Arrays of Objects

Morten Kromberg
Dyalog Ltd.
South Barn, Minchens Court,
Minchens Lane, Bramley RG26 5BH, UK
+44 1256 830 030
mkrom@dyalog.com

ABSTRACT

This paper discusses key design decisions faced by a language design team while adding Object Oriented language features to Dyalog, a modern dialect of APL. Although classes and interfaces are first-class language elements in the new language, and arrays can both contain and be contained by objects, arrays ***are*** not objects. The use of object oriented features is optional, and users can elect to remain entirely in the functional and array paradigms of traditional APL. The choice of arrays as a "higher" level of organization allows APL's elegant notation for array manipulation to extend smoothly to arrays of objects.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications – *Object-oriented languages, APL*; D.3.3 [**Programming Languages**]: Language Constructs and Features *–classes and objects, data types and structures, patterns.*

General Terms

Algorithms, Performance, Design, Economics, Reliability, Experimentation, Human Factors, Languages, Theory.

Keywords

Arrays, Object Orientation, Functional Programming, Multi-paradigm Languages, Language Design.

1. INTRODUCTION

"APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums." Edsger Dijkstra, 1975. [6]

It is hard to be sure what Edsger Dijkstra was so unhappy about, but it *is* clear that APL was not invented as a notation for the description of loops, tree traversals and other algorithms that traditional "Computer Science" wants to teach. APL is a very simple notation designed by Dr. Kenneth E. Iverson, a Mathematician, as an extensible "domain oriented notation" for describing operations on arrays of data. [3]

1.1 Typical Uses of APL

Users of APL tend to "live in their data". Interactive interpreters allow them to inspect, almost *feel* their way forward, discovering successful snippets of code through experimentation and collecting them into imperative or functional programs according to taste. This does not mean that experienced APL developers who understand the task never plan ahead; they sometimes do "architect" and write pages of code without running them first (there do exist APL systems with more than a million lines of code), but the ability to stop a program at any point and return to experimentation is a key factor for those individuals who are attracted to the use of APL as a tool of thought.

Many of the most successful APL programmers are domain experts from other engineering fields than computer science or software engineering; they are stock brokers and traders who became coders, actuaries and crystallographers, electrical and chemical engineers or other people who chose APL because it allowed them to convert ideas directly into an executable notation and create marketable products without learning "how to program" or (worse) complete and negotiate a requirement specification.

The following examples will hopefully illustrate how APL can be used to explore a problem space. The user typically starts by extracting data from a source, in this case an Excel spreadsheet:

```
      'XL' ⎕WC 'OLEClient' 'Excel.Application'
      data←XL.ActiveSheet.UsedRange.Value2
```

Commentary: ⎕WC is a built-in *system function* which is used to create Windows objects (WC is short for Window Create). It is typically used to create forms and other GUI objects, but can also be used to create an OLE Client object. To avoid conflicts with user-defined names, all APL *primitive functions* are denoted using non-alphabetic symbols, and all system functions (corresponding to core library methods in other systems) have names which begin with the symbol ⎕ (quad). The user proceeds to examine the data extracted from Excel:

```
      data
 Sales      [Null]  [Null]
 Region   Quantity   Price
 East           10    25
 East            5    27.5
 West           15    22
 West           10    24
 West           12    21
 South          11    29
```

In an APL session, user input is indented from the left margin (and in this article, will also be bold). Output starts at the margin.

Referring to **data** without assigning the result causes APL to display the array in the session log. This paper will not explain all the details of the APL expressions shown. Further documentation is available free of charge in electronic form [4].

Imagine that our task is to calculate the average unit price by region. To do this, we need to compute the total sales value for each region and divide it by the number of units sold in the region. The user sets out by extracting the relevant data from the sheet into three variables:

```
   data←2 0↓data
   ↓data
 East 10 25  East 5 27.5  West 15 22 …
   region quantity price←↓[1]data
```

As most of the symbols used in APL, the down arrow (↓) has two flavours: When called with a left argument it is called *drop*. The **2 0↓** means drop 2 rows and 0 columns from the (two dimensional) right argument. When called without a left argument, ↓ is called *split*, and removes a dimension of the right argument by splitting the array into nested elements "along" that dimension.

