_SOMETHING SEMI_GLOBAL
[3]    :Else
[4]        DO_SOMETHING_ELSE SEMI_GLOBAL
[5]    :EndIf
     ∇

     ∇ GOO
[1]    SEMI_GLOBAL←1
     ∇
```

In this example, it is formally impossible to predict in which order APL will execute statements in `BUG`, `FOO` or `GOO` from `BUG[2]` onwards. For example, the actual sequence of execution may be:

```
      BUG[1] → BUG[2] → FOO[1] → FOO[2] →
               DO_SOMETHING[1]
```

or

```
      BUG[1] → BUG[2] → BUG[3] → GOO[1] →
               FOO[1] → FOO[2] → FOO[3] →
               FOO[4] → DO_SOMETHING_ELSE[1]
```

This is because APL may switch from one thread to another between any two lines in a defined function. In practice, because APL gives each thread a significant time-slice, it is likely to execute many lines, maybe even hundreds of lines, in one thread before switching to another. However, you must not rely on this; **thread-switching may occur at any time between lines in a defined function**.

Secondly, consider the possibility that APL switches from the `FOO` thread to the `GOO` thread after `FOO[1]`. If this happens, the value of `SEMI_GLOBAL` passed to `DO_SOMETHING` will be 1 and not 0. Here is another source of error.

In fact, in this case, there are two ways to resolve the problem. To ensure that the value of `SEMI_GLOBAL` remains the same from `FOO[1]` to `FOO[2]`, you may use diamonds instead of separate statements, e.g.

```
      :If SEMI_GLOBAL=0 ⋄ DO_SOMETHING SEMI_GLOBAL
```

Even better, although less efficient, you may use `:Hold` to synchronise access to the variable, for example:

```
      ∇ FOO
[1]    :Hold 'SEMI_GLOBAL'
[2]        :If SEMI_GLOBAL=0
[3]            DO_SOMETHING SEMI_GLOBAL
[4]        :Else
[5]            DO_SOMETHING_ELSE SEMI_GLOBAL
[6]        :EndIf
[7]    :EndHold
     ∇

     ∇ GOO
[1]    :Hold 'SEMI_GLOBAL'
[2]        SEMI_GLOBAL←1
[3]    :EndHold
     ∇
```

Now, although you still cannot be sure which of FOO and GOO will run first, you can be sure that SEMI_GLOBAL will not change (because GOO cuts in) within FOO.

Note that the string used as the argument to :Hold is completely arbitrary, so long as threads competing for the same resource use the same string.

A Caution

These types of problems are inherent in all multithreading programming languages, and not just with Dyalog APL. ***If you want to take advantage of the additional power provided by multithreading, it is advisable to think carefully about the potential interaction between different threads.***

Threads & Niladic Functions

- In common with other operators, the spawn operator & may accept monadic or dyadic functions as operands, but not niladic functions. This means that, using spawn, you cannot start a thread that consists only of a niladic function
- If you wish to invoke a niladic function asynchronously, you have the following choices:
- Turn your niladic function into a monadic function by giving it a dummy argument which it ignores.
- Call your niladic function with a dynamic function to which you give an argument that is implicitly ignored. For example, if the function NIL is niladic, you can call it asynchronously using the expression: {NIL}& 0
- Call your function via a dummy monadic function, e.g.

```
      ∇ NIL_M DUMMY
[1]     NIL
      ∇
      NIL_M& ''
```

- Use execute, e.g.

```
⍎& 'NIL'
```

Note that niladic functions *can* be invoked asynchronously as callback functions. For example, the statement:

```
      ⎕WS'Event' 'Select' 'NIL&'
```

will execute correctly as a thread, even though NIL is niladic. This is because callback functions are invoked directly by ⎕DQ rather than as an operand to the spawn operator.

Threads & External Functions

External functions in dynamic link libraries (DLLs) defined using the ⎕NA interface may be run in separate C threads. Such threads:

- **take advantage of multiple processors** if the operating system permits.
- allow APL to **continue processing in parallel** during the execution of a ⎕NA function.

When you define an external function using ⎕NA, you may specify that the function be run in a separate C thread by appending an ampersand (&) to the function name, for example:

```
      'beep'⎕NA'user32|MessageBeep& i'
      ⍝ MessageBeep will run in a separate C thread
```

When APL first comes to execute a multi-threaded ⎕NA function, it starts a new C-thread, executes the function within it, and waits for the result. Other APL threads may then run in parallel.

Note that when the ⎕NA call finishes and returns its result, its new C-thread is retained to be re-used by any subsequent multithreaded ⎕NA calls made within the same APL thread. Thus any APL thread that makes any multi-threaded ⎕NA calls maintains a separate C-thread for their execution. This C-thread is discarded when its APL thread finishes.

Note that there is no point in specifying a ⎕NA call to be multi-threaded, unless you wish to execute other APL threads at the same time.

In addition, if your ⎕NA call needs to access an APL GUI object (strictly, a window or other handle) it should normally run within the same C-thread as APL itself, and not in a separate C-thread. This is because Windows associates objects with the C-thread that created them. Although you *can* use a multi-threaded ⎕NA call to access (say) a Dyalog APL Form via its window handle, the effects may be different than if the ⎕NA call was not multi-threaded. In general, ⎕NA calls that access APL (GUI) objects should not be multi-threaded.

If you wish to run the same ⎕NA call in separate APL threads at the same time, you must ensure that the DLL is *thread-safe*. Functions in DLLs which are not *thread-safe*, must be prevented from running concurrently by using the `:Hold` control structure. Note that all the standard Windows API DLLs **are** *thread safe*.

Notice that you may define two separate functions (with different names), one single-threaded and one multi-threaded, associated with the same function in the DLL. This allows you to call it in either way.

Synchronising Threads

Threads may be synchronised using *tokens* and a *token pool*.

An application can synchronise its threads by having one thread add tokens into the pool whilst other threads wait for tokens to become available and retrieve them from the pool.

Tokens possess two separate attributes, a *type* and a *value*.

The *type* of a token is a positive or negative integer scalar. The *value* of a token is any arbitrary array that you might wish to associate with it.

The token pool may contain up to 2*31 tokens; they do not have to be unique neither in terms of their types nor of their values.

The following system functions are used to manage the token pool:

`⎕TPUT`	Puts tokens into the pool.
`⎕TGET`	If necessary waits for, and then retrieves some tokens from the pool.
`⎕TPOOL`	Reports the types of tokens in the pool
`⎕TREQ`	Reports the token requests from specific threads

A simple example of a thread synchronisation requirement occurs when you want one thread to reach a certain point in processing before a second thread can continue. Perhaps the first thread performs a calculation, and the second thread must wait until the result is available before it can be used.

This can be achieved by having the first thread put a specific type of token into the pool using `⎕TPUT`. The second thread waits (if necessary) for the new value to be available by calling `⎕TGET` with the same token type.

Notice that when `⎕TGET` returns, the specified tokens are *removed* from the pool. However, *negative* token types will satisfy an infinite number of requests for their positive equivalents.

The system is designed to cater for more complex forms of synchronisation. For example, a *semaphore* to control a number of resources can be implemented by keeping that number of tokens in the pool. Each thread will take a token while processing, and return it to the pool when it has finished.

A second complex example is that of a *latch* which holds back a number of threads until the coast is clear. At a signal from another thread, the latch is opened so that all of the threads are released. The latch may (or may not) then be closed again to hold up subsequently arriving threads. A practical example of a latch is a ferry terminal.

Semaphore Example

A *semaphore* to control a number of resources can be implemented by keeping that number of tokens in the pool. Each thread will take a token while processing, and return it to the pool when it has finished.

For example, if we want to restrict the number of threads that can have sockets open at any one time.

```
    sock←99                  ⍝ socket-token
                               any +ive number will do).
    ⎕TPUT 5/sock             ⍝ add 5 socket-tokens to
pool.

    ∇ sock_open ...
[1]   :If sock=⎕TGET sock    ⍝ grap a socket token
[.]       ...                ⍝ do stuff.
[.]       ⎕TPUT sock         ⍝ release socket token
[.]   :Else
[.]       error'sockets off' ⍝ sockets switched off by
                               retract (see below).
[.]   :EndIf
    ∇

    0 ⎕TPUT ⎕treq ⎕tnums     ⍝ retract socket "service"
                               with 0 value.
```

Latch Example

A *latch* holds back a number of threads until the coast is clear. At a signal from another thread, the latch is opened so that all of the threads are released. The latch may (or may not) then be closed again to hold up subsequently arriving threads.

A visual example of a latch might be a ferry terminal, where cars accumulate in the queue until the ferry arrives. The barrier is then opened and all (up to a maximum number) of the cars are allowed through it and on to the ferry. When the last car is through, the barrier is re-closed.

```
    tkt←6                         ⍝ 6-token: ferry
ticket.

    ∇ car ...
[1]   ⎕TGET tkt                   ⍝ await ferry.
[2]   ...

    ∇ ferry ...
[1]   arrives in port
[2]   ⎕TPUT(↑,/⎕treq ⎕tnums)⍴tkt  ⍝ ferry tickets for
all.
[3]   ...
```

Note that it is easy to modify this example to provide a maximum number of ferry places per trip by inserting `max_places↑` between `⎕TPUT` and its argument. If fewer cars than the ferry capacity are waiting, the `↑` will fill with trailing 0s. This will not cause problems because zero tokens are ignored.

Let us replace the car ferry with a new road bridge. Once the bridge is ready for traffic, the barrier could be opened permanently by putting a *negative* ticket in the pool.

```
    ⎕TPUT -tkt      ⍝ open ferry barrier permananently.
```

Cars could choose to take the last ferry if there are places:

```
    ∇ car ...
[1]   :Select ⎕TGET tkt
[2]   :Case  tkt ⋄ take the last ferry.
[3]   :Case -tkt ⋄ ferry full: take the new bridge.
[4]   :End
```

The above `:Select` works because by default, `⎕TPUT -tkt` puts a *value* of `-tkt` into the token.

Debugging Threads

If a thread sustains an untrapped error, its execution is *suspended* in the normal way. If the *Pause on Error* option (see User Guide) is set, all other threads are *paused*. If *Pause on Error* option (see User Guide) is not set, other threads will continue running and it is possible for another thread to encounter an error and suspend.

Using the facilities provided by the Tracer and the Threads Tool (see User Guide) it is possible to interrupt (suspend) and restart individual threads, and to pause and resume individual threads, so any thread may be in one of three states - *running*, *suspended* or *paused*.

The Tracer and the Session may be connected with any suspended thread and you can switch the attention of the Session and the Tracer between suspended threads using `)TID` or by clicking on the appropriate tab in the Tracer. At this point, you may:

- Examine and modify local variables for the currently suspended thread.
- Trace and edit functions in the current thread.
- Cut back the stack in the currently suspended thread.
- Restart execution.
- Start new threads

The error message from a thread other than the base is prefixed with its thread number:

```
260:DOMAIN ERROR
Div[2] rslt←num÷div
      ^
```

State indicator displays: `)SI` and `)SINL` have been extended to show threads' tree-like calling structure.

```
      )SI
·    #.Calc[1]
&5
·    ·    #.DivSub[1]
·    &7
·    ·    #.DivSub[1]
·    &6
·    #.Div[2]*
&4
#.Sub[3]
#.Main[4]
```

Here, `Main` has called `Sub`, which has spawned threads `4` and `5` with functions: `Div` and `Calc`. Function `Div`, after spawning `DivSub` in each of threads `6` and `7`, have been suspended at line `[2]`.

Removing stack frames using *Quit* from the Tracer or `→` from the session affects only the current thread. When the final stack frame in a thread (other than the base thread) is removed, the thread is expunged.

`)RESET` removes all but the base thread.

Note the distinction between a *suspended* thread and a *paused* thread.

A *suspended* thread is stopped at the beginning of a line in a defined function or operator. It may be connected to the Session so that expressions executed in the Session do so in the context of that thread. It may be *restarted* by executing `→line` (typically, `→⎕LC`).

A *paused* thread is an inactive thread that is currently being ignored by the thread scheduler. A paused thread may be paused within a call to `⎕DQ`, a call on an external function, at the beginning of a line, or indeed at any of the thread-switching points described earlier in this chapter.

A paused thread may be *resumed* only by the action of a menu item or button. A paused thread resumes only in the sense that it ceases to be ignored by the thread scheduler and will therefore be switched back to at some point in the future. It does not actually continue executing until the switch occurs.

External Variables

An external variable is a variable whose contents (value) reside not in the workspace, but in a file. An external variable is associated with a file by the system function `⎕XT`. If at the time of association the file exists, the external variable assumes its value from the contents of the file. If the file does not exist, the external variable is defined but a `VALUE ERROR` occurs if it is referenced before assignment. Assignment of an array to the external variable or to an indexed element of the external variable has the effect of updating the file. The value of the external variable or the value of indexed elements of the external variable is made available in the workspace when the external variable occurs in an expression. No special restrictions are placed on the usage of external variables.

Normally, the files associated with external variables remain permanent in that they survive the APL session or the erasing of the external variable from the workspace. External variables may be accessed concurrently by several users, or by different nodes on a network, provided that the appropriate file access controls are established. Multi-user access to an external variable may be controlled with the system function `⎕FHOLD` between co-operating tasks.

Refer to the sections describing the system functions `⎕XT` and `⎕FHOLD` in *Chapter 6* for further details.

Examples

