A Quick Introduction to

Object Oriented Programming For (Impatient) APL Programmers

Version Dated September 26th, 2006

Introduction

Many readers of *An Introduction to Object Oriented Programming for APL Programmers* have commented that it was a bit long winded. Some found it to be a bit "evangelical" or "lecturing". APL developers are an impatient bunch, who want quick results and tend to be convinced by short, powerful code fragments rather than philosophical arguments about why some newfangled technology is good for you.

This *Quick Introduction for Impatient APL Programmers* will therefore dive straight in to the code, with very few explanations.

All of the examples used in this quick introduction are stored as Unicode script files and loaded using SALT– the Simple APL Library Tool. If you want to read more about SALT, it is explained in a separate document which can be found in the Manuals folder below the main Dyalog folder.

If you don't want to read about SALT but just get going with the examples, set the registry entry **`HKEY_CURRENT_USER\Dyalog\Dyalog APL/W 11.0\SALT\AddSALT`** to "1" before starting the workspace. This will cause SALT to be loaded into the Dyalog Session when APL is started.

Example 1: Naming Data

```
:Class Product
    :Field Public Name
    :Field Public Price

    ∇ New(name price)
      :Access Public
      :Implements Constructor
      Name Price←name price
    ∇
:EndClass ⍝ Product
```

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

    ∇ New(product quantity)
      :Access Public
      :Implements Constructor
      Product Quantity←product quantity
      Price←(#.Products.Name⍳⊂Product)
                  ⊃#.Products.Price,¯1
    ∇
:EndClass ⍝ Sale
```

The above classes allow us to work with product sales data using easily recognisable names to refer to elements of the data, but without losing the array nature of the language.

```
      ⎕ML←3

      ⎕SE.SALT.List 'Samples\QuickIntro'
Type  Name     Version  Size  Last Update
      Product           570   08-05-2006 08:59:46
      Sale              756   05-05-2006 13:15:22

      ⎕SE.SALT.Explore 'Samples\QuickIntro' ⍝ See the SALT files

      ⎕SE.SALT.Load 'Samples\QuickIntro\Product'
      ⎕SE.SALT.Load 'Samples\QuickIntro\Sale'

      Products←{⎕NEW Product ⍵}¨('Widget' 100)('Thingamabob' 90)
                               ('Gadget' 150)
      Products.Name
Widget  Thingamabob  Gadget
      ⊃Products.(Name Price)
Widget       100
Thingamabob   90
Gadget       150

      Sales←{⎕NEW Sale ⍵}¨('Widget' 1)('Thingamabob' 5)
                          ('Widget' 10)('Gadget' 100)

      ((Sales.Quantity>10)/Sales).(Price←Price×.9) ⍝ Apply Discount
      Sales.(Amount←Price×Quantity) ⍝ Compute Amount for all Sales
      ⊃((Sales.Amount>500)/Sales).(Product Price Quantity Amount)
Widget  100  10  1000
Gadget  135 100 13500
```

Since the introduction of "references", it has been possible to simulate classes and delivered similar functionality to the above in Dyalog APL using arrays of references to namespaces. This requires building a homegrown "object framework" to create new

instances of the sales or products namespaces. The addition of classes to Dyalog APL provides a single, efficient object framework which everyone can share.

Example 2: Hiding Interface Complexity

We can make APL component files easier to use by implementing a class which makes the components accessible in the form of an array, using file system functions "under the covers". The code for this class, which covers most of the component file functionality including code for manipulating the access matrix, covers a couple of pages, and can be found in the file Classes\Dyalog\ComponentFile.dyalog.

Since we're in a hurry we will not discuss the code, but look at how this class has the potential to simplify application code:

```
      ⎕SE.SALT.Load 'Dyalog\ComponentFile'
      Budget←⎕NEW ComponentFile 'c:\temp\budget'
      Budget.Append¨('Q1' 'Q2' 'Q3' 'Q4')
                   (11 12 14)(12 15 10)(13 17 18)(19 17 10)
1 2 3 4 5

      Budget[1 2]          ⍝ Read 2 components
  Q1  Q2  Q3  Q4   11 12 14
      Budget[3]←⊂13 15 10 ⍝ Replace a component

      ⊃¯4↑⎕Budget          ⍝ "Mix" last 4 components
11 12 14
13 15 10
13 17 18
19 17 10
      Budget.⎕nl ¯2        ⍝ List of "Properties" exposed
 Access  Components  Count  Name  Version
      Budget.Name
c:\temp\budget