Above, the user drops the 2 title rows from data. He splits data but discovers that the default is to split along the last dimension, while he wanted to split the array into individual columns. This is done by identifying the axis which is to be removed in square brackets following the function symbol (**↓[1]**). Finally, each column is assigned to the three variables, which are (in some ways) more convenient to use than the original matrix. To get comfortable with the new variables, the user performs a couple of experimental calculations and selections:

```
   ⊃region (quantity×price)
 East   East  West  West  West  South
  250  137.5   330   240   252    319
   (region∈'West' 'South')/quantity
15 10 12 11
```

The function *mix* (⊃) is the inverse of *split*, it joins the two arrays on the right (`region`, and the result of `quantity×price`) together, to form a two-row matrix (in this case, simply for display purposes). The final expression above performs a selection, displaying `quantity` for regions West and South. The users' next task is to find a way to do arithmetic "by region":

```
   region
 East East West  West  West  South
   ∪region
 East  West  South
   (∪region)⍳region
1 1 2 2 2 3
```

The symbol ∪ is the function *unique*, which finds the unique elements) of its right argument.. The function *index of*, denoted by ⍳, returns the first position in the left argument where each element of its right argument is found. Having interactively developed a successful algorithm for classifying a vector, the used can now create a *dynamic function* from it, which is the equivalent of a lambda expression[7]:

```
   classify←{(∪⍵)⍳⍵}
```

In its simplest form, a dynamic function is an expression within curly braces, within which the right argument is referred to as **⍵** and the left argument (if present) as **⍺**:

```
   classify region
1 1 2 2 2 3
```

Note that all APL functions, primitive or user-defined, have the same precedence and take the result of the entire expression to their right as an argument (as in mathematical expressions of the form *f g h x*). Arguments do not need to be parenthesized. Many functions also accept a left argument, which often does need to be in parentheses in order to force it to be completely evaluated before the function is called, as in:

```
   (classify region) ⊂ quantity
 10 5   15 10 12   11
```

In the expression above, the function *partitioned enclose* (⊂) is used to cut the right argument into pieces according to the result of (`classify region`). The user has realized that he can compute region sums by applying the plus reduction (**+/**) to each partition using the each operator (¨):

```
   +/¨ (classify region) ⊂ quantity
15 37 11
```

Reduction (**/**) is an *operator*, which has the effect of inserting function to its left between each element of a list. For example, `(+/1 2 3)` is equivalent to `(1+2+3)`, and `(×/1 2 3)` is equivalent to `(1×2×3)`.

On a roll now, our user defines a function to do the regional sum for any argument which has the same length as the `region` variable:

```
   sumReg←{+/¨(classify region)⊂⍵}
   sumReg quantity
15 37 11
   (sumReg quantity×price) ÷ sumReg quantity
25.83333333 22.21621622 29
```

In other words, the average price by region is the regional sales total divided by the regional number of units sold.

The point of this example is not to suggest that the resulting expression is particularly elegant for this simple case. Some would argue that SQL might be a better choice for this simple calculation.

Array purists will probably argue that naming the columns was a bad thing to do, because it removed our ability to apply the partitioned enclose to the original two-dimensional data matrix, and that we should re-factor the entire solution based only on **data**:

```
   regdata←(classify data[;1])⊂[1]0 1↓data
   (∪region),regdata
 East   10 5      25 27.5
 West   15 10 12  22 24 21
 South  11        29
```

The first statement above uses the classification of the first column of data (regions) to partition the 2 columns containing quantity and price (**0 1↓data**). The result of this is stored in `regdata`, a 2-column matrix with one row per unique region. The first column contains lists of prices, the second lists of prices

(as can be seen in the result of the second statement above, which catenates unique regions to `regdata`). We can define a general function to compute weighted averages:

```
    wtdavg←{(+/⍺×⍵)÷+/⍺}
    1 2 wtdavg 10 16
14
```

… and reduce each row of `regdata` using this function:

```
    wtdavg/regdata
25.83333333 22.21621622 29
```

Since each row of `regdata` has two elements, the function is called once for each row, with the content of the two columns as the left and right arguments, respectively.