```
      'ARRAY' ⎕XT 'V'

      V←⍳10
      V[2] + 5
7

      ⎕EX'V'

      'ARRAY' ⎕XT 'F'
      F
1 2 3 4 5 6 7 8 9 10
```

Component Files

A component file is a data file maintained by Dyalog APL. It contains a series of APL arrays known as components which are accessed by reference to their relative positions or component number within the file. A set of system functions is provided to perform a range of file operations. (See *Language Reference*.) These provide facilities to create or delete files, and to read and write components. Facilities are also provided for multi-user access including the capability to determine who may do what, and file locking for concurrent updates. (See *User Guide*.)

Auxiliary Processors

Auxiliary Processors (APs) are non-APL programs which provide Dyalog APL users with additional facilities. They run as separate tasks, and communicate with the Dyalog APL interpreter through pipes (UNIX) or via an area of memory (Windows). Typically, APs are used where speed of execution is critical, such as in screen management software, or for utility libraries. Auxiliary Processors may be written in any compiled language, although 'C' is preferred and is directly supported.

When an Auxiliary Processor is invoked from Dyalog APL, one or more *external functions* are fixed in the active workspace. Each external function behaves as if it was a locked defined function, but is in effect an entry point into the Auxiliary Processor. An external function occupies only a negligible amount of workspace. (See *User Guide*.)

Migration Level

⎕ML determines the degree of migration of the Dyalog APL language towards IBM's APL2. Unless otherwise stated, the manual assumes ⎕ML has a value of 0.

Key to Notation

The following definitions and conventions apply throughout this manual:

`f`	A function, or an operator's left argument when a function.
`g`	A function, or an operator's right argument when a function.
`A`	An operator's left argument when an array.
`B`	An operator's right argument when an array.
`X`	The left argument of a function.
`Y`	The right argument of a function.
`R`	The explicit result of a function.
`[K]`	Axis specification.
`[I]`	Index specification.
`{X}`	The left argument of a function is optional.
`{R}←`	The function may or may not return a result, or the result may be suppressed.

function may refer to a primitive function, a system function, a defined (canonical, dynamic or assigned) function or a derived (from an operator) function.

Chapter 2:

Defined Functions & Operators

A defined function is a program that takes 0, 1, or 2 arrays as **arguments** and may produce an array as a result. A defined operator is a program that takes 1 or 2 functions or arrays (known as **operands**) and produces a **derived function** as a result. To simplify the text, the term **operation** is used within this chapter to mean function or operator.

Canonical Representation

Operations may be defined with the system function `⎕FX` (Fix) or by using the editor within definition mode. Applying `⎕CR` to the character array representing the name of an already established operation will produce its canonical representation. A defined operation is composed of lines. The first line (line 0) is called the operation HEADER. Remaining lines are APL statements, called the BODY.

The operation header consists of the following parts:

1. its model syntactical form,
2. an optional list of local names, each preceded by a semi-colon (`;`) character,
3. an optional comment, preceded by the symbol `⍝`.

Only the model is required. If local names and comments are included, they must appear in the prescribed order.

Model Syntax

The model for the defined operation identifies the name of the operation, its valence, and whether or not an explicit result may be returned. Valence is the number of explicit arguments or operands, either 0, 1 or 2; whence the operation is termed NILADIC, MONADIC or DYADIC respectively. Only a defined function may be niladic. There is no relationship between the valence of a defined operator, and the valence of the derived function which it produces. Defined functions and derived functions produced by defined operators may be ambivalent, i.e. may be executed monadically with one argument, or dyadically with two. An ambivalent operation is identified in its model by enclosing the left argument in braces.

The value of a result-returning function or derived function may be suppressed in execution if not explicitly used or assigned by enclosing the result in its model within braces. Such a suppressed result is termed SHY.

The tables below show all possible models for defined functions and operators respectively.

Defined Functions

Result	**Niladic**	**Monadic**	**Dyadic**	**Ambivalent**
None	`f`	`f Y`	`X f Y`	`{X} f Y`
Explicit	`R←f`	`R←f Y`	`R←X f Y`	`R←{X} f Y`
Suppressed	`{R}←f`	`{R}←f Y`	`{R}←X f Y`	`{R}←{X} f Y`

Note:The right argument `Y` and/or the result `R` may be represented by a single name, or as a blank-delimited list of names surrounded by parentheses. For further details, see "Namelists" on page 68.

Derived Functions produced by Monadic Operator

Result	Monadic	Dyadic	Ambivalent
None	`(A op)Y`	`X(A op)Y`	`{X}(A op)Y`
Explicit	`R←(A op)Y`	`R←X(A op)Y`	`R←{X}(A op)Y`
Suppressed	`{R}←(A op)Y`	`{R}←X(A op)Y`	`{R}←{X}(A op)Y`

Derived Functions produced by Dyadic Operator

Result	Monadic	Dyadic	Ambivalent
None	`(A op B)Y`	`X(A op B)Y`	`{X}(A op B)Y`
Explicit	`R←(A op B)Y`	`R←X(A op B)Y`	`R←{X}(A op B)Y`
Suppressed	`{R}←(A op B)Y`	`{R}←X(A op B)Y`	`{R}←{X}(A op B)Y`

Statements

A statement is a line of characters understood by APL. It may be composed of:

1. a LABEL (which must be followed by a colon `:`), or a CONTROL STATEMENT (which is preceded by a colon), or both,
2. an EXPRESSION (see "Expressions" on page 17),
3. a SEPARATOR (consisting of the diamond character `⋄` which must separate adjacent expressions),
4. a COMMENT (which must start with the character `⍝`).

Each of the four parts is optional, but if present they must occur in the given order except that successive expressions must be separated by `⋄`. Any characters occurring to the right of the first comment symbol (`⍝`) that is not within quotes is a comment.

Comments are not executed by APL. Expressions in a line separated by `⋄` are taken in left-to-right order as they occur in the line. For output display purposes, each separated expression is treated as a separate statement.

Examples

```
      5×10
50

      MULT: 5×10
50

      MULT: 5×10 ⋄ 2×4
50
8

      MULT: 5×10 ⋄ 2×4  ⍝ MULTIPLICATION
50
8
```

Global & Local Names

The following names, if present, are local to the defined operation:

1. the result,
2. the argument(s) and operand(s),
3. additional names in the header line following the model, each name preceded by a semi-colon character,
4. labels,
5. the argument list of the system function `⎕SHADOW` when executed,
6. a name assigned within a Dynamic Function.

All names in a defined operation must be valid APL names. The same name may be repeated in the header line, including the operation name (whence the name is localised). Normally, the operation name is not a local name.

The same name may not be given to both arguments or operands of a dyadic operation. The name of a label may be the same as a name in the header line. More than one label may have the same name. When the operation is executed, local names in the header line after the model are initially undefined; labels are assigned the values of line numbers on which they occur, taken in order from the last line to the first; the result (if any) is initially undefined.

In the case of a defined function, the left argument (if any) takes the value of the array to the left of the function when called; and the right argument (if any) takes the value of the array to the right of the function when called. In the case of a defined operator, the left operand takes the value of the function or array to the left of the operator when called; and the right operand (if any) takes the value of the function or array to the right of the operator when called.

During execution, a local name temporarily excludes from use an object of the same name with an active definition. This is known as LOCALISATION or SHADOWING. A value or meaning given to a local name will persist only for the duration of execution of the defined operation (including any time whilst the operation is halted). A name which is not local to the operation is said to be GLOBAL. A global name could itself be local to a pendent operation. A global name can be made local to a defined operation during execution by use of the system function `⎕SHADOW`. An object is said to be VISIBLE if there is a definition associated with its name in the active environment.

Examples

```
      A←1

     ∇ F
[1]    A←10
[2]  ∇

      F ⍝ <A> NOT LOCALISED IN <F>, GLOBAL VALUE REPLACED
      A
10
      A←1
      )ERASE F

     ∇ F;A
[1]    A←10
[2]  ∇

      F  ⍝ <A> LOCALISED IN <F>, GLOBAL VALUE RETAINED
      A
1
```

Any statement line in the body of a defined operation may begin with a LABEL. A label is followed by a colon (`:`). A label is a constant whose value is the number of the line in the operation defined by system function `⎕FX` or on closing definition mode.

The value of a label is available on entering an operation when executed, and it may be used but not altered in any expression.

Example

```
      ⎕VR'PLUS'
     ∇ R←{A} PLUS B
[1]    →DYADIC ⍴⍨2=⎕NC'A' ⋄ R←B ⋄ →END
[2]   DYADIC: R←A+B
[3]   END:
     ∇

      1 ⎕STOP'PLUS'

      2 PLUS 2

PLUS[1]
      DYADIC
2

      END
3
```

Namelists

The right argument and the result of a function may be specified in the function header by a single name or by a *Namelist*. In this context, a Namelist is a blank-delimited list of names surrounded by a single set of parentheses.

Names specified in a Namelist are automatically local to the function; there is no need to localise them explicitly using semi-colons.

If the *right argument* of a function is declared as a Namelist, the function will only accept a right argument that is a vector whose length is the same as the number of names in the Namelist. Calling the function with any other argument will result in a `LENGTH ERROR` in the calling statement. Otherwise, the elements of the argument are assigned to the names in the Namelist in the specified order.

Example:

```
     ∇ IDN←Date2IDN(Year Month Day)
[1]    'Year is ',⍕Year
[2]    'Month is ',⍕Month
[3]    'Day is ',⍕Day
[4] ...
     ∇

      Date2IDN 2004 4 30
Year is 2004
Month is 4
Day is 30

      Date2IDN 2004 4
LENGTH ERROR
      Date2IDN 2004 4
     ^
```

Note that if you specify a *single* name in the Namelist, the function may be called only with a 1-element vector or a scalar right argument.

If the *result* of a function is declared as a Namelist, the values of the names will automatically be stranded together in the specified order and returned as the result of the function when the function terminates.

Example:

```
     ∇ (Year Month Day)←Birthday age
[1]    Year←1949+age
[2]    Month←4
[3]    Day←30
     ∇
      Birthday 50
1999 4 30
```

Function Declaration Statements

Function Declaration statements are used to identify the characteristics of a function in some way.

The following declarative statements are provided.

- :Access
- :Attribute
- :Implements
- :Signature

With one exception, these statements are not executable statements and may theoretically appear anywhere in the body of the function. However, it is recommended that you place them at the beginning before any executable statements. The exception is:

```
:Implements Constructor <[:Base expr]>
```

In addition to being declarative (declaring the function to be a Constructor) this statement also executes the Constructor in the Base Class whether or not it includes :Base expr. Its position in the code is therefore significant.

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 "Methods" on page 164). 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" on page 167.

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

The `:Implements` statement identifies the function to be one of the following types.

```
:Implements Constructor <[:Base expr]>
:Implements Destructor
:Implements Method <InterfaceName.MethodName>
:Implements Trigger <name1><,name2,name3,...>
```

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 function 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 variable name1, name2, etc.

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 specifies 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 specifies 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 statements 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[]
Vec3
```

Control Structures

Control structures provide a means to control the flow of execution in your APL programs.

Traditionally, lines of APL code are executed one by one from top to bottom and the only way to alter the flow of execution is using the branch arrow. So how do you handle logical operations of the form “If this, do that; otherwise do the other”?

In APL this is often not a problem because many logical operations are easily performed using the standard array handling facilities that are absent in other languages. For example, the expression:

```
      STATUS←(1+AGE<16)⊃'Adult' 'Minor'
```

sets `STATUS` to `'Adult'` if `AGE` is 16 or more; otherwise sets `STATUS` to `'Minor'`.

Things become trickier if, depending upon some condition, you wish to execute one set of code instead of another, especially when the code fragments cannot conveniently be packaged as functions. Nevertheless, careful use of array logic, defined operators, the execute primitive function and the branch arrow can produce high quality maintainable and comprehensible APL systems.

Control structures provide an additional mechanism for handling logical operations and decisions. Apart from providing greater affinity with more traditional languages, Control structures may enhance comprehension and reduce programming errors, especially when the logic is complex. Control structures are not, however, a replacement for the standard logical array operations that are so much a part of the APL language.

Control Structures are blocks of code in which the execution of APL statements follows certain rules and conditions. Control structures are implemented using a set of *control words* that all start with the colon symbol (:). Control Words are case-insensitive.

There are eight different types of control structures defined by the control words, `:If`, `:While`, `:Repeat`, `:For`, `:Select`, `:With`, `:Trap` and `:Hold`. Each one of these control words may occur only at the beginning of an APL statement and indicates the start of a particular type of control structure.

Within a control structure, certain other control words are used as qualifiers. These are `:Else`, `:ElseIf`, `:AndIf`, `:OrIf`, `:Until`, `:Case` and `:CaseList`.

A third set of control words is used to identify the end of a particular control structure. These are `:EndIf`, `:EndWhile`, `:EndRepeat`, `:EndFor`, `:EndSelect`, `:EndWith`, `:EndTrap` and `:EndHold`. Although formally distinct, these control words may all be abbreviated to `:End`.

Finally, the `:GoTo`, `:Return`, `:Leave` and `:Continue` control words may be used to conditionally alter the flow of execution within a control structure.