      ⎕fnames  ⍝ The file was tied when the instance was created
c:\temp\budget
      )erase Budget
      ⎕fnames ⍝ "Destructor" method has run and untied the file
```

It has always been possible to hide many of the details of working with component files by writing a set of utility functions. A class-based implementation of component file utility functions has a number of benefits:

- The variable which is the handle to the file also provides access to all the functionality exposed by the file, and does not need to be remembered and passed as an additional argument to all file functions.
- When there are no longer any references to the file handle, the file is untied. If the file handle is localised, the file will be untied when the appplication ends.
- APL functions for selection or ordering, like ⊃, ↑, ↓, / or ⌽, can be applied to the collection of components. Expressions like `(1↑⌽⎕Budget)` or

`(0 0 0 0 1/⎕Budget)`, which would both return the last element of the "default property" of the file, only cause one `⎕FREAD` to be performed.

- Classes simplify the use of shared code without name conflicts (the `ComponentFile` class can easily be renamed, or even defined and used anonymously).

The source code management system takes advantage of the last point above and provides a `New` command, which can be used to create instances without using the name ComponentFile in the workspace:

```
      cf←⎕SE.SALT.New 'Dyalog\ComponentFile' 'c:\temp\budget'
```

Example 3: Sharing Code

Classes can be *extended*, when they are used as the basis for a *derived* class. We can use the `ComponentFile` class as the basis for several specialized file types, for example:

```
:Class KeyedFile : ComponentFile
    :Field Private keys←⍬

    ∇ Open filename
      :Implements Constructor :Base filename
      :Access Public Instance

      :If 0=⎕BASE.Count ⍝ Empty: Initialise
          {}Append'Keyed Component File'
          {}Append keys
          {}Append¨3⍴⊂'Reserved'
      :Else             ⍝ Not empty: Validate Format
          'Not a KeyedFile'⎕SIGNAL
                      ('Keyed Component File'≠1⊃Components)/11
          keys←2⊃Components
          'KeyedFile Damaged'⎕SIGNAL((5+⍴keys)≠⎕BASE.Count)/11
      :EndIf
    ∇

    ⍝ --- Instance Properties ---

    :Property Count
    :Access Public Instance
        ∇ r←Get
          r←⍴keys
        ∇
    :EndProperty ⍝ Count

    :Property Keys
    :Access Public Instance
        ∇ r←Get ⍝ Provide Read Access to Keys
          r←keys
        ∇
    :EndProperty
```

(continued on next page)

```
    :Property Keyed Default Item
    :Access Public Instance
        ∇ r←Get arg
          r←Components[5+keys⍳1⊃arg.Indexers] ⍝ Read components
        ∇
        ∇ Set args;m;ix
          ix←1⊃args.Indexers       ⍝ Extract index on 1st dimension
          :If ∨/m←~ix∈keys         ⍝ Are any keys unknown?
              keys←keys,m/ix       ⍝ Add new keys
              Components[2]←⊂keys  ⍝ Rewrite key component
              {}Append¨m/args.NewValue ⍝ Append data
              Components[5+keys⍳(~m)/ix]←(~m)/args.NewValue ⍝ Replace
          :Else                    ⍝ All keys known
              Components[5+keys⍳ix]←args.NewValue ⍝ Replace all values
          :EndIf
        ∇
    :EndProperty

:EndClass ⍝ KeyedFile
```

The code above defines a new class called **KeyedFile** which provides all the functionality of **ComponentFile**, but reserves the first 5 components of the file and uses them to provide "keyed" access to the file:

```
      ⎕SE.SALT.Load 'Dyalog\ComponentFile'    ⍝ If not already loaded
      ⎕SE.SALT.Load 'Samples\Files\KeyedFile'
      k1←⎕new KeyedFile 'c:\temp\keyedfile'
      k1['Jan' 'Feb' 'Mar']←↓3 4⍴⍳12
      k1['Mar' 'Apr']←(12 11 10 9)(0 0 0 0)
      k1[⊂'Mar']
 12 11 10 9
```

KeyedFile passes much of the functionality of **ComponentFile** straight through:

```
      k1.Name
c:\temp\keyedfile
      k1.Components[2]
  Jan  Feb  Mar  Apr
```

… but **KeyedFile** has redefined the **Count** property to return the number of keys rather than the number of components in the file:

```
      k1.Keys
 Jan  Feb  Mar  Apr
      k1.Count
4
```

Note that the code which implements the extensions in **KeyedFile** can use properties provided by **ComponentFile**. So it reads the keys from the file with a simple:

```
      keys←2⊃Components
```

In the `KeyedFile` code, `⎕BASE` is used to refer to base class functionality in the event of a name conflict. Users of the class can access `ComponentFile` functionality which has been overridden by "casting" the instance:

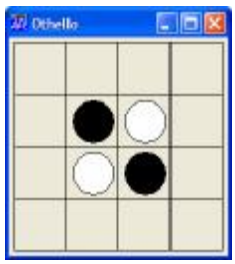
```
      (ComponentFile ⎕CLASS k1).Count
9
```

Inheritance provides a single mechanism which makes it easy to build upon work done by others, without having to understand all the details of the implementation of underlying classes – and in particular without having to compromise due to potential name conflicts between the base and the extended functionality.

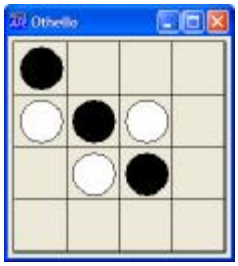
Example 4: Extending the Platform

In addition to extending classes written in APL, it is possible to derive from classes written in other languages, or those provided by the Microsoft.Net framework, and the GUI classes provided by Dyalog APL. Building upon or combining GUI classes to create "custom controls" can help simplify GUI development:

```
    ⎕SE.SALT.Load 'Samples\GUI\Othello' ⍝ SALT can also load Namespaces
    Game←Othello.New 4 4
```




```
      Game.Cells
 .   .   .   .
 .   ⊛   ○   .
 .   ○   ⊛   .
 .   .   .   .
      Game.Cells.State
0 0 0 0
0 1 2 0
0 2 1 0
0 0 0 0
      Game.Cells[2;1 2].State←2  ⍝ White moves
      (2↑1 1⌽Game.Cells).State←1 ⍝ Black moves
```



`Game` is an instance of the built-in class `Form`, which contains a 4 by 4 array of instances of a class called `OthelloSquare`, which derives from the Dyalog GUI class called `Static`. A `Static` is a container which can have a border and contain graphical objects. We "extend" the Static by drawing a circle in it and colouring the circle black or white depending on value of the property `State`. The so-called *setter* function for the `State` property catches assignments to the property causes the circle to be redrawn when the value changes. The definition of class `OthelloSquare` is:

```
:Class OthelloSquare : 'Static'
    ⍝ Implements one Square on an Othello Board

    _State←0 ⍝ Contains value of State property: 0=Empty, 1=Black, 2=White
    :Field Public onClick←'' ⍝ "Event": Put name of Callback fn Here

    :Property Default State
    :Access Public
        ∇ r←Get
          r←_State
        ∇
        ∇ Set args;i
          ⎕SIGNAL(args.NewValue∊0 1 2)↓11          ⍝ Is State Valid?
          i←1+_State←args.NewValue
          Circle.FillCol←(¯16(0 0 0)(255 255 255))[i] ⍝ Button Face, Blk, White
          Circle.FCol←i⊃¯16 0 0                    ⍝ Black outline if not Empty
          ⎕DF i⊃'.⍟o'                              ⍝ Set Display Form
        ∇
    :EndProperty

    ∇ Create args
      :Access Public
      :Implements Constructor :Base args,⊂'BCol' ¯16
      Circle←⎕NEW'Circle'(('Points'(0.5×Size))('Radius'(0.4×⊃Size))
                                              ('FStyle' 0))
      onMouseUp←Circle.onMouseUp←'Select'
      State←0 ⍝ Initially Empty (Calls 'Set' function above)
    ∇

    ∇ r←Select msg
      :If 3=⎕NC onClick           ⍝ If Callback function defined
          (⍎onClick)⎕THIS _State ⍝ Call it passing Ref and Current State
      :EndIf
    ∇

:EndClass ⍝ Square
```

With this class, the function which sets up the Othello game is quite simple (it is displayed at the top of the next page). Once the array of `OthelloSquare`s has been set up, updating the screen only requires setting `State` of selected cells to 0, 1 or 2 – and clicks on squares can be redirected to the callback function of your choice.

```
     ∇ Game←New GridSize;Size;CellIDs;Centre
[1]   ⍝ Initialize Controlling Variables
[2]    ⎕SE.SALT.Load'Samples\GUI\OthelloSquare'
[3]
[4]    Size←50 50                      ⍝ Cell Size
[5]    CellIDs←⍳GridSize               ⍝ Identity of each cell
[6]
[7]   ⍝ Build the Board
[8]    'Game'⎕WC'Form' 'Othello'('Size'(11+Size×GridSize))
                                                ('Coord' 'Pixel')
[9]    Game.Cells←{Game.⎕NEW OthelloSquare (('Size'(1+Size))
                          ('Posn'(5+Size×⍵-1))('Data'⍵))}¨CellIDs
[10]   Game.Cells.onClick←⊂'#.Othello.Click' ⍝ Set up Callback
[11]
[12]  ⍝ Set up the game:
[13]   Centre←0 1+⌊0.5×GridSize
[14]   Game.Cells[Centre;Centre].State←2 2⍴1 2 2 1
[15]   Game.⎕DF'[Othello Game]'
     ∇
```

The callback function in the Othello namespace just flips the state of the cell, and the OthelloSquare setter for the State property takes care of the rest:

```
     ∇ Click msg;Cell
[1]    Cell←⍎msg
[2]    Cell.(State←1+1=State) ⍝ Flip State
     ∇
```

Implementing the algorithm which actually *plays* Othello is left as an exercise for the reader – but updating the screen should now be the easy part!

To illustrate an alternative way to organize the code, the script file OthelloGame.dyalog contains the same example implemented as a class called OthelloGame, with a nested class called Square inside it. To examine this alternative:

```
      )CLEAR
CLEAR WS
      ⎕SE.SALT.Load 'Samples\GUI\OthelloGame'
      Game←⎕NEW OthelloGame (4 4)
      Game.Cells
 .  .  .  .
 .  ⊛  ○  .
 .  ○  ⊛  .
 .  .  .  .
```

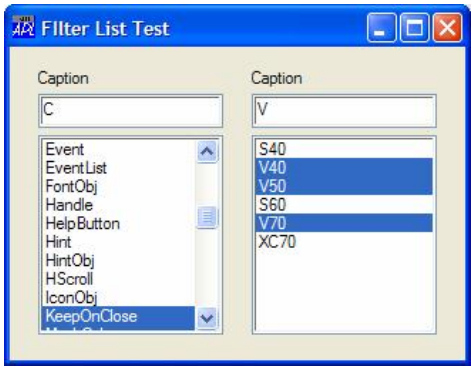
Or simply:

```
      )CLEAR
      Game←⎕SE.SALT.New 'Samples\GUI\OthelloGame' (4 4)
```

… Which creates an instance of the game without naming the class in the workspace.

The final example, in a namespace called `Samples\GUI\FilterListDemo`, shows how we can combine two Dyalog GUI objects – a DropEdit Combo, and ListBox, to create an simple object which provides filtering.

```
      ⎕SE.SALT.Load 'Samples\GUI\FilterListDemo'
      f←FilterListDemo.New
```



The code for the `FilterList` class is also about a page of APL. The above example (the code for the `FilterListDemo.New` function is listed below) shows a form containing two instances of `FilterList` side by side.

```
     ∇ FLForm←New
[1]    ⎕SE.SALT.Load 'Samples\GUI\GUITools'
[2]    ⎕SE.SALT.Load 'Samples\GUI\FilterList'
[3]
[4]    FLForm←⎕NEW'Form'(('Caption' 'FIlter List Test')
                        ('Size'(220 320))('Coord' 'Pixel'))
[5]
[6]    :With FLForm
[7]        TO←⎕NEW'TipField'⍬
[8]        FL1←⎕NEW #.FilterList(('Size'(200 150))('Posn'(10 10))
          ('Caption' 'Properties')('Items'(⎕NL-2))('Filter' 'C'))
[9]        FL2←⎕NEW #.FilterList(('Size'(200 150))('Posn'(10 160))
          ('Caption' 'Volvos')('Items'
            ('S40' 'V40' 'V50' 'S60' 'V70' 'XC70'))('Filter' 'V'))
[10]       FL2.TipObj←'TO' ⋄ FL2.Tip←'Volvos'
[11]   :EndWith
     ∇
```

The above examples have hopefully whetted your appetite for learning more about object oriented programming in APL. If they haven't, it may well be because the type of APL applications you write would not really benefit from the use of classes. Many APL applications have data in a form which is very well suited to its use. As when enclosed arrays were introduced, classes provide new ways to "encapsulate" data and you can easily get seduced into splitting your data up into a large number of very small objects, which may not only slow your system down – it may make actually your data harder to work with and your data structures *LESS* flexible. With a bad object design, you will not be able to see the forest for all the objects.

However, we believe that there are a number of applications which can be simplified through the use of classes – where objects are a valuable tool of thought for APL porgrammers. In addition to this, being able to implement classes in APL makes it much easier to interface to systems written in other languages, which are almost always provided as “class libraries”. And last, but definitely not least: An understanding of objects will help us talk to people using other tools and languages – and perhaps interest some of them in what we are doing!

If any of this sounds interesting, please read the Introduction to OO for APL Programmers or head straight for the Dyalog APL Version 11.0 documentation.