It is interesting to compare the two alternative expressions that we have arrived at:

```
(sumReg quantity×price) ÷ sumReg quantity
wtdavg/ ({classify data[;1]) ⊂[1] 0 1↓data
```

The first algorithm, where we have separated data by kind and named the groups, may appear more maintainable and flexible at first glance. Unfortunately, this approach is closely bound to the data structure given in this example.

On the other hand, the more "array-oriented" algorithm can relatively easily be extended to other data structures, for example computing weighted averages of any three-column matrix containing keys, values and weights. If we were to add more information, either in the form of discounts and discounted prices, or other classifications than region, it is easy to reuse the expression or extend it to compute everything at once. Keeping the data together as columns of an array is key to the brevity and flexibility of the array-oriented solution.

Hopefully, the example has illustrated the experimental nature of working with arrays that APL users take for granted. An object oriented APL implementation which removed this capability would not be tolerated.

2. ARRAYS CONTAINING OBJECTS

The preceding example did contain one use of an object, in the reference to `XL.ActiveSheet.UsedRange.Value2`[1]. Dyalog supports the "classical" dot notation for navigation of internal and (in this case external) object hierarchies. The example uses a reference to a single Excel Range object, which has an array property called Value2 – which APL very comfortably turns into an APL array for manipulation by the APL user. No problems here. But let us take a look at what happens if we introduce an object based representation of data using a very simple **Sale** class with three fields:

```
:Class Sale
    :Field Public Region
    :Field Public Quantity
    :Field Public Price

    ∇ make(r q p)
      :Access Public
      :Implements Constructor
      Region Quantity Price←r q p
    ∇

:EndClass
```

We can now "cast" each row of the data array into an instance of the **Sale** class using the following expression:

```
sales←{⎕NEW Sale ⍵}¨↓data
```

The each operator (`¨`) is often used where other languages would use a loop. Here, an instance of **Sale** is created for each row of data. The result is an array of instances of Sale.

2.1 Array.MemberName

In order for the object representation to be attractive to APL users, they need to be able to make selections and computations as easily as before. In particular, each array of member data embedded within (any selection from) the array of objects needs to be as easily accessible as it was when all the data was held in an array. We want to be able to do things like partition sales.Price by sales.Region:

```
   sales.Region
 East  East  West  West  West  South
   (classify sales.Region)⊂sales.Price
 25 27.5   22 24 21   29
```

Select by Region (and so forth):

```
   selection←sales.Region∈⊂'East'
   (selection/sales).Quantity
10 5
```

In order to provide APL users with easy access to the arrays of member data that are contained within an array of objects, we decided that an expression in the form **array.member** should be mapped to each object *contained in the array*, rather than the member property *of the array*. This means that the language views arrays as a "higher" level of organization than objects. The arrays themselves have no named members – in this sense, *arrays are not objects*, as far as the APL user is concerned.

The only generally available language that we are aware of which uses the dot in this way is SQL, where names in the form **table.column** can be viewed as a reference to the "column" property of all the records contained in the "table" array. One wonders whether it is relevant, or merely a curiosity, that early work on SQL (prototypes of IBM's "System R") were modeled in APL, back in 1977 [2].

An experimental extension to C# called C⍵ [1] supports *generalized member access* to *streams*, which are described as *structural types*. As Dyalog, C⍵ maps a name following the dot to each element of the stream[2]. Thus, the semantics of the dot vary depending on the type of object, and this seems problematic.

[1] Dyalog supports COM and .NET objects on Windows platforms. Under Linux and Unixes, only internal objects are currently supported.

[2] I learned of C⍵ from one of my reviewers – thanks!

There is also a scripting language for the Macintosh called f-script which embeds an APL-inspired extension to Smalltalk called OOPAL, which has similar capabilities to the above, but uses a notation derived from Smalltalk [5].