Control words, including qualifiers such as `:Else` and :`ElseIf`, may occur only at the beginning of a line or expression in a diamond-separated statement. The only exceptions are `:In` and `:InEach` which must appear on the same line within a `:For` expression.

Key to Notation

The following notation is used to describe Control Structures within this section:

`aexp`	an expression returning an array,
`bexp`	an expression returning a single Boolean value (0 or 1),
`var`	loop variable used by `:For` control structure,
`code`	0 or more lines of APL code, including other (nested) control structures,
`andor`	*either* one or more `:AndIf` statements, *or* one or more `:OrIf` statements.

```
|
.------------------------.
|                        |
|<---------------.       |<---------------.
|                |       |                |
code             |       code             |
|                |       |                |
|                |       |                |
:AndIf bexp-----'        :OrIf bexp------'
|                        |
|<-----------------------'
|
```

Access Statement `:Access`

The `:Access` statement may be used to define the characteristics of a Class, the characteristics of a defined function (Method) in a Class, or the characteristics of other Class members.

:Access Statement in a Function/Method.

:Access Statement in a Class or in other members of a Class.

Attribute Statement `:Attribute`

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

:Attribute Statement for a Class.

:Attribute statement for a Method.

If Statement :If bexp

The simplest `:If` control structure is a single condition of the form:

```
[1]   :If AGE<21
[2]       expr 1
[3]       expr 2
[5]   :EndIf
```

If the test condition (in this case `AGE<21`) is true, the statements between the `:If` and the `:EndIf` will be executed. If the condition is false, none of these statements will be run and execution resumes after the `:EndIf`. Note that the test condition to the right of `:If` must return a single element Boolean value 1 (true) or 0 (false).

`:If` control structures may be considerably more complex. For example, the following code will execute the statements on lines `[2-3]` if `AGE<21` is 1 (true), **or alternatively**, the statement on line `[6]` if `AGE<21` is 0 (false).

```
[1]   :If AGE<21
[2]       expr 1
[3]       expr 2
[5]   :Else
[6]       expr 3
[7]   :EndIf
```

Instead of a single condition, it is possible to have multiple conditions using the `:ElseIf` control word. For example:

```
[1]   :If WINEAGE<5
[2]       'Too young to drink'
[5]   :ElseIf WINEAGE<10
[6]       'Just Right'
[7]   :ElseIf WINEAGE<15
[8]       'A bit past its prime'
[9]   :Else
[10]     'Definitely over the hill'
[11]  :EndIf
```

Notice that APL executes the expression(s) associated with the **first** condition that is true or those following the `:Else` if **none** of the conditions are true.

The `:AndIf` and `:OrIf` control words may be used to define a block of conditions and so refine the logic still further. You may qualify an `:If` or an `:ElseIf` with one or more `:AndIf` statements **or** with one or more `:OrIf` statements. You may not however mix `:AndIf` and `:OrIf` in the same conditional block. For example:

```
[1]   :If WINE.NAME≡'Chateau Lafitte'
[2]   :AndIf WINE.YEAR∊1962 1967 1970
[3]       'The greatest?'
[4]   :ElseIf WINE.NAME≡'Chateau Latour'
[5]   :Orif WINE.NAME≡'Chateau Margaux'
[6]   :Orif WINE.PRICE>100
[7]       'Almost as good'
[8]   :Else

[9]       'Everyday stuff'
[10]  :EndIf
```

Please note that in a `:If` control structure, the conditions associated with each of the condition blocks are executed in order until an entire condition block evaluates to true. At that point, the APL statements following this condition block are executed. None of the conditions associated with any other condition block are executed. Furthermore, if an `:AndIf` condition yields 0 (false), it means that the entire block must evaluate to false so the system moves immediately on to the next block without executing the other conditions following the failing `:AndIf`. Likewise, if an `:OrIf` condition yields 1 (true), the entire block is at that point deemed to yield true and none of the following `:OrIf` conditions in the same block are executed.

:If Statement

```
|
:If bexp
|
.---------.
|         |
|         andor
|         |
|<--------'
|
code
|
|<--------------------------------------.
|                                       |
.---------.---------.                   |
|         |         |                   |
|         :Else     :ElseIf bexp        |
|         |         |                   |
|         |         .---------.         |
|         |         |         |         |
|         |         |         andor     |
|         |         |         |         |
|         |         |<--------'         |
|         |         |                   |
|         code      code                |
|         |         |                   |
|<--------'         `-------------------'
|
:End[If]
|
```

While Statement :While bexp

The simplest `:While` loop is :

```
[1]   I←0
[2]   :While I<100
[3]       expr1
[4]       expr2
[5]       I←I+1
[6]   :EndWhile
```

Unless `expr1` or `expr2` alter the value of `I`, the above code will execute lines `[3-4]` 100 times. This loop has a single condition; the value of `I`. The purpose of the `:EndWhile` statement is solely to mark the end of the iteration. It acts the same as if it were a branch statement, branching back to the `:While` line.

An alternative way to terminate a `:While` structure is to use a `:Until` statement. This allows you to add a second condition. The following example reads a native file sequentially as 80-byte records until it finds one starting with the string `'Widget'` or reaches the end of the file.

```
[1]   I←0
[2]   :While I<⎕NSIZE ¯1
[3]       REC←⎕NREAD ¯1 82 80
[4]       I←I+⍴REC
[5]   :Until 'Widget'≡6⍴REC
```

Instead of single conditions, the tests at the beginning and end of the loop may be defined by more complex ones using `:AndIf` and `:OrIf`. For example:

```
[1]   :While 100>i
[2]   :AndIf 100>j
[3]       i j←foo i j
[4]   :Until 100<i+j
[5]   :OrIf i<0
[6]   :OrIf j<0
```

In this example, there are complex conditions at both the start and the end of the iteration. Each time around the loop, the system tests that both `i` and `j` are less than or equal to 100. If either test fails, the iteration stops. Then, after `i` and `j` have been recalculated by `foo`, the iteration stops if `i+j` is equal to or greater than 100, or if either `i` or `j` is negative.

:While Statement

```
|
:While bexp
|
.-------.
|       |
|       andor
|       |
|<------'
|
code
|
.---------------.
|               |
:End[While]     :Until bexp
|               |
|               .-------.
|               |       |
|               |       andor
|               |       |
|               |<------'
|               |
|<--------------'
|
```

Repeat Statement :Repeat

The simplest type of `:Repeat` loop is as follows. This example executes lines `[3-5]` 100 times. Notice that as there is no conditional test at the beginning of a `:Repeat` structure, its code statements are executed at least once.

```
[1]   I←0
[2]   :Repeat
[3]       expr1
[4]       expr2
[5]       I←I+1
[6]   :Until I=100
```

You can have multiple conditional tests at the end of the loop by adding `:AndIf` or `:OrIf` expressions. The following example will read data from a native file as 80-character records until it reaches one beginning with the text string `'Widget'` or reaches the end of the file.

```
[1]   :Repeat
[2]       REC←⎕NREAD ¯1 82 80
[3]   :Until 'Widget'≡6⍴REC
[4]   :OrIf 0=⍴REC
```

A `:Repeat` structure may be terminated by an `:EndRepeat` (or `:End`) statement in place of a conditional expression. If so, your code must explicitly jump out of the loop using a `:Leave` statement or by branching. For example:

```
[1]   :Repeat
[2]       REC←⎕NREAD ¯1 82 80
[3]      :If 0=⍴REC
[4]      :OrIf 'Widget'≡6⍴REC
[5]         :Leave
[6]      :EndIf
[7]   :EndRepeat
```

:Repeat Statement

```
|
:Repeat
|
code
|
.----------------.
|                |
:End[Repeat]     :Until bexp
|                |
|                .--------.
|                |        |
|                |        andor
|                |        |
|                |<-------'
|                |
|<---------------'
|
```

For Statement `:For var :In[Each] aexp`

Single Control Variable

The `:For` loop is used to execute a block of code for a series of values of a particular control variable. For example, the following would execute lines `[2-3]` successively for values of `I` from 3 to 5 inclusive:

```
[1]   :For I :In 3 4 5
[2]       expr1 I
[3]       expr2 I
[4]   :EndFor
```

The way a `:For` loop operates is as follows. On encountering the `:For`, the expression to the right of `:In` is evaluated and the result stored. This is the *control array*. The *control variable*, named to the right of the `:For`, is then assigned the first value in the control array, and the code between `:For` and `:EndFor` is executed. On encountering the `:EndFor`, the control variable is assigned the next value of the control array and execution of the code is performed again, starting at the first line after the `:For`. This process is repeated for each value in the control array.

Note that if the control array is empty, the code in the `:For` structure is not executed. Note too that the control array may be any rank and shape, but that its elements are assigned to the control variable in ravel order.

The control array may contain any type of data. For example, the following code resizes (and compacts) all your component files

```
[1]   :For FILE :In (↓⎕FLIB '')~¨' '
[2]       FILE ⎕FTIE 1
[3]       ⎕FRESIZE 1
[4]       ⎕FUNTIE 1
[5]   :EndFor
```

You may also nest `:For` loops. For example, the following expression finds the timestamp of the most recently updated component in all your component files.

```
[1]   TS←0
[2]   :For FILE :In (↓⎕FLIB '')~¨' '
[3]       FILE ⎕FTIE 1
[4]       START END←2⍴⎕FSIZE 1
[5]       :For COMP :In (START-1)↓⍳END-1
[6]           TS⌈←¯1↑⎕FREAD FILE COMP
[7]       :EndFor
[8]       ⎕FUNTIE 1
[9]   :EndFor
```

Multiple Control Variables

The `:For` control structure can also take multiple variables. This has the effect of doing a strand assignment each time around the loop.

For example `:For a b c :in (1 2 3)(4 5 6)`, sets `a b c←1 2 3`, first time around the loop and `a b c←4 5 6`, the second time.

Another example is `:For i j :In ⍳⍴Matrix`, which sets `i` and `j` to each row and column index of `Matrix`.

:InEach Control Word

```
      :For var ... :InEach value ...
```

In a `:For` control structure, the keyword `:InEach` is an alternative to `:In`.

For a single control variable, the effect of the keywords is identical but for multiple control variables the values vector is inverted.

The distinction is best illustrated by the following equivalent examples:

```
      :For a b c :In (1 2 3)(3 4 5)(5 6 7)(7 8 9)
          ⎕←a b c
      :EndFor

      :For a b c :InEach (1 3 5 7)(2 4 6 8)(3 5 7 9)
           ⎕←a b c
      :EndFor
```

In each case, the output from the loop is:

```
1 2 3
3 4 5
5 6 7
7 8 9
```

Notice that in the second case, the number of items in the values vector is the same as the number of control variables. A more typical example might be.

```
      :For a b c :InEach avec bvec cvec
          ...
      :EndFor
```

Here, each time around the loop, control variable `a` is set to the next item of `avec`, `b` to the next item of `bvec` and `c` to the next item of `cvec`.

:For Statement

```
|
:For var :In[Each] aexp
|
code
|
:End[For]
|
```

Select Statement :Select aexp

A `:Select` structure is used to execute alternative blocks of code depending upon the value of an array. For example, the following displays `'I is 1'` if the variable `I` has the value 1, `'I is 2'` if it is 2, or `'I is neither 1 nor 2'` if it has some other value.

```
[1]   :Select I
[2]   :Case 1
          'I is 1'
[4]   :Case 2
[5]       'I is 2'
[6]   :Else
[7]       'I is neither 1 nor 2'
[8]   :EndSelect
```

In this case, the system compares the value of the array expression to the right of the `:Select` statement with each of the expressions to the right of the `:Case` statements and executes the block of code following the one that matches. If none match, it executes the code following the `:Else` (which is optional). Note that comparisons are performed using the `≡` primitive function, so the arrays must match exactly. Note also that not all of the `:Case` expressions are necessarily evaluated because the process stops as soon as a matching expression is found.

Instead of a `:Case` statement, you may also use a `:CaseList` statement. If so, the *enclose of* the array expression to the right of `:Select` is tested for membership of the array expression to the right of the `:CaseList` using the `∊` primitive function.

Example

```
[1]   :Select ?6 6
[2]   :Case 6 6
[3]       'Box Cars'
[4]   :Case 1 1
[5]       'Snake Eyes'
[6]   :CaseList 2ρ¨ι6
[7]       'Pair'
[8]   :CaseList (ι6),¨⌽ι6
[9]       'Seven'
[10]  :Else
[11]      'Unlucky'
[12]  :EndSelect
```

:Select Statement

```
|
:Select aexp
|
|<----------------------------------------------.
|                                               |
.-------.-------.---------------.               |
|       |       |               |               |
|       :Else   :Case aexp      :CaseList aexp  |
|       |       |               |               |
|       |       |<--------------'               |
|       |       |                               |
|       code    code                            |
|       |       |                               |
|<------'       `-------------------------------'
|
:End[Select]
```

With Statement :With obj

`:With` is a control structure that may be used to simplify a series of references to an object or namespace. `:With` changes into the specified namespace for the duration of the control structure, and is terminated by `:End[With]`. For example, you could update several properties of a Grid object `F.G` as follows:

```
:With F.G
     Values←4 3⍴0
     RowTitles←'North' 'South' 'East' 'West'
     ColTitles←'Cakes' 'Buns' 'Biscuits'
:EndWith
```

`:With` is analogous to `⎕CS` in the following senses:

- The namespace argument to `:With` is interpreted relative to the current space.
- With the exception of those with name class 9, local names in the containing defined function continue to be visible in the new space.
- Global references from within the `:With` control structure are to names in the new space.
- Exiting the defined function from within a `:With` control structure causes the space to revert to the one from which the function was called.

On leaving the `:With` control structure, execution reverts to the original namespace. Notice however that the interpreter does not detect branches `(→)` out of the control structure. `:With` control structures can be nested in the normal fashion:

```
[1]   :With 'x'              ⍝ Change to #.x
[2]       :With 'y'          ⍝ Change to #.x.y
[3]           :With ⎕SE      ⍝ Change to ⎕SE
[4]               ...        ⍝ ... in ⎕SE
[5]           :EndWith       ⍝ Back to #.x.y
[6]       :EndWith           ⍝ Back to #.x
[7]   :EndWith               ⍝ Back to #
```

:With Statement

```
|
:With namespace (ref or name)
|
code
|
:End[With]
|
```

Hold Statement :Hold tkn

Whenever more than one thread tries to access the same piece of data or shared resource at the same time, you need some type of synchronisation to control access to that data. This is provided by `:Hold`.

`:Hold` provides a mechanism to control thread entry into a critical section of code. `tkns` must be a simple character vector or scalar, or a vector of character vectors. `tkns` represents a set of 'tokens', all of which must be acquired before the thread can continue into the control structure. `:Hold` is analogous to the component file system `⎕FHOLD`.

Within the whole active workspace, a token with a particular value may be held only once. If the hold succeeds, the current thread *acquires* the tokens and execution continues with the first phrase in the control structure. On exit from the structure, the tokens are released for use by other threads. If the hold fails, because one or more of the tokens is already in use:

1. If there is no `:Else` clause in the control structure, execution of the thread is blocked until the requested tokens become available.
2. Otherwise, acquisition of the tokens is abandoned and execution resumed immediately at the first phrase in the `:Else` clause.

`tkns` can be either a single token:

```
    'a'
    'Red'
    '#.Util'
    ''
    'Program Files'
```

… or a number of tokens:

```
    'red' 'green' 'blue'
    'doe' 'a' 'deer'
    ,¨'abc'
    ↓⎕nl 9
```

Pre-processing removes trailing blanks from each token before comparison, so that, for example, the following two statements are equivalent:

```
    :Hold 'Red' 'Green'
    :Hold ↓2 5⍴'Red  Green'
```

Unlike `⎕FHOLD`, a thread does not release all existing tokens before attempting to acquire new ones. This enables the nesting of holds, which can be useful when multiple threads are concurrently updating parts of a complex data structure.

In the following example, a thread updates a critical structure in a child namespace, and then updates a structure in its parent space. The holds will allow all 'sibling' namespaces to update concurrently, but will constrain updates to the parent structure to be executed one at a time.

```
:Hold ⎕cs''            ⍝ Hold child space
    ...                ⍝ Update child space
    :Hold ##.⎕cs''     ⍝ Hold parent space
        ...            ⍝ Update Parent space
    :EndHold
    ...
:EndHold
```

However, with the nesting of holds comes the possibility of a 'deadlock'. For example, consider the two threads:

Thread 1	**Thread 2**

`:Hold 'red'`
`    ...`
`    :Hold 'green'`
`        ...`
`    :EndHold`
`:EndHold`
	`:Hold 'green'`
	`    ...`
	`    :Hold 'red'`
	`        ...`
	`    :EndHold`
	`:EndHold`

In this case if both threads succeed in acquiring their first hold, they will both block waiting for the other to release its token.

If this deadlock situation is detected acquisition of the tokens is abandoned. Then:

1. If there is an `:Else` clause in the control structure, execution jumps to the `:Else` clause.
2. Otherwise, APL issues an error `(1008) DEADLOCK`.

You can avoid deadlock by ensuring that threads always attempt to acquire tokens in the same chronological order, and that threads never attempt to acquire tokens that they already own.

Note that token acquisition for any particular `:Hold` is atomic, that is, either *all* of the tokens or *none* of them are acquired. The following example *cannot* deadlock:

Thread 1	Thread 2

`:Hold 'red'`
`    ...`
`    :Hold 'green'`
`        ...`
`    :EndHold`
`:EndHold`
	`:Hold 'green' 'red'`
	`    ...`
	`    :EndHold`

Examples

`:Hold` could be used for example, during the update of a complex data structure that might take several lines of code. In this case, an appropriate value for the token would be the name of the data structure variable itself, although this is just a programming convention: the interpreter does not associate the token value with the data variable.

```
:Hold'Struct'
    ...                 ⍝ Update Struct
    Struct ← ...
:EndHold
```

The next example guarantees exclusive use of the current namespace:

```
:Hold ⎕CS''             ⍝ Hold current space
    ...
:EndHold
```

The following example shows code that holds two positions in a vector while the contents are exchanged.

```
:Hold ⍕¨to fm
    :If >/vec[fm to]
        vec[fm to]←vec[to fm]
    :End
:End
```

Between obtaining the next available file tie number and using it:

```
:Hold '⎕FNUMS'
    tie←1+⌈/0,⎕FNUMS
    fname ⎕FSTIE tie
:End
```

The above hold is not necessary if the code is combined into a single line:

```
fname ⎕FSTIE tie←1+⌈/0,⎕FNUMS
```

or,

```
tie←fname ⎕FSTIE 0
```

Note that `:Hold`, like its component file system counterpart `⎕FHOLD`, is a device to enable *co-operating* threads to synchronise their operation.

`:Hold` does not *prevent* threads from updating the same data structures concurrently, it prevents threads only from `:Hold`-ing the same tokens.

:Hold Statement

```
|
:Hold token(s)
|
code
|
|-------.
|       |
|       :Else
|       |
|       code
|       |
|<------.
|
:End[Hold]
|
```

Trap Statement :Trap ecode

`:Trap` is an error trapping mechanism that can be used in conjunction with, or as an alternative to, the `⎕TRAP` system variable. It is equivalent to APL2's `⎕EA`, except that the code to be executed is not restricted to a single expression and is not contained within quotes (and so is slightly more efficient).

Operation

The segment of code immediately following the `:Trap` keyword is executed. On completion of this segment, if no error occurs, control passes to the code following `:End[Trap]`.

If an error does occur, the event code (error number) is noted and:

- If the error occurred within a sub-function, the system cuts the execution stack back to the function containing the `:Trap` keyword. In this respect, `:Trap` behaves like `⎕TRAP` with a `'C'` qualifier.
- The system searches for a `:Case[List]` representing the event code.
- If there is such a `:Case[List]`, or failing that, an `:Else` keyword, execution continues from this point.

Otherwise, control passes to the code following `:End[Trap]` and no error processing occurs.

Note that the error trapping is in effect **only** during execution of the initial code segment. It is disabled (or surrendered to outer level `:Trap`s or `⎕TRAP`s) immediately a trapped error occurs. In particular, the error trap is no longer in effect during processing of `:Case[List]`'s argument or in the code following the `:Case[List]` or `:Else` statement. This avoids the situation sometimes encountered with `⎕TRAP` where an infinite 'trap loop' occurs. If an error which is not specified occurs, it is processed by outer `:Trap`s, `⎕TRAP`s, or default system processing in the normal fashion.

Note that the statement `:trap ⍬` results in no errors being trapped.

Examples

```
    ∇ lx
[1]   :Trap 1000        ⍝ Cutback and exit on interrupt
[2]       Main ...
[3]   :EndTrap
    ∇

    ∇ ftie←Fcreate file      ⍝ Create null component file
[1]   ftie←1+⌈/0,⎕fnums      ⍝ next tie number.
[2]   :Trap 22               ⍝ Trap FILE NAME ERROR
[3]       file ⎕fcreate ftie ⍝ Try to create file.
[4]   :Else
[5]       file ⎕ftie ftie    ⍝ Tie the file.
[6]       file ⎕ferase ftie  ⍝ Drop the file.
[7]       file ⎕fcreate ftie ⍝ Create new file.
[8]   :EndTrap
    ∇

    ∇ lx ⍝ Distinguish various cases
[1]   :Trap 0 1000
[2]       Main ...
[3]   :Case 1002
[4]       'Interrupted ...'
[5]   :CaseList 1 10 72 76
[6]       'Not enough resources'
[7]   :CaseList 17+⍳20
[8]       'File System Problem'
[9]   :Else
[10]      'Unexpected Error'
[11]  :EndTrap
    ∇
```

Note that `:Trap`s can be nested:

```
    ∇ ntie←Ntie file                  ⍝ Tie native file
[1]   ntie←¯1+⌊/0,⎕nnums              ⍝ Next native tie num
[2]   :Trap 22                        ⍝ Trap FILE NAME
ERROR
[3]       file ⎕ntie ntie             ⍝ Try to tie file
[4]   :Else
[5]       :Trap 22                    ⍝ Trap FILE NAME
ERROR
[6]           (file,'.txt')⎕ntie ntie ⍝ Try with .txt
extn
[7]       :Else
[8]           file ⎕ncreate ntie      ⍝ Create null file.
[9]       :EndTrap
[10]  :EndTrap
    ∇
```

:Trap Statement

```
|
:Trap <ecode>
|
code
|
|<-------------------------------------.
|                                      |
.-------.-------.                      |
|       |       |                      |
|       :Else   :Case[List] <ecode>    |
|       |       |                      |
|       |       |                      |
|       |       |                      |
|       code    code                   |
|       |       |                      |
|<------'       `----------------------'
|
:End[Trap]
|
```

Where `ecode` is a scalar or vector of `⎕TRAP` event codes (see*Language Reference*).

Note that within the `:Trap` control structure, `:Case` is used for a single event code and `:CaseList` for a vector of event codes.

GoTo Statement :GoTo aexp

A `:GoTo` statement is a direct alternative to → (branch) and causes execution to jump to the line specified by the first element of `aexp`.

The following are equivalent. See *Language Reference* for further details.

```
→Exit
:GoTo Exit

→(N<I←I+1)/End
:GoTo (N<I←I+1)/End

→1+⎕LC
:GoTo 1+⎕LC

→10
:GoTo 10
```

Return Statement :Return

A `:Return` statement causes a function to terminate and has exactly the same effect as `→0`.

The `:Return` control word takes no argument.

A `:Return` statement may occur anywhere in a function or operator.

Leave Statement :Leave

A :Leave statement is used to explicitly terminate the execution of a block of statements within a `:For`, `:Repeat` or `:While` control structure.

The `:Leave` control word takes no argument.

Continue Statement :Continue

A `:Continue` statement starts the next iteration of the immediately surrounding `:For`, `:Repeat` or `:While` control loop.

When executed within a `:For` loop, the effect is to start the body of the loop with the next value of the iteration variable.

When executed within a `:Repeat` or `:While` loop, if there is a trailing test that test is executed and, if the result is true, the loop is terminated. Otherwise the leading test is executed in the normal fashion.

Section Statement :Section

Functions and scripted objects (classes, namespaces etc.) can be subdivided into Sections with `:Section` and `:EndSection` statements. Both statements may be followed by an optional and arbitrary name or description. The purpose is to split the function up into sections that you can open and close in the Editor, thereby aiding readability and code management. Sections have no effect on the execution of the code, but must follow the nesting rules of other control structures.

For further information, See User Guide.

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
[1]    :Implements Trigger A
[2]    ...

     ∇ TEST;A
[1]    ⎕FX ⎕OR'TRIG'
[2]    A←10
```

Example

The following function displays information when the value of variables A or B changes.

```
     ∇ TRIG arg
[1]    :Implements Trigger A,B
[2]    arg.Name'is now 'arg.NewValue
[3]    :Trap 6 ⍝ VALUE ERROR
[4]        arg.Name'was    'arg.OldValue
[5]    :Else
[6]        arg.Name' was     [undefined]'
[7]    :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
invoked
```

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
```

See "Trigger Fields" on page 163 for information on how a Field (in a Class) may be used as a Trigger.

Idiom Recognition

Idioms are commonly used expressions that are recognised and evaluated internally, providing a significant performance improvement.

For example, the idiom `BV/⍳⍴A` (where `BV` is a Boolean vector and `A` is an array) would (in earlier Versions of Dyalog APL) have been evaluated in 3 steps as follows:

1. Evaluate `⍴A` and store result in temporary variable `temp1` (`temp1` is just an arbitrary name for the purposes of this explanation)
2. Evaluate `⍳temp1` and store result in temporary variable `temp2`.
3. Evaluate `BV/temp2`
4. Discard temporary variables

In the current Version of Dyalog APL, the expression is recognised in its entirety and processed in a single step as if it were a single primitive function. In this case, the resultant improvement in performance is between 2 and 4.5.

Idiom recognition is precise; an expression that is almost identical but not exactly identical to an expression given in the Idiom List table will not be recognised.

For example, `⎕AV⍳` will be recognised as an idiom, but `(⎕AV)⍳` will not. Similarly, `(,)/` would not be recognized as the Join idiom.

Idiom List

In the following table, arguments to the idiom have types and ranks as follows:

Type	Description	Rank	Description
C	Character	S	Scalar or 1-item vector
B	Boolean	V	Vector
N	Numeric	M	Matrix
P	Nested (pointer)	A	Array (any rank)
A	Any type		

For example: NV: numeric vector, CM: character matrix, PV: nested vector.