2.2 Array.(Expression)

An expression of the form Array.MemberName can be explained as the evaluation of MemberName in the context of each element of Array. A natural extension to this is to evaluate any parenthesized expression following the dot within the same contexts, for example:

```
   sales.(Price×Quantity)
250 137.5 330 240 252 319
```

If we allow this, we can retain *most* of the expressive power of the pure array solution. Say we want to apply a volume-based discount in the East region. We select the relevant records:

```
   eastsales←(sales.Region∈⊂'East')/sales
   eastsales.(Price Quantity)
25 10  27.5 5
```

We have collected references to sales in region East into a smaller array called **eastsales**, and inspected the price and quantity.

```
   eastsales.(Quantity+.≥10 20)
1 0
```

The inner product ***f.g*** is defined as the ***f*** reduction of the result of the application of ***g***. Logical functions like **≥** return 0 for false and 1 for true, so the **+.≥** above counts the number of elements of the list (10 20) which each **Quantity** is greater than or equal to (10 is greater than 1 value, 5 is greater than 0 values). Below, we adjust the price by 0, 10 or 20%, and check the results.

```
  eastsales.(Price-←0.1×Quantity+.≥10 20)
  ⊃eastsales.(Region Price Quantity)
 East  22.5 10
 East  27.5  5
```

This interpretation of a parenthesized expression following the dot provides functionality similar to the *map* operators in Python, Mathematica, Ruby and others. In Ruby, we could write **sales.(Price×Qty)** as:

```
sales.map { |item| item.Price * item.Qty }
```

Indeed, Dyalog itself provides an almost identical mechanism to that of Pythons *map*:

```
   {⍵.Price×⍵.Quantity}¨sales
```

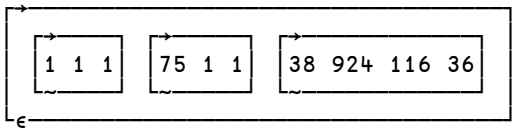
The *each* operator (**¨**) applies the function to its left to each element of the data argument (or arguments, if a left argument is also supplied).

We feel that the simplification provided by direct references to members within a parenthesized expression is a significant one, making the notation a better tool of thought for data exploration without the "unnecessary" reference to *item* or (in Dyalog) **⍵**.

2.3 Array.(Expression1).(Expression2)

Dotted expressions can be nested: If the result of **Array.Expression1** is an array of objects, then **Expression2** is mapped to the elements of that array. Nested expressions may result in nested results[3]:

```
XL.(⊃Workbooks).(⊃Sheets).UsedRange.Count
```



The function *all from* (**⊃**) extracts all the items of an enumerable object into an array. In the above example, Excel had three workbooks open, the first one seems to have three empty sheets (Excel seems to think that even an empty sheet has one cell), the second has 75 used cells in the first sheet, and the third workbook has four sheets, all of which contain some data. Because arrays were returned at two levels in the dotted expression (XL, UsedRange and Count all return a single item), the result has two levels of nesting, the outer level corresponding to workbooks and the inner to sheets.

Although Dyalog only supports internal classes from version 11 (released in October 2006), the current form of the dot notation dates back to the year 2000, when support for *arrays of references* was added. We wholeheartedly agree with the creators of C**ω** that "the power is in the dot".

It is probably also worth pointing out that, because the main focus of this paper is the notation used to access data embedded in objects, this paper only shows a few of the object-oriented features of Dyalog. In a nutshell, Dyalog objects are modeled after C# objects, with support for features like *properties*, *events* and *attributes*. Dyalog classes can consume and derive from any .Net class (& vice versa).

2.4 Introducing New Names

Dyalog takes a strict view of encapsulation; privacy is not implemented by convention, but policed by the interpreter. An attempt to reference a private member of a class or instance gives the standard VALUE ERROR that APL would report for any reference to an unassigned name[4].

However, APL developers are not accustomed to declaring variables before use, and many will find it hard to understand why they should not be allowed to create temporary variables for analytical purposes. For example:

```
   sales.(discount←0.1×Quantity+.≥10 20)
   sales.(amt←(Price×Quantity)-discount)
   sales.amt
223.9 137.5 329.9 239.9 251.9 318.9
```

The temporary variables **discount** and **amt** which are created above are not injected into the actual instances of the Sale objects. The above expressions are not Contextual Class Extensions as described in [8], nor do they modify the class in the way that some hash-table-based object systems (like Python) allow.

---

[3] The result has been specially formatted to show its structure.

[4] None of the extensions described in this document have required any deviations from the ISO 8485 standard for APL.