Expression	Description
`⍴⍴A`	Rank
`BV/⍳NS`	Sequence selection
`BV/⍳⍴A`	Index selection
`NA⊃¨⊂A`	Array selection
`A{}A`	Sink
`A{⍺}A`	Left (Lev)
`A{⍵}A`	Right (Dex)
`A{⍺ ⍵}A`	Link
`{0}A`	Zero
`{0}¨A`	Zero Each
`,/PV`	Join
`⍪/PV`	Join along first axis
`⊃⌽A`	Upper right item (`⎕ML<2`)
`↑⌽A`	Upper right item ((`⎕ML<2`)
`⊃⌽,A`	Lower right item ((`⎕ML<2`)
`↑⌽,A`	Lower right item ((`⎕ML<2`)
`0=⍴V`	Zero shape
`0=⍴⍴A`	Zero rank
`0=≡A`	Zero depth
`⎕AV⍳CA`	Atomic vector index (Classic Edition only; use (`⎕UCS`)
`M{(↓⍺)⍳↓⍵}M`	Matrix Iota
`↓⍉↑PV`	Nested vector transpose ((`⎕ML<2`)
`↓⍉⊃PV`	Nested vector transpose ((`⎕ML<2`)
`∧\' '=CA`	Mask of leading blanks.
`+/∧\' '=CA`	Number of leading blanks
`+/∧\BA`	Number of leading ones

Expression	**Description**
`{(∨\' '≠⍵)/⍵}CV`	Trim leading blanks
`{(+/∧\' '=⍵)↓⍵}CV`	Trim leading blanks
`~∘' '¨↓CA`	No-blank split
`{(+/∨\' '≠⌽⍵)↑¨↓⍵}CA`	No-trailing-blank split
`⊃∘⍴¨A`	Length of first axis of each sub-array (`⎕ML<2`)
`↑∘⍴¨A`	Length of first axis of each sub-array (`⎕ML≥2`)
`A,←A`	Catenate To
`A⍪←A`	Catenate to (along first axis)
`{⍵[⍋⍵]}V`	Sort up vector
`{⍵[⍒⍵]}V`	Sort down vector
`{⍵[⍋⍵;]}M`	Sort up matrix
`{⍵[⍒⍵;]}M`	Sort down matrix
`1=≡A`	Is depth 1
`1=≡,A`	Is simple (depth 1 or 0)
`0∊⍴A`	Is null
`~0∊⍴A`	Is non-null
`⊣⌿A`	First sub-array along first axis
`⊣/A`	First sub-array along last axis
`⊢⌿A`	Last sub-array along first axis
`⊢/A`	Last sub-array along last axis
`*○N`	Euler's idiom
`0=⊃⍴`	Is first dimension empty (`⎕ML<2`)
`0≠⊃⍴`	Is first dimension not empty (`⎕ML<2`)

Notes

Sequence Selection `/⍳` and **Index Selection** `/⍳⍴`, as well as providing an execution time advantage, reduce intermediate workspace usage and consequently, the incidence of memory compactions and the likelihood of a `WS FULL`.

Array Selection `NV⊃¨⊂A`, is implemented as `A[NV]`, which is significantly faster. The two are equivalent but the former may now be used as a matter of taste with no performance penalty.

Join `,/` is currently special-cased only for vectors of vectors or scalars. Otherwise, the expression is evaluated as a series of concatenations. Recognition of this idiom turns **join** from an *n-squared* algorithm into a linear one. In other words, the improvement factor is proportional to the size of the argument vector.

Upper and **Lower Right Item** now take constant time. Without idiom recognition, the time taken depends linearly on the number of items in the argument.

Zero Depth `0=≡` takes a small constant time. Without idiom recognition, time taken would depend on the size and depth of the argument, which in the case of a deeply nested array, could be significant.

Nested vector transpose `↓⍉↑` is special-cased only for a vector of nested vectors, each of whose items is of the same length.

Matrix Iota `{(↓⍺)⍳↓⍵}`. As well as being quicker, the Matrix Iota idiom can accommodate much larger matrices. It is particularly effective when bound with a left argument using the compose operator:

```
      find←mat∘{(↓⍺)⍳↓⍵}      ⍝ find rows in mat table.
```

In this case, the internal hash table for `mat` is retained so that it does not need to be generated each time the monadic derived function `find` is applied to a matrix argument.

Trim leading blanks `{(∨\' '≠⍵)/⍵}` and `{(+/∧\' '=⍵)↓⍵}` are two codings of the same idiom. Both use the same C code for evaluation.

No-blank split `~∘' '¨↓` typically takes a character matrix argument and returns a vector of character vectors from which, all blanks have been removed. An example might be the character matrix of names returned by the system function `⎕NL`. In general, this idiom accommodates character arrays of any rank.

No-trailing-blank split `{(+/∨\' '≠⌽⍵)↑¨↓⍵}` typically takes a character matrix argument and returns a vector of character vectors. Any embedded blanks in each row are preserved but trailing blanks are removed. In general, this idiom accommodates character arrays of any rank.

Lengths `⊃∘⍴¨A` (`⎕ml<2`) or `↑∘⍴¨A` (`⎕ml>2`) avoids having to create an intermediate nested array of shape vectors.

For an array of vectors, this idiom quickly returns a *simple array* of the length of each vector.

```
      ⊃∘⍴¨ 'Hi' 'Pete' ⍝ Vector Lengths
2 4
```

For an array of matrices, it returns a simple array of the number of rows in each matrix.

```
      ⊃∘⍴¨⎕CR¨↓⎕NL 3   ⍝ Lines in functions
5 21...
```

Catenate To `A,←A` and `A⍪←A`optimise the catenation of an array to another array along the last and first dimension respectively.

Among other examples, this idiom optimises *repeated* catenation of a scalar or vector to an existing vector.

```
      props,←⊂ 'Posn' 0 0
      props,←⊂'Size' 50 50
      vector,←2+4
```

Note that the idiom is not applied if the value of vector `V` is shared with another symbol in the workspace, as illustrated in the following examples:

In this first example, the idiom *is* used to perform the catenation to `V1`.

```
      V1←⍳10
      V1,←11
```

In the second example, the idiom *is not* used to perform the catenation to `V1`, because its value is at that point shared with `V2`.

```
      V1←⍳10
      V2←V1
      V1,←11
```

In the third example, the idiom *is not* used to perform the catenation to `V` in `Join[1]` because its value is at that point shared with the array used to call the function.

```
     ∇ V←V Join A
[1]    V,←A
     ∇
      (⍳10) Join 11
1 2 3 4 5 6 7 8 9 10 11
```

Sub-array selection idioms `⊢⌿A`, `⊢/A`, `⊣⌿A`, and `⊣/A` return the first (respectively last) rank `(0⌈¯1+⍴⍴A)` sub-array along the first (respectively last) axis of `A`. For example, if `V` is a vector, then:

`⊣/V`	First item of vector
`⊢/V`	Last item of vector

Similarly, if `M` is a matrix, then:

`⊣⌿M`	First row of matrix
`⊣/M`	First column of matrix
`⊢⌿M`	Last row of matrix
`⊢/M`	Last column of matrix

The idiom generalises uniformly to higher-rank arrays.

Euler's idiom `*○N` produces accurate results for right argument values that are a multiple of `0J0.5`. This is so that Euler's famous identity `0=1+*○0J1` holds, even though the machine cannot represent multiples of pi, including `○0J1`, accurately.

Search Functions and Hash Tables

Primitive dyadic *search* functions, such as `⍳` (index of) and `∊` (membership) have a *principal* argument in which items of the other *subject* argument are located.

In the case of `⍳`, the principal argument is the one on the left and in the case of `∊`, it is the one on the right. The following table shows the principal (P) and subject (s) arguments for each of the functions.

`P ⍳ s`	Index of
`s ∊ P`	Membership
`s ∩ P`	Intersection
`P ∪ s`	Union
`s ~ P`	Without
`P {(↓⍺)⍳↓⍵} s`	Matrix Iota (idiom)
`P∘⍋ and P∘⍒`	Sort

The Dyalog APL implementation of these functions already uses a technique known as *hashing* to improve performance over a simple linear search. (Note that `⍷` (find) does not employ the same hashing technique, and is excluded from this discussion.)

Building a *hash table* for the principal argument takes a significant time but is rewarded by a considerably quicker search for each item in the subject. Unfortunately, the hash table is discarded each time the function completes and must be reconstructed for a subsequent call (even if its principal argument is identical to that in the previous one).

For optimal performance of *repeated* search operations, the hash table may be retained between calls, by binding the function with its principal argument using the primitive `∘` (compose) operator. The retained hash table is then used directly whenever this monadic derived function is applied to a subject argument.

Notice that retaining the hash table pays off only on a second or subsequent application of the derived function. This usually occurs in one of two ways: either the derived function is named for later (and repeated) use, as in the first example below or it is applied repeatedly as the operand of a primitive or defined operator, as in the second example.

Example: naming a derived function.

```
      words←'red' 'ylo' 'grn' 'brn' 'blu' 'pnk' 'blk'

      find←words∘⍳                  ⍝ monadic find
function
      find'blk' 'blu' 'grn' 'ylo'  ⍝
7 5 3 2
      find'grn' 'brn' 'ylo' 'red'  ⍝ fast find
3 4 2 1
```

Example: repeated application by (¨) each operator.

```
      ∊∘⎕A¨'This' 'And' 'That'
 1 0 0 0  1 0 0  1 0 0 0
```

Locked Functions & Operators

A defined operation may be locked by the system function `⎕LOCK`. A locked operation may not be displayed or edited. The system function `⎕CR` returns an empty matrix of shape 0 0 and the system functions `⎕NR` and `⎕VR` return an empty vector for a locked operation.

Stop, trace and monitor settings may be established by the system functions `⎕STOP`, `⎕TRACE` and `⎕MONITOR` respectively. Existing stop, trace and monitor settings are cancelled when an operation is locked.

A locked operation may not be suspended, nor may a locked operation remain pendent when execution is suspended. The state indicator is cut back as described below.

The State Indicator

The state of execution is dynamically recorded in the STATE INDICATOR. The state indicator identifies the chain of execution for operators, functions and the evaluated or character input/output system variables (⎕ and ⍞). At the top of the state indicator is the most recently activated operation.

Execution may be suspended by an interrupt, induced by the user, the system, or by a signal induced by the system function ⎕SIGNAL or by a stop control set by the system function ⎕STOP. If the interrupt (or event which caused the interrupt) is not defined as a trappable event by the system variable ⎕TRAP, the state indicator is cut back to the first of either a defined operation or the evaluated input prompt (⎕) such that there is no locked defined operation in the state indicator. The topmost operation left in the state indicator is said to be SUSPENDED. Other operations in the chain of execution are said to be PENDENT.

The state indicator may be examined when execution is suspended by the system commands)SI and)SINL. The names of the defined operations in the state indicator are given by the system functions ⎕SI and ⎕XSI while the line numbers at which they are suspended or pendent is given by the system variable ⎕LC.

Suspended execution may be resumed by use of the Branch function (see *Language Reference*). Whilst execution is suspended, it is permitted to enter any APL expression for evaluation, thereby adding to the existing state indicator. Therefore, there may be more than one LEVEL OF SUSPENSION in the state indicator. If the state indicator is cut back when execution is suspended, it is cut back no further than the prior level of suspension (if any).

Examples

```
      ∇ F
[1]     G
      ∇

      ∇ G
[1]     'FUNCTION G'+
      ∇

       ⍎'F'
SYNTAX ERROR
G[1] 'FUNCTION G'+
      ^

       )SI
#.G[1]*
#.F[1]
⍎
```

```
⎕LOCK'G'

      ⍎'F'
SYNTAX ERROR
F[1] G
     ^

      )SI
#.F[1]*
⍎
#.G[1]*
#.F[1]
⍎
```

A suspended or pendent operation may be edited by the system editor or redefined using ⎕FX provided that it is visible and unlocked. However, pendent operations retain their original definition until they complete, or are cleared from the State Indicator. When a new definition is applied, the state indicator is repaired if necessary to reflect changes to the operations, model syntax, local names, or labels.

Dynamic Functions & Operators

A Dynamic Function (operator) is an alternative function definition style suitable for defining small to medium sized functions. It bridges the gap between operator expressions: `rank←⍴∘⍴` and full 'header style' definitions such as:

```
    ∇ rslt←larg func rarg;local...
```

In its simplest form, a dynamic function is an APL expression enclosed in curly braces `{}` possibly including the special characters `⍺` and `⍵` to represent the left and right arguments of the function respectively. For example:

```
      {(+/⍵)÷⍴⍵} 1 2 3 4     ⍝ Arithmetic Mean (Average)
2.5
      3 {⍵*÷⍺} 64           ⍝ ⍺th root
4
```

Dynamic functions can be named in the normal fashion:

```
      mean←{(+/⍵)÷⍴⍵}
      mean¨(2 3)(4 5)
 2.5  4.5
```

Dynamic Functions can be defined and used in any context where an APL function may be found, in particular:

- In immediate execution mode as in the examples above.
- Within a defined function or operator.
- As the operand of an operator such as each (¨).
- Within another dynamic function.
- The last point means that it is easy to define nested local functions.

Multi-Line Dynamic Functions

The single expression which provides the result of the Dynamic Function may be preceded by any number of assignment statements. Each such statement introduces a name which is local to the function.

For example in the following, the expressions `sum←` and `num←` create **local** variables `sum` and `num`.

```
mean←{          ⍝ Arithmetic mean
    sum←+/⍵     ⍝ Sum of elements
    num←⍴⍵      ⍝ Number of elements
    sum÷num     ⍝ Mean
}
```

Note that Dynamic Functions may be commented in the usual way using `⍝`.

When the interpreter encounters a local definition, a new local name is created. The name is shadowed dynamically exactly as if the assignment had been preceded by: `⎕shadow` *name* `⋄`.

It is **important** to note the distinction between the two types of statement above. There can be **many** assignment statements, each introducing a new local variable, but only a **single** expression where the result is not assigned. As soon as the interpreter encounters such an expression, it is evaluated and the result returned immediately as the result of the function.

For example, in the following,

```
mean←{          ⍝ Arithmetic mean
    sum←+/⍵     ⍝ Sum of elements
    num←⍴⍵      ⍝ Number of elements
    sum,num     ⍝ Attempt to show sum,num (wrong)!
    sum÷num     ⍝ ... and return result.
}
```

... as soon as the interpreter encounters the expression `sum,num`, the function terminates with the two element result (`sum,num`) and the following line is not evaluated.

To display arrays to the session from within a Dynamic function, you can use the explicit display forms `⎕←` or `⍞←` as in:

```
mean←{          ⍝ Arithmetic mean
    sum←+/⍵     ⍝ Sum of elements
    num←⍴⍵      ⍝ Number of elements
    ⎕←sum,num   ⍝ show sum,num.
    sum÷num     ⍝ ... and return result.
}
```

Note that local definitions can be used to specify local nested Dynamic Functions:

```
rms←{                       ⍝ Root Mean Square
    root←{⍵*0.5}            ⍝ ∇ Square root
    mean←{(+/⍵)÷⍴⍵}         ⍝ ∇ Mean
    square←{⍵×⍵}            ⍝ ∇ Square
    root mean square ⍵
}
```

Default Left Argument

The special syntax: `⍺←expr` is used to give a default value to the left argument if a Dynamic Function is called monadically. For example:

```
root←{      ⍝ ⍺th root
    ⍺←2     ⍝ default to sqrt
    ⍵*÷⍺
}
```

The expression to the right of `⍺←` is evaluated *only* if its Dynamic Function is called with no left argument.

Guards

A Guard is a Boolean-single valued expression followed on the right by a `':'`. For example:

```
      0≡≡⍵:          ⍝ Right arg simple scalar
      ⍺<0:           ⍝ Left arg negative
```

The guard is followed by a single APL expression: the result of the function.

```
      ⍵≥0: ⍵*0.5     ⍝ Square root if non-negative.
```

A Dynamic function may contain any number of guarded expressions each on a separate line (or collected on the same line separated by diamonds). Guards are evaluated in turn until one of them yields a 1. The corresponding expression to the right of the guard is then evaluated as the result of the function.

If an expression occurs without a guard, it is evaluated immediately as the default result of the function. For example:

```
      sign←{
          ⍵>0: '+ve'      ⍝ Positive
          ⍵=0: 'zero'     ⍝ zero
               '-ve'      ⍝ Negative (Default)
      }
```

Local definitions and guards can be interleaved in any order.

Note again that any code following the first unguarded expression (which terminates the function) could never be executed and would therefore be redundant.

```
      log←{                       ⍝ Append ⍵ to file ⍺.
          tie←⍺ ⎕fstie 0          ⍝ tie number for file,
          cno←⍵ ⎕fappend tie      ⍝ new component number,
          tie←⎕funtie tie         ⍝ untie file,
          1:rslt←cno              ⍝ comp number as shy
result.
      }
```

Shy Result

Dynamic Functions are usually 'pure' functions that take arguments and return explicit results. Occasionally, however, the main purpose of the function might be a side-effect such as the display of information in the session, or the updating of a file, and the value of a result, a secondary consideration. In such circumstances, you might want to make the result 'shy', so that it is discarded unless the calling context requires it. This can be achieved by assigning a dummy variable after a (true) guard:

```
      log←{                        ⍝ Append ⍵ to file ⍺.
          tie←⍺ ⎕fstie 0           ⍝ tie number for file,
          cno←⍵ ⎕fappend tie       ⍝ new component number,
          tie←⎕funtie tie          ⍝ untie file,
          1:rslt←cno               ⍝ comp number as shy
result.
      }
```

Static Name Scope

When an inner (nested) Dynamic Function refers to a name, the interpreter searches for it by looking outwards through enclosing Dynamic Functions, rather than searching back along the execution stack. This regime, which is more appropriate for nested functions, is said to employ **static scope** instead of APL's usual **dynamic scope**. This distinction becomes apparent only if a call is made to a function defined at an outer level. For the more usual inward calls, the two systems are indistinguishable.

For example, in the following function, variable `type` is defined both within `which` itself and within the inner function `fn1`. When `fn1` calls outward to `fn2` and `fn2` refers to `type`, it finds the outer one (with value `'static'`) rather than the one defined in `fn1`:

```
      which←{

          type←'static'

          fn1←{
              type←'dynamic'
              fn2 ⍵
          }

          fn2←{
              type ⍵
          }

          fn1 ⍵
      }

      which'scope'
 static  scope
```

Tail Calls

A novel feature of the implementation of Dynamic Functions is the way in which tail calls are optimised.

When a Dynamic Function calls a sub-function, the result of the call may or may not be modified by the calling function before being returned. A call where the result is passed back immediately without modification is termed a tail call.

For example in the following, the first call on function `fact` is a tail call because the result of `fact` is the result of the whole expression, whereas the second call isn't because the result is subsequently multiplied by `⍵`.

```
      (⍺×⍵)fact ⍵-1        ⍝ Tail call on fact.
      ⍵×fact ⍵-1           ⍝ Embedded call on fact.
```

Tail calls occur frequently in Dynamic Functions, and the interpreter optimises them by re-using the current stack frame instead of creating a new one. This gives a significant saving in both time and workspace usage. It is easy to check whether a call is a tail call by tracing it. An embedded call will pop up a new trace window for the called function, whereas a tail call will re-use the current one.

Using tail calls can improve code performance considerably, although at first the technique might appear obscure. A simple way to think of a tail call is as a **branch with arguments**. The tail call, in effect, branches to the first line of the function after installing new values for `⍵` and `⍺`.

Iterative algorithms can almost always be coded using tail calls.

In general, when coding a loop, we use the following steps; possibly in a different order depending on whether we want to test at the 'top' or the 'bottom' of the loop.

1. Initialise loop control variable(s).`⍝ init`
2. Test loop control variable.`⍝ test`
3. Process body of loop.`⍝ proc`
4. Modify loop control variable for next iteration.`⍝ mod`
5. Branch to step 2.`⍝ jump`

For example, in classical APL you might find the following:

```
     ∇ value←limit loop value⍝ init
[1]   top:→(⎕CT>value-limit)/0⍝ test
[2]    value←Next value⍝ proc, mod
[3]    →top⍝ jump
     ∇
```

Control structures help us to package these steps:

```
     ∇ value←limit loop value⍝ init
[1]    :While ⎕CT<value-limit⍝ test
[2]        value←Next value⍝ proc, mod
[3]    :EndWhile⍝ jump
     ∇
```

Using tail calls:

```
loop←{⍝ init
    ⎕CT>⍺-⍵:⍵⍝ test
    ⍺ ∇ Next ⍵⍝ proc, mod, jump
}
```

Error-Guards

An **error-guard** is (an expression that evaluates to) a vector of error numbers, followed by the digraph: `::`, followed by an expression, the *body* of the guard, to be evaluated as the result of the function. For example:

```
        11 5 :: ⍵×0 ⍝ Trap DOMAIN and LENGTH errors.
```

In common with `:Trap` and `⎕TRAP`, error numbers 0 and 1000 are catchalls for synchronous errors and interrupts respectively.

When an error is generated, the system searches statically upwards and outwards for an error-guard that matches the error. If one is found, the execution environment is unwound to its state immediately *prior* to the error-guard's execution and the body of the error-guard is evaluated as the result of the function. This means that, during evaluation of the body, the guard is no longer in effect and so the danger of a hang caused by an infinite 'trap loop', is avoided.

Notice that you can provide 'cascading' error trapping in the following way:

```
0::try_2nd
0::try_1st
   expr
```

In this case, if `expr` generates an error, its immediately preceding: `0::` catches it and evaluates `try_1st` leaving the remaining error-guard in scope. If `try_1st` fails, the environment is unwound once again and `try_2nd` is evaluated, this time with no error-guards in scope.

Examples:

`Open` returns a handle for a component file. If the exclusive tie fails, it attempts a share-tie and if this fails, it creates a new file. Finally, if all else fails, a handle of 0 is returned.

```
open←{                      ⍝ Handle for component file ⍵.
    0::0                    ⍝ Fails:: return 0 handle.
    22::⍵ ⎕FCREATE 0        ⍝ FILE NAME:: create new one.
    24 25::⍵ ⎕FSTIE 0       ⍝ FILE TIED:: try share tie.
           ⍵ ⎕FTIE 0        ⍝ Attempt to open file.
}
```

An error in `div` causes it to be called recursively with *improved* arguments.

```
div←{                       ⍝ Tolerant division:: ⍺÷0 → ⍺.
    ⍺←1                     ⍝ default numerator.
    5::↑∇/↓↑⍺ ⍵             ⍝ LENGTH:: stretch to fit.
    11::⍺ ∇ ⍵+⍵=0           ⍝ DOMAIN:: increase divisor.
    ⍺÷⍵                     ⍝ attempt division.
}
```

Notice that some arguments may cause `div` to recur twice:

```
       6 4 2 div 3 2
→      6 4 2 div 3 2 0
→      6 4 2 div 3 2 1
→      2 2 2
```

The final example shows the unwinding of the local environment before the error-guard's body is evaluated. Local name `trap` is set to describe the domain of its following error-guard. When an error occurs, the environment is unwound to expose `trap`'s statically correct value.

```
      add←{
          trap←'domain' ⋄ 11::trap
          trap←'length' ⋄  5::trap
          ⍺+⍵
      }

      2 add 3         ⍝ Addition succeeds
5

      2 add 'three'   ⍝ DOMAIN ERROR generated.
domain

      2 3 add 4 5 6   ⍝ LENGTH ERROR generated.
length
```

Dynamic Operators

The operator equivalent of a dynamic function is distinguished by the presence of either of the compound symbols `⍺⍺` or `⍵⍵` anywhere in its definition. `⍺⍺` and `⍵⍵` represent the left and right operand of the operator respectively.

Example

The following monadic `each` operator applies its function operand only to unique elements of its argument. It then distributes the result to match the original argument. This can deliver a performance improvement over the primitive each (¨) operator if the operand function is costly and the argument contains a significant number of duplicate elements. Note however, that if the operand function causes side effects, the operation of dynamic and primitive versions will be different.

```
      each←{                  ⍝ Fast each:

          shp←⍴⍵              ⍝ Shape and ...

          vec←,⍵              ⍝ ... ravel of arg.

          nub←∪vec            ⍝ Vector of unique elements.

          res←⍺⍺¨nub          ⍝ Result for unique elts.

          idx←nub⍳vec         ⍝ Indices of arg in nub ...
          shp⍴idx⊃¨⊂res       ⍝ ... distribute result.
      }
```

The dyadic `else` operator applies its left (else right) operand to its right argument depending on its left argument.

```
      else←{
          ⍺: ⍺⍺ ⍵     ⍝ True: apply Left operand
             ⍵⍵ ⍵     ⍝ Else,  ..   Right   ..
      }
      0 1 ⌈else⌊¨ 2.5       ⍝ Try both false and true.
2 3
```

Recursion

A recursive Dynamic Function can refer to itself using its name explicitly, but because we allow unnamed functions, we also need a special symbol for implicit self-reference: '∇'. For example:

```
      fact←{          ⍝ Factorial ⍵.
          ⍵≤1: 1      ⍝ Small ⍵, finished,
          ⍵×∇ ⍵-1     ⍝ Otherwise recur.
      }
```

Implicit self-reference using '∇' has the further advantage that it incurs less interpretative overhead and is therefore quicker. Tail calls using '∇' are particularly efficient.