Ad hoc definitions are placed in a *namespace*[5] which surrounds each instance. The user of a class can inject variables and functions into this space, without breaking the encapsulation of the underlying objects. These names can be used in expressions mapped to the object, and Dyalog system functions for inspecting metadata can be used to separate the different classes of names if required:

```
      sales[1].({ω,[1.5] ⎕NC ω} ⎕NL -2)
 amt         2.1
 discount    2.1
 Price      ¯2.2
 Quantity   ¯2.2
 Region     ¯2.2
```

The above displays the name and name class for all variables, fields or properties accessible within `sales[1]`. A name class of `2.1` identifies an "ordinary" APL variable name, while `¯2.2` indicates a field exposed by the underlying object hierarchy.

An assignment to a public name exposed by the underlying object will be mapped to the named member.

Note that it is possible to introduce variables (or functions) which have the same names as private members of the class. There is no ambiguity, as these names are not visible from the outside.

2.5 Heterogenous Objects

There is no requirement that all the objects in an array be of the same class. Even when all elements of an array have the same class, the mechanism described above allows names to be injected into selected elements. As a result, the elements of an array may not all expose the same names, and this can obviously lead to errors:

```
   sales[1].xyz←99
   sales[1 2].xyz
VALUE ERROR
   …
```

In principle, these errors are no different from any other error that could arise during the mapping of a name or expression to the objects, and can easily be dealt with using selection based on the class of the name (nameclass not equal to zero):

```
   hasxyz←sales.(0≠⎕NC 'xyz')/sales
   hasxyz.xyz
99
```

… or using error guards, depending on the application. The following function returns the right argument (null) for any elements where the expression **xyz** fails:

```
      sales.{0::⍵ ⋄ xyz} ⎕null
99  [Null]  [Null]  [Null]  [Null]  [Null]
```

Diamond (`⋄`) is the statement separator. The first expression in the above function declares an error guard (denoted by `::`), which states that the right argument (`⍵`) should be returned on error code `0` (any error).

[5] Dyalog *namespaces* date back to the previous millennium, and are perhaps best described as "classless instances" into which any APL variable or function can be dynamically injected.

2.6 Expressions Mapped to External Objects

The namespace surrounding each object in an array also provides an "evaluation space" in which the APL interpreter can execute APL expressions which refer to members exposed by external objects (for example, COM or .NET objects). This allows very easy application of APL syntax to external objects and arrays of these:

```
   sheets←XL.ActiveWorkbook.(⎕Sheets)
```

(form an array from the Sheets collection of the active workbook).

```
   sheets.Name⍳⊂'Sheet2'
2
   sheets.('2'=¯1↑Name)
0 1 0
```

The first expression above refers to `sheets.Name` (resulting in a three-element array of character arrays), and searches for the name 'Sheet2' (which is found in the 2nd position). The second expression executes the APL expression `('2'=¯1↑Name)` in the context of each of the three sheet objects, resulting in a three-element boolean list indicating whether the last element of each name is '2'.

Note that injection of APL variables is also allowed for external objects, as in the following expression. The creation of such a variable is not communicated to Excel, the new variable is held in the evaluation space which only exists within the Dyalog interpreter.

```
XL.ActiveCell.(area←RowHeight×ColumnWidth)
```

3. EMPTY ARRAYS OF OBJECTS

In the original design of the APL language, great care was taken to resolve edge conditions where an intermediate value is an empty array. For example, the sum of quantities greater than 1000 (there are none) is zero:

```
   +/(Quantity>1000)/Quantity
0
```

The general rule applied above is that the reduction (`f/`) of an empty array returns the identity element of `f`. Thus 0 is returned for +, 1 for ×, 0 for ∨ (or), 1 for ∧ (and), ¯1.797E308[6] for ⌈ (maximum), and so on. This definition of reduction on an empty array is a consequence of the desire to maintain the following identity for associative `f`.

```
    (f/X) f (f/Y)  ≡  f/ X,Y
```

APL users would take a very dim view of object orientation if it had the side effect that, after 40 years of simplicity, application code needed to handle *empties* as a special case. In other words, we need:

```
   north←(sales.Region∈⊂'North')/sales
   +/north.Quantity
0
```

In classical APL (1966), resolving these problems was relatively straightforward, as the language only recognized two types, characters and numbers (the fact that "numbers" could be one-bit "booleans", one-, two-, four- or eight-byte integers, floats or complex numbers, was an "optimization issue" for the interpreter to worry about). So-called second-generation APL systems (which

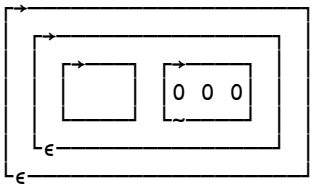
[6] The smallest IEEE double-precision floating point number.

appeared around 1983) support *nested arrays*, in which any item of an array could be another array, for example:

```
data←('East'(1 2 3))('West'(4 5 6))
```

data is a 2-element array. Each of its elements contains a 2-element array; the first of which is a 4-element character array ('East' or 'West'), the second a 3-element integer array (1 2 3) or (4 5 6). When zero items are selected from a *nested* array, 2nd generation APL systems remember the type and structure of the first element of the array and will subsequently use this to generate prototypical items if functions like take (↑) or expand (\) are applied to the empty array. The following expression takes zero elements from **data** to form an empty array, and then proceeds to take one element from the empty array:

```
DISPLAY 1↑0↑data
```



DISPLAY is a library function which shows us structural information, so we can see that the result is a nested array with one element, it contains a nested array with two elements, and that the leaf arrays are a character array containing four blanks (a bit hard, but you *can* count them) and a numeric array containing three zeros. In other words, the type and structure of the first element of **data** has been preserved. This is not *always* ideal; many applications would actually be better off with the leaf arrays (at least those of type character) having zero elements – but it avoids special-casing empties in a lot of cases[7].

Given the successful track record of these strategies (40 years for the simple case and nearly 25 for the nested one), "all" we needed to do was to remember the class of the first element of the array from which nothing was selected, and come up with a definition of the prototypical element of a class. We briefly considered trying to generate "something appropriate" from the class definition, but given that classes can contain references to other classes – and the fact that the nested array prototypes are already only a partial success in many applications, we decided it would be better to let the class implementor define a suitable prototypical element.

Dyalog defines the prototypical element of a class as a (new) instance of the class, created by calling a constructor which takes no arguments. Thus, to be able to work with empty arrays of instances of the **Sale** class, we need to add a constructor to the class which takes no arguments:

```
∇ make0
  :Access Public
  :Implements Constructor
  Region Quantity Price←'' 0 0
∇
```

We are now able to work with empty selections from **sales** and have them behave as an APL user would expect:

```
  +/(0/sales).(Price×Quantity)
0
```

In computing the above result, Dyalog created a prototypical instance, computed **Price×Quantity**, and subsequently applied "the usual rules" to create an empty array from this.

The ability to manufacture fill elements also allows the user to create arrays of prototypes and fill in the details later:

```
north←2↑0↑sales
north.Region←⊂'North'
north.(Quantity Price)←(10 22)(17 21.5)
sales←sales,north
```

For classes which have one or more constructors which take arguments, the interpreter cannot take responsibility for manufacturing the prototypical instance, and the special prototype constructor must be defined. However, for simple classes, this is not necessary. For example, if we have the following class:

```
:Class Point
    :Field Public x←0
    :Field Public y←0
:EndClass
```

We can handle empty Point arrays without further ado:

```
abc←⎕NEW¨3⍴Point
abc.(x y)←(0 0)(20 0)(0 10)
high←(abc.y>50)/abc
+/high.y
0
```

4. SELECTING DATA

An important theme of much of our design work has been to ensure that primitive functions can be applied directly to data which has been embedded in objects. There are two drivers for this: First and foremost preservation of the expressive power of the language, secondly the performance of our interpreter, which is closely linked to the complexity of the syntax which needs to be parsed (once the parsing is done, the system is looping round the arrays involved in the expression, in highly optimized, compiled C).

Some languages make a distinction between *fields*, which are members which contain values which can be directly manipulated, and *properties*, which are accessed through function calls, with optional selection criteria being passed as parameters to the *getter* or *setter* method. Properties have posed a couple of interesting challenges.

Many popular languages support indexing of properties using square brackets – including assignments in the form (**instance.prop[i] = value**). These languages translate such statements into calls to the getter or setter, passing the index as a parameter.

In Dyalog, indexing is only one of a wide variety of selection mechanisms. This also applies to assignments; the language supports a general mechanism known as *selective specification*, in which most simple – and a gradually growing number of *compound* - selection expressions are allowed to the left of the assignment arrow. For example:

```
(3↑name)←'Mr.'
((name='.')/name)←','
(1 1⌽mat)←1
```

[7] Some will argue that this is because APL is lacking a *string* datatype, but that discussion is worthy of a paper of its own.