Recursive Dynamic Operators refer to their derived functions, that is the operator bound with its operand(s) using ∇ or the operator itself using the compound symbol: ∇∇. The first form of self reference is by far the more frequently used.

```
      pow←{           ⍝ Function power.
          ⍺=0:⍵       ⍝ Apply function operand ⍺ times.
          (⍺-1)∇ ⍺⍺ ⍵ ⍝ ⍺⍺ ⍺⍺ ⍺⍺ ... ⍵
      }
```

The following example shows a rather contrived use of the second form of (operator) self reference. The exp operator composes its function operand with itself on each recursive call. This gives the effect of an exponential application of the original operand function:

```
      exp←{               ⍝ Exponential fn application.
          ⍺=0:⍺⍺ ⍵        ⍝ Apply operand 2*⍺ times.
          (⍺-1)⍺⍺∘⍺⍺ ∇∇ ⍵ ⍝ (⍺⍺∘⍺⍺)∘( ... ) ... ⍵
      }
      succ←{1+⍵}          ⍝ Successor (increment).
      10 succ exp 0
1024
```

Example: Pythagorean triples

The following sequence shows an example of combining Dynamic Functions and Operators in an attempt to find Pythagorean triples: `(3 4 5)(5 12 13)` ...

```
      sqrt←{⍵*0.5}               ⍝ Square root.

      sqrt 9 16 25
3 4 5

      hyp←{sqrt+/⊃⍵*2}           ⍝ Hypoteneuse of
triangle.

      hyp(3 4)(4 5)(5 12)
5 6.403124237 13

      intg←{⍵=⌊⍵}                ⍝ Whole number?

      intg 2.5 3 4.5
0 1 0

      pyth←{intg hyp ⍵}          ⍝ Pythagorean pair?

      pyth(3 4)(4 9)(5 12)
1 0 1

      pairs←{,⍳⍵ ⍵}              ⍝ Pairs of numbers 1..⍵.

      pairs 3
 1 1  1 2  1 3  2 1  2 2  2 3  3 1  3 2  3 3

      filter←{(⍺⍺ ⍵)/⍵}          ⍝ Op: ⍵ filtered by ⍺⍺.

      pyth filter pairs 12       ⍝ Pythagorean pairs 1..12
 3 4  4 3  5 12  6 8  8 6  9 12  12 5  12 9
```

So far, so good, but we have some duplicates: `(6 8)` is just double `(3 4)`.

```
      rpm←{                     ⍝ Relatively prime?
          ⍵=0:⍺=1               ⍝ C.f. Euclid's gcd.
          ⍵ ∇ ⍵|⍺
      }/¨                       ⍝ Note the /¨

      rpm(2 4)(3 4)(6 8)(16 27)
0 1 0 1

      rpm filter pyth filter pairs 20
 3 4  4 3  5 12  8 15  12 5  15 8
```

We can use an operator to combine the tests:

```
      and←{                         ⍝ Lazy parallel 'And'.
          mask←⍺⍺ ⍵                 ⍝ Left predicate
selects...
          mask\⍵⍵ mask/⍵            ⍝ args for right
predicate.
      }

      pyth and rpm filter pairs 20
 3 4  4 3  5 12  8 15  12 5  15 8
```

Better, but we still have some duplicates: `(3 4) (4 3).`

```
      less←{</⊃⍵}
      less(3 4)(4 3)
1 0

      less and pyth and rpm filter pairs 40
 3 4  5 12  7 24  8 15  9 40  12 35  20 21
```

And finally, as promised, triples:

```
      {⍵,hyp ⍵}¨less and pyth and rpm filter pairs 35
 3 4 5  5 12 13  7 24 25  8 15 17  12 35 37  20 21 29
```

A Larger Example

Function `tokens` uses nested local D-Fns to split an APL expression into its constituent tokens. Note that all calls on the inner functions: `lex`, `acc`, and the unnamed D-Fn in each token case, are *tail calls*. In fact, the *only* stack calls are those on function: `all`, and the unnamed function: `{⍵∨¯1⌽⍵}`, within the 'Char literal' case.

```
    tokens←{                            ⍝ Lex of APL src
line.
        alph←⎕A,⎕Á,'_∆⍙',26↑17↓⎕AV  ⍝ Alphabet for names.
        all←{+/∧\α∊ω}                ⍝ No. of leading α∊ω.
        acc←{(α,↑/ω)lex⊃↓/ω}         ⍝ Accumulate tokens.
        lex←{
            0=⍴ω:α ⋄ hd←↑ω          ⍝ Next char else
done.

            hd=' ':α{               ⍝ White Space.
                size←ω all' '
                α acc size ω
            }ω

            hd∊alph:α{              ⍝ Name
                size←ω all alph,⎕D
                α acc size ω
            }ω

            hd∊'⎕:':α{              ⍝ System Name/Keyword
                size←ω all hd,alph
                α acc size ω
            }ω

            hd='''':α{              ⍝ Char literal
                size←+/∧\{ω∨¯1⌽ω}≠\hd=ω
                α acc size ω
            }ω

            hd∊⎕D,'¯':α{            ⍝ Numeric literal
                size←ω all ⎕D,'.¯E'
                α acc size ω
            }ω

            hd='⍝':α acc(⍴ω)ω       ⍝ Comment
            α acc 1 ω               ⍝ Single char token.
        }
        (0⍴⊂'')lex,ω
    }
      display tokens'xtok←size↑srce ⍝ Next token'
.→---------------------------------------------------.
| .→---. .→. .→---. .→. .→---. .→-. .→-----------. |
| |xtok| |←| |size| |↑| |srce| |  | |⍝ Next token| |
| '----' '-' '----' '-' '----' '--' '------------' |
'∊---------------------------------------------------'
```

Restrictions

Currently **multi-line** Dynamic Functions can't be typed directly into the session. The interpreter attempts to evaluate the first line with its trailing left brace and a `SYNTAX ERROR` results.

Dynamic Functions need not return a result. However even a non-result-returning expression will terminate the function, so you can't, for example, call a non-result-returning function from the middle of a Dynamic Function.

You can trace a Dynamic Function **only** if it is defined on more than one line. Otherwise it is executed atomically in the same way as an execute (`⍎`) expression. This deliberate restriction is intended to avoid the confusion caused by tracing a line and seeing nothing change on the screen.

Dynamic Functions do not currently support `⎕CS`.

Supplied Workspaces

You can find more examples of dynamic functions and operators in workspaces in the **samples\dfns** directory.

DFNS.DWS - a selection of utility functions.

MIN.DWS - an example application.

APL Line Editor

The APL Line Editor described herein is included for completeness and for adherence to the ISO APL standard. See *User Guide* for a description of the more powerful full-screen editor, ⎕ED.

Using the APL Line Editor, functions and operators are defined by entering Definition Mode. This mode is opened and closed by the del symbol, ∇. Within this mode, all evaluation of input is deferred. The standard APL line editor (described below) is used to create and edit operations within definition mode.

Operations may also be defined using the system function ⎕FX (implicit in a ⎕ED fix) which acts upon the canonical (character), vector, nested or object representation form of an operation. (See *Language Reference* for details.)

Functions may also be created dynamically or by function assignment.

The line editor recognises three forms for the opening request.

Creating Defined Operation

The opening ∇ symbol is followed by the header line of a defined operation. Redundant blanks in the request are permitted except within names. If acceptable, the editor prompts for the first statement of the operation body with the line-number 1 enclosed in brackets. On successful completion of editing, the defined operation becomes the active definition in the workspace.

Example

```
      ∇R←FOO
[1]  R←10
[2]  ∇

      FOO
10
```

The given operation name must not have an active referent in the workspace, otherwise the system reports `defn error` and the system editor is not invoked:

```
      )VARS
SALES  X  Y

      ∇R←SALES Y
defn error
```

The header line of the operation must be syntactically correct, otherwise the system reports `defn error` and the system editor is not invoked:

```
      ∇R←A B C D:G
defn error
```

Listing Defined Operation

The ∇ symbol followed by the name of a defined operation and then by a closing ∇, causes the display of the named operation. Omitting the function name causes the suspended operation (i.e. the one at the top of the state indicator) to be displayed and opened for editing.

Example

```
      ∇FOO∇
     ∇ R←FOO
[1]    R←10
     ∇

      )SI
#.FOO[1] *

     ∇
     ∇ R←FOO
[1]    R←10
[2]
```

Editing Active Defined Operation

Definition mode is entered by typing ∇ followed optionally by a name and editing directive.

The ∇ symbol on its own causes the suspended operation (i.e. the one at the top of the state indicator) to be displayed. The editor then prompts for a statement or editing directive with a line-number one greater than the highest line-number in the function. If the state indicator is empty, the system reports `defn error` and definition mode is not entered.

The ∇ symbol followed by the name of an active defined operation causes the display of the named operation. The editor then prompts for input as described above. If the name given is not the name of an active referent in the workspace, the opening request is taken to be the creation of a new operation as described in paragraph 1. If the name refers to a pendent operation, the editor issues the message `warning pendent operation` prior to displaying the operation. If the name refers to a locked operation, the system reports defn error and definition mode is not entered.

The ∇ symbol followed by the name of an active defined operation and an editing directive causes the operation to be opened for editing and the editing directive actioned. If the editing directive is invalid, it is ignored by the editor which then prompts with a line-number one greater than the highest line-number in the operation. If the name refers to a pendent operation, the editor issues the message `warning pendent operation` prior to actioning the editing directive. If the name refers to a locked operation, the system reports `defn error` and definition mode is not entered.

On successful completion of editing, the defined operation becomes the active definition in the workspace which may replace an existing version of the function. Monitors, and stop and trace vectors are removed.

Example

```
      ∇FOO[2]
[2]  R←R*2
[3]  ∇
```

Editing Directives

Editing directives, summarised in Figure 2(iv) are permitted as the first non-blank characters either after the operation name on opening definition mode for an active defined function, or after a line-number prompt.

Syntax	Description
`∇`	Closes definition mode
`[⎕]`	Displays the entire operation
`[⎕n]`	Displays the operation starting at line n
`[n⎕]`	Displays only line n
`[∆n]`	Deletes line n
`[n∆m]`	Deletes m lines starting at line n
`[n]`	Prompts for input at line n
`[n]s`	Replaces or inserts a statement at line n
`[n⎕m]`	Edits line n placing the cursor at character position m where an Edit Control Symbol performs a specific action.

Line Numbers

Line numbers are associated with lines in the operation. Initially, numbers are assigned as consecutive integers, beginning with [0] for the header line. The number associated with an operation line remains the same for the duration of the definition mode unless altered by editing directives. Additional lines may be inserted by decimal numbering. Up to three places of decimal are permitted. On closing definition mode, operation lines are re-numbered as consecutive integers.

The editor always prompts with a line number. The response may be a statement line or an editing directive. A statement line replaces the existing line (if there is one) or becomes an additional line in the operation:

```
      ∇R←A PLUS B
[1]  R←A+B
[2]
```

Position

The editing directive [n], where n is a line number, causes the editor to prompt for input at that line number. A statement or another editing directive may be entered. If a statement is entered, the next line number to be prompted is the previous number incremented by a unit of the display form of the last decimal digit. Trailing zeros are not displayed in the fractional part of a line number:

```
[2]   [0.8]
[0.8] ⍝ MONADIC OR DYADIC +
[0.9] ⍝ A ←→ OPTIONAL ARGUMENT
[1]
```

The editing directive [n]s, where n is a line number and s is a statement, causes the statement to replace the current contents of line n, or to insert line n if there is none:

```
[1] [0] R←{A} PLUS B
[1]
```

Delete

The editing directive [∆n], where n is a line number, causes the statement line to be deleted. The form [n∆m], where n is a line number and m is a positive integer, causes m consecutive statement lines starting from line number n to be deleted.

Edit

The editing directive `[n⎕m]`, where `n` is a line number and `m` is an integer number, causes line number `n` to be displayed and the cursor placed beneath the `m`{th} character on a new line for editing. The response is taken to be edit control symbols selected from:

/	to delete the character immediately above the symbol.
1 to 9	to insert from 1 to 9 spaces immediately prior to the character above the digit.
A to Z	to insert multiples of 5 spaces immediately prior to the character above the letter, where A = 5, B = 10, C = 15 and so forth.
,	to insert the text after the comma, including explicitly entered trailing spaces, prior to the character above the comma, and then re-display the line for further editing with the text inserted and any preceding deletions or space insertions also effected.
.	to insert the text after the comma, including explicitly entered trailing spaces, prior to the character above the comma, and then complete the edit of the line with the text inserted and any preceding deletions or space insertions also effected.

Invalid edit symbols are ignored. If there are no valid edit symbols entered, or if there are only deletion or space insertion symbols, the statement line is re-displayed with characters deleted and spaces inserted as specified. The cursor is placed at the first inserted space position or at the end of the line if none. Characters may be added to the line which is then interpreted as seen.

The line number may be edited.

Examples

```
[1]   [1⎕7]
[1]   R←A+B
     ,→(0=⎕NC'A')⍴1←⎕LC ⋄
[1]   →(0=⎕NC'A')⍴1←⎕LC ⋄ R←A+B
                               .⋄→END
[2]   R←B
[3]   END:
[4]
```

The form `[n⎕0]` causes the line number `n` to be displayed and the cursor to be positioned at the end of the displayed line, omitting the edit phase.

Display

The editing directive `[⎕]`causes the entire operation to be displayed. The form `[⎕n]` causes all lines from line number n to be displayed. The form `[n⎕]` causes only line number `n` to be displayed:

```
[4]   [0⎕]
[0]   R←{A} PLUS B
[0]
[0]   [⎕]
[0]   R←{A} PLUS B
[0.1] ⍝ MONADIC OR DYADIC +
[1]   →(0=⎕NC'A')⍴1+⎕LC ⋄ R←A+B ⋄→END
[2]   R←B
[3]   'END:
[4]
```

Close Definition Mode

The editing directive `∇` causes definition mode to be closed. The new definition of the operation becomes the active version in the workspace. If the name in the operation header (which may or may not be the name used to enter definition mode) refers to a pendent operation, the editor issues the message `warning pendent operation` before exiting. The new definition becomes the active version, but the original one will continue to be referenced until the operation completes or is cleared from the State Indicator.