The meaning of these three statements is: 1) Set the first three characters of **name** to 'Mr.', 2) Replace full stops in **name** with commas, 3) set the elements on the diagonal of **mat** to 1.

4.1 Triggers

It appeared to us that properties have two important uses (as compared to fields): First, to allow a name to refer to data which is not simply an in-memory array, but needs to be retrieved, filtered or generated before being made available. Secondly, to allow an instance to react to the modification of a property, for example by updating the caption of a window or echoing the change to some form of permanent storage.

The performance implications of going through get/set functions led us to believe that our users would probably favor fields over properties more often than users of other languages.

In order to allow direct access to field values, but retain the ability to react to modifications, Dyalog allows a function to be declared as a *trigger* for one or more fields. The interpreter guarantees that the trigger will be called "a short time" after the field has been modified, at this point the documentation is deliberately vague and discourages users from writing code that might depend on the exact timing.

4.2 Numbered vs. Keyed Indexers

As previously mentioned, some object oriented languages allow properties to be *indexed*. From a performance point of view, indexing can be divided into two categories which we call *numbered* and *keyed*: Each index provided can either be a number (an integer) which directly identifies a "physical" position within a data array, or it can be a key (any array) which will be looked up in a list of keys and used to locate or manufacture the requested data.

If we have an instance of a **File** class which has a **Records** property which is indexed using a record *number* and returns the array stored at that index in the file, then users of many languages (including Dyalog) would expect the expression `iFile.Records[2]` to make a single call to the getter to read the file.

However, an APL user who wanted to read every third record would *also* feel that it was natural to write:

```
(iFile.Count⍴0 0 1)/iFile.Records
```

The above creates a selection mask of length `iFile.Count` using the pattern `0 0 1`, and uses that to *compress* the `Records` property. He might also want to supply an array of indices, and expect the following statement to make three calls to the getter, without having to write an explicit loop:

```
iFile.Records[3 6 9]
```

Dyalog allows both of the above if a property is declared as being *numbered*, and the property also provides a *shaper* function in addition to the necessary getter and setter functions. In this case, selection primitives interrogate the shape, and proceed to compute the resulting indices without further reference to the object. The interpreter loops over the getter when the final index set has been found. In the current implementation, one call is made for each index, but it is envisaged that future releases will allow the class implementor to declare the *rank* of the getter and setter functions, and optionally handle multiple indices in a single call.

For *keyed* properties, the array within square brackets is simply passed to the getter, and further selection operations can only be applied after the getter has returned the requested data.

4.3 Default Numbered Properties

Some object oriented languages support the notion of a *default property*, sometimes called an *indexer*, which allows indexing to be applied directly to the object. For example, if **Records** is the default property of the **File** class, then we can simply write `iFile[2]` as shorthand for `iFile.Records[2]`.

This notation causes a problem in APL, because the reference to iFile – in the same way as the number 2 - is an array with zero dimensions (known as a *scalar* in APL terminology). Scalars cannot be indexed in this way – and any attempt to index should therefore give a RANK ERROR.

Because the use of default properties is both elegant and widely supported by other languages, we decided to replace the RANK ERROR by indexing of the default property. The ISO standard explicitly allows extensions which replace existing errors[8].

However, it is not viable to "cheat" in this manner for other selection primitives. Expressions like the following already have well-defined interpretations:

```
3↑iFile
1 0 0/iFile
```

The first one returns a 3-element array containing a pointer to **iFile** followed by two prototypical instances of the **File** class (as discussed in section 3). The second one performs *scalar extension* of the single reference to **iFile**, making three copies of it in order to match the length of the left argument, and then selects one of them (according to the selection mask).