If the name in the operation header is the name of a visible variable or label, the editor reports `defn error` and remains in definition mode. It is then necessary to edit the header line or quit.

If the header line is changed such that it is syntactically incorrect, the system reports `defn error`, and re-displays the line leaving the cursor beyond the end of the text on the line. Backspace/linefeed editing may be used to alter or cancel the change:

```
[3]   [0⎕]              - display line 0
[0]   R←{A} PLUS B
[0]   R←{A} PLUS B:G;H - put syntax error in line 0
defn error
[0]   R←{A} PLUS B:G;H - line redisplayed
                  ;G;H - backspace/linefeed editing
[1]
```

Local names may be repeated. However, the line editor reports warning messages as follows:

1. If a name is repeated in the header line, the system reports "warning duplicate name" immediately.
2. If a label has the same name as a name in the header line, the system reports "warning label name present in line 0" on closing definition mode.
3. If a label has the same name as another label, the system reports "warning duplicate label" on closing definition mode.

Improper syntax in expressions within statement lines of the function is not detected by the system editor with the following exceptions:

- If the number of opening parentheses in each entire expression does not equal the number of closing parentheses, the system reports "warning unmatched parentheses", but accepts the line.
- If the number of opening brackets in each entire expression does not equal the number of closing brackets, the system reports "warning unmatched brackets", but accepts the line.

These errors are not detected if they occur in a comment or within quotes. Other syntactical errors in statement lines will remain undetected until the operation is executed.

Example

```
[4]   R←(A[;1)=2)⌿⍎EXP,'×2
warning unmatched parentheses
warning unmatched brackets
[5]
```

Note that there is an imbalance in the number of quotes. This will result in a `SYNTAX ERROR` when this operation is executed.

Quit Definition Mode

The user may quit definition mode by typing the INTERRUPT character. The active version of the operation (if any) remains unchanged.

Chapter 3:

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 re-fixed, will the new base class be detected.

If you edit, re-fix 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 re-fixing 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.

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" on page 143 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" on page 150 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*. See"Namelists" on page 68 for details. 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" on page 150 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 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) creates 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 types 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:
   - if it is a reference is to a Field, the value of the Field is obtained
   - if it is a reference is to a Property, the PropertyGet function is run
   - if it is a reference is to a Method, the method is executed
   - 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 invokes 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" on page 143 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 take 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 Base 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`, `)OFF` 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" on page 163 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" on page 162.

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 175) 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
DateTimeFormatInfo).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 field 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" on page 167.

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 which **must appear first** in the body of the Property.
- 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 175).

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 175) 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 (conceptually a vector) 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.

If the expression uses indexing, APL checks that the index or indices are within the bounds of these dimensions, and then calls the PropertyGet or PropertySet function. If the expression specifies a single index, APL calls the PropertyGet or PropertySet function once. If the expression specifies multiple indices, APL calls the function successively.

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.

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.

When APL calls a monadic PropertyGet or PropertySet function, it supplies an argument of type PropertyArguments.

Example

The ComponentFile Class (see page 175) 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 175) 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
     ^
```

ComponentFile 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)'
    ∇

    ∇ 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']←10×2 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){,(⊂⍣α)ω}¨k
      k←⊃(∘.{⊃,/⊂¨α ω})/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 175) used previously.

```
:Class KeyedFile: ComponentFile
    :Field Public Keys
    ⎕ML←0

    ∇ 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←⊃arg.Indexers
          ⎕SIGNAL(~∧/keys∊Keys)/3
          r←{2⊃⍵⊃⎕BASE.Component}¨Keys⍳keys
        ∇
        ∇ set arg;new;keys;vals
          vals←arg.NewValue
          keys←⊃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 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.

Penguin Class Example

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

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

```
:Interface BirdBehaviour
∇ R←Fly ⍝ Returns description of flying capability
∇
∇ R←Lay ⍝ Returns description of egg-laying behaviour
∇
∇ R←Sing ⍝ Returns description of bird-song
∇
: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 in Classes

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` contains 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.

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)~⊖''
        TIME←24 60⊥HOURS MINUTES
        TIME←10×⌊0.5+TIME÷10
        :If ~NEAREST
            I←TEETIMESιTIME
            :If I>ρTEETIMES
            :OrIf (ρNAMES)>⊃,/+/0=ρ¨GOLFERS[I;]
                R.Message←'Not available'
                :Return
            :EndIf
        :Else
            :If ~∨/FREE←(ρNAMES)≤⊃,/+/0=ρ¨GOLFERS
                R.Message←'Not available'
                :Return
            :EndIf
            FREETIMES←(FREE×TEETIMES)+32767×~FREE
            DIFF←|FREETIMES-TIME
            I←DIFFι⌊/DIFF
        :EndIf
        J←(⊃,/0=ρ¨GOLFERS[I;])/ι4
        GOLFERS[I;(ρNAMES)↑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<ρ¨GOLFERS[I;])/GOLFERS[I;]
        R.(OK TeeTime)←1 SLOT
      ∇

:EndClass
```

GolfService Example

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.

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
- 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
OK
```

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 Namespace 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 Date-Time 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: Why ?" on page 147 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" on page 69.

: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.

Properties and Methods defined in an Interface, and the Class functions that implement the Interface, **may not** contain :Access Statements.

: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" on page 184.

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" on page 167.

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
```

:Implements Statement

The `:Implements` statement identifies the function to be one of the following types.

```
:Implements Constructor <[:Base expr]>
:Implements Destructor
:Implements Method <InterfaceName.MethodName>
:Implements Trigger <name1><,name2,name3,...>
```

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 function 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 variable name1, name2, etc.

: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. Its 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 just 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`	A vector that identifies the elements of the Property that are to be referenced or assigned.

PropertyGet Function 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.

The result `R` may be any array. However, for a Keyed Property, `R` must conform to the rank and shape specified by `ipa.Indexers` or be scalar.

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

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 `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 .

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. If any index was elided, the corresponding element of `ipa.Indexers` is `⎕NULL`. However, if the Keyed Property is being assigned in its entirety, without square-bracket indexing, `ipa.Indexers` is undefined (`VALUE ERROR`).

PropertyShape Function 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 . `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`.

Symbolic Index

+	See add/identity/plus
−	See minus/negate/subtract
×	See multiply/signum/times
÷	See divide/reciprocal
⌹	See matrix divide/matrix inverse
|	See magnitude/residue
⌈	See ceiling/maximum
⌊	See floor/minimum
*	See exponential/power
⍟	See logarithm
<	See less
>	See greater
≤	See less or equal
≥	See greater or equal
=	See equal
≠	See not equal
≡	See depth/match
≢	See not match
~	See excluding/not/without
∧	See and/caret pointer
∨	See or
⍲	See nand
⍱	See nor
∪	See union/unique
∩	See intersection
⊂	See enclose/partition/partitioned enclose
⊃	See disclose/mix/pick
?	See deal/roll
!	See binomial/factorial
⍋	See grade up
⍒	See grade down
⍎	See execute
⍕	See format
⊥	See decode
⊤	See encode
○	See circular/pi times
⍉	See transpose
⌽	See reverse/rotate
⊖	See reverse first/rotate first
,	See catenate/laminate/ravel
⍪	See catenate first/table
⍳	See index generator/index of
⍴	See reshape/shape
∊	See enlist/membership/type
⍷	See find
↑	See disclose/mix/take/ancestry
↓	See drop/split
←	See assignment
→	See abort/branch
.	See name separator/decimal point/inner product
∘.	See outer product
∘	See compose
/	See compress/replicate/reduce
⌿	See replicate first/reduce

	first
\	See expand/scan
⍀	See expand first/scan first
¨	See each
⍨	See commute
&	See spawn
⍣	See power operator
⍬	See zilde
¯	See negative sign
_	See underbar character
∆	See delta character
⍙	See delta-underbar character
''	See quotes
⎕	See index/axis
[]	See indexing/axis
()	See parentheses
{}	See braces
⍺	See left argument
⍺⍺	See left operand
⍵	See right argument
⍵⍵	See right operand
#	See Root object
##	See parent object
⋄	See statement separator
⍝	See comment symbol
∇	See function self/del editor
∇∇	See operator self
;	See name separator/array separator
:	See label colon
`:AndIf`	See and if condition
`:Access`	See access statement
`:Case`	See case qualifier
`:CaseList`	See caselist qualifier
`:Class`	See class statement
`:Continue`	See continue branch
`:Else`	See else qualifier
`:ElseIf`	See else-if condition
`:End`	See general end control
`:EndClass`	See endclass statement
`:EndFor`	See end-for control
`:EndHold`	See end-hold control
`:EndIf`	See end-if control
`:EndNamespace`	See endnamespace
`:EndProperty`	See endproperty statement
`:EndRepeat`	See end-repeat control
`:EndSelect`	See end-select control
`:EndTrap`	See end-trap control
`:EndWhile`	See end-while control
`:EndWith`	See end-with control
`:Field`	See field statement
`:For...:In...`	See for statement
`:GoTo`	See go-to branch
`:Hold`	See hold statement
`:Include`	See include statement
`:If`	See if statement
`:Implements`	See implements statement
`:Interface`	See interface statement
`:Leave`	See leave branch
`:Namespace`	See namespace statement
`:OrIf`	See or-if condition
`:Property`	See property statement
`:Repeat`	See repeat statement
`:Return`	See return branch
`:Section`	See section statement
`:Select`	See select statement

⎕FUNTIE	See file untie	⎕NUNTIE	See native file untie
⎕FX	See fix definition	⎕NXLATE	See native file translate
⎕INSTANCES	See instances	⎕OFF	See sign off APL
⎕IO	See index origin	⎕OR	See object representation
⎕KL	See key label	⎕OPT	See variant
⎕LC	See line counter	⎕PATH	See search path
⎕LOAD	See load workspace	⎕PFKEY	See program function key
⎕LOCK	See lock definition	⎕PP	See print precision
⎕LX	See latent expression	⎕PROFILE	See profile application
⎕MAP	See map file	⎕PW	See print width
⎕ML	See migration level	⎕REFS	See cross references
⎕MONITOR	See monitor	⎕R	See replace
⎕NA	See name association	⎕RL	See random link
⎕NAPPEND	See native file append	⎕RTL	See response time limit
⎕NC	See name class	⎕S	See search
⎕NCREATE	See native file create	⎕SAVE	See save workspace
⎕NERASE	See native file erase	⎕SD	See screen dimensions
⎕NEW	See new instance	⎕SE	See session namespace
⎕NL	See name list	⎕SH	See execute shell command/start AP
⎕NLOCK	See native file lock	⎕SHADOW	See shadow name
⎕NNAMES	See native file names	⎕SI	See state indicator
⎕NNUMS	See native file numbers	⎕SIGNAL	See signal event
⎕NQ	See enqueue event	⎕SIZE	See size of object
⎕NR	See nested representation	⎕SM	See screen map
⎕NREAD	See native file read	⎕SR	See screen read
⎕NRENAME	See native file rename	⎕SRC	See source
⎕NREPLACE	See native file replace	⎕STACK	See state indicator stack
⎕NRESIZE	See native file resize	⎕STATE	See state of object
⎕NS	See namespace	⎕STOP	See stop control
⎕NSI	See namespace indicator	⎕SVC	See shared variable control
⎕NSIZE	See native file size	⎕SVO	See shared variable offer
⎕NTIE	See native file tie	⎕SVQ	See shared variable query
⎕NULL	See null item	⎕SVR	See shared variable retract

⎕SVS	See shared variable state
⎕TC	See terminal control
⎕TCNUMS	See thread child numbers
⎕TGET	See get tokens
⎕THIS	See this space
⎕TID	See thread identity
⎕TKILL	See thread kill
⎕TNAME	See thread name
⎕TNUMS	See thread numbers
⎕TPOOL	See token pool
⎕TPUT	See put tokens
⎕TREQ	See token requests
⎕TRACE	See trace control
⎕TRAP	See trap event
⎕TS	See time stamp
⎕TSYNC	See threads synchronise
⎕UCS	See unicode convert
⎕USING	See using path
⎕VFI	See verify and fix input
⎕VR	See vector representation
⎕WA	See workspace available
⎕WC	See window create object
⎕WG	See window get property
⎕WN	See window child names
⎕WS	See window set property
⎕WSID	See workspace identification
⎕WX	See window expose names
⎕XSI	See extended state indicator
⎕XT	See external variable
)CLASSES	See list classes
)CLEAR	See clear workspace
)CMD	See command
)CONTINUE	See continue off
)COPY	See copy workspace
)CS	See change space
)DROP	See drop workspace
)ED	See edit object
)ERASE	See erase object
)EVENTS	See list events
)FNS	See list functions
)HOLDS	See held tokens
)LIB	See workspace library
)LOAD	See load workspace
)METHODS	See list methods
)NS	See namespace
)OBJECTS	See list objects
)OBS	See list objects
)OFF	See sign off APL
)OPS	See list operators
)PCOPY	See protected copy
)PROPS	See list properties
)RESET	See reset state indicator
)SAVE	See save workspace
)SH	See shell command
)SI	See state indicator
)SINL	See state indicator name
)TID	See thread identity
)VARS	See list variables
)WSID	See workspace identity
)XLOAD	See quiet-load workspace

Index