Dyalog supports a functional notation for indexing, denoted by `⌷` (a symbol originally formed on typewriter terminals by overstriking the index brackets by typing "[", backspace, "]"). For example, the following statements are equivalent:

```
array[3]
3⌷array
```

The advantage of having an index function rather than using the traditional "special syntax" is that functions can be used with operators like *each*. For example, we can extract the third element of each sub-array using:

```
3⌷¨'One' 'Two' 'Three'
eor
```

As previously mentioned, when used without a left argument, `⌷` is the function *all of*, which selects all elements of its (right) argument. On arrays, *all of* is the identity function, but if the argument is a reference to an object with a default property (as is generally the case for enumerable objects), *all of* returns the array containing all of the enumerated elements. This allows the user to apply selection functions to the default property by inserting the single symbol `⌷` immediately to the left of the object reference:

```
3↑⌷File
1 0 0/⌷File
```

[8] Although Dyalog has added many features to APL over the last 25 years, conformance with the ISO 8485 standard for APL remains important to our user base.

When evaluating expressions like `1 0 0/⍳`, the interpreter is able to defer calling the getter for the default property until the complete index set has been computed.

All of is also useful in conjunction with external objects like collections, where it can be used to gather the elements of the default property into an array, for example:

```
      XL.ActiveWorkbook.(⍳Sheets).Name
Sheet1  Sheet2  Sheet3
```

To summarize: Fields with *triggers* provide APL users with a half-way house between fields and properties, preserving the ability to perform selective specification on the field value, but allowing modification of the field to trigger an action. *Numbered properties* extend the use of APL selection functions to indexed properties in an efficient manner. Monadic `⍳` makes selections applicable to numbered default properties. Together, these features ensure that data which has been embedded in objects remains within easy (and efficient) reach of all of our selection primitives.

5. CONCLUSION

The intention of this paper has been to describe the successful incorporation of classes as first class citizens of an array language called Dyalog. The resulting language allows data to be encapsulated in objects without compromising the power and flexibility of the APL notation upon which Dyalog is based. In particular, we are pleased that the experimental nature of APL seems to have been maintained, despite our initial fears that object orientation might result in users no longer be able to see the data for all the objects.

This paper does not discuss the benefits of using objects for the type of analytical applications that have traditionally been written in APL. For many analytical applications, the benefits are perhaps marginal. Indeed - one of the key design criteria of the latest version of Dyalog has been to ensure that the object oriented features can be completely ignored by users who want to remain in the functional & array oriented paradigms.

Object orientation makes it significantly easier to integrate Dyalog with modern object-oriented platforms in ways which are natural for users who are accustomed to thinking in terms of arrays. It also makes it possible to provide Dyalog-based components to users of other languages in the form of objects. We hope that it will also make it easier for object oriented developers to experience the power of array thinking – especially for analytical applications.

6. ACKNOWLEDGMENTS

First of all, thanks to the team at Dyalog – in particular Peter Donnelly, Geoff Streeter, John Scholes and John Daintree – and all the others who worked on the product during the 25 years which have culminated in Object Oriented Dyalog! I am also very grateful to Stefano Lanzavecchia for his feedback and interesting discussions about similar capabilities in other languages, and Nicolas Delcros for many constructive suggestions.

I would also like to thank the reviewers of my draft paper; it was worth writing this paper simply to read these reviews, in particular the references to related work: APL users and implementors alike tend to inhabit a parallel universe to that of "mainstream" software development, and are unaware of very much relevant research. We hope that this paper will serve as a token of our desire to change this situation!

7. REFERENCES

[1] Bierman, Meijer and Schulte: *The essence of data access in Cω*, Microsoft (2005) http://research.microsoft.com/Comega/

[2] Blasgen, M.W. and Eswaran, K.P.: *Storage and access in relational databases.* IBM Systems Journal 16, No. 4 (1977) http://www.research.ibm.com/journal/sj/164/ibmsj1604E.pdf

[3] Iverson, Kenneth E. *A Programming Language*. Wiley 1962

[4] Kromberg, Morten: *Introduction to Object Oriented Programming for APL Programmers,* available free with complete product documentation from http://www.lulu.com.

[5] Mougin, Philippe and Ducasse, Stéphane: *OOPAL: Integrating Array Programming in Object-Oriented Programming*, Proceedings of ACM SIGPLAN, 2003.

[6] Murphy, Paul: *APL, COBOL & Dijkstra.* http://blogs.zdnet.com/Murphy/?p=568

[7] Scholes, John: *D-Functions in Dyalog APL*. *http://www.dyalog.com/download/dfns.pdf*

[8] Thorup, Kresten: *Contextual Class Extensions,* AOP Workshop at ECOOP'97.