Module20: Dyalog APL Classes

§ 20.1 User defined Classes

§§ 20.1.1 The :Class Structure

In modern computer parlance, a *class* is a blueprint, or scripted template, describing how to build instances which are created from the class definition. An *instance* of a class is an object with methods and properties that is created from the class template.

In Dyalog APL terms, a class is an object created from a script in much the same way as a function is a program created from a script. Both are editable via ⎕ED, both may be fixed from a character representation and both may be traced when run.

In a sense there already are classes in Dyalog. The 70+ built-in GUI objects are object factories for those particular types of objects (`Form`, `Button`, `Group` etc…). However, it is not currently possible to examine these classes directly. The property named `Type` is essentially the *dataType* of the instance.

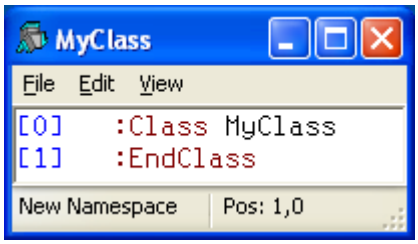
Dyalog version 11 allows you to write classes of your own and generate instances without replicating underlying program code. They have a lot in common with pure namespaces, but with extra functionality. For example, editing and fixing a class, immediately changes all existing instances.

The essence of a class is the class structure in which methods, properties and fields (class *members*) may be defined between the opening and closing keywords. (A *structure* is an essentially multi-line construct that may or may not be amenable to diamondization, depending on the vagaries of the implementation.)

```
:Class MyClass¶...¶:EndClass§          ⍝ Basic class structure wherein members will be defined
```

In an analogous manner to `)ED ∇MyFunction`, a new class may be edited by keysymbol circle (`○`).

```
      )ED ○MyClass
```



All class-specific details are defined inside this control structure.

```
      {RefSc}←{BSc}⎕FIX VecCharVec  ⍝ Fix a visible (if BSc=default=1) class (with Ref)
```

Alternatively, like functions through ⎕FX, a complete character representation may be fixed by ⎕FIX.

```
      ⎕FIX':Class MyClass' .. ':EndClass'
```

This creates a new object called `MyClass`, which is reported by the objects and classes commands.

```
       )Classes                          ⍝ Lists all classes in the space
```

```
      )OBS
MyClass
      )CLASSES
MyClass
```

As yet there are no instances of our class, only the (empty) class definition.

```
      InstRefSc←⎕NEW ClassRefSc          ⍝ Creates an instance of a class (with no argument)
```

New instances of a class may be created (*instantiated*) with ⎕NEW, and what is more, like ⎕DQ,

YOU CAN TRACE INTO IT!

In this way you can follow the steps in the formation of an instance of a class.

```
      MyInst←⎕NEW MyClass
      )Classes
MyClass
      )Obs
MyClass MyInst
```

```
      InstRefVec←⎕INSTANCES ClassRefSc   ⍝ Return all instances of a class
```

All the instances of a particular class (and their ancestry) are returned by ⎕INSTANCES.

```
      ⎕INSTANCES MyClass
 #.[MyClass]
      ⍴⎕INSTANCES MyClass ↳ 1
```

We can create a vector of instances

```
      MyInsts←⎕NEW¨5⍴⊂MyClass
      MyInsts ↳ #.[MyClass] #.[MyClass] #.[MyClass] #.[MyClass] #.[MyClass]
      ⍴MyInsts ↳ 5
```

whose names are reported by the relevant classification.

```
      ⎕NL 2 ↳ MyInsts
      ↓⎕NL 9 ↳ MyClass  MyInst
```

```
      OldCharArr←⎕DF NewCharArr      ⍝ Sets the display form of the current space
```

The display form of all objects - namespaces, GUI objects, classes or instances – may be assigned to any character array, for example,

```
      MyInsts.⎕DF 5↑2⍴¨⎕A
      MyInsts ↳ AA  BB  CC  DD  EE
```

Class definitions may be nested within broader classes, and class definitions may specify other (base) classes or interfaces from which methods and properties may be inherited.

```
:Class MyClass : MyBase¶...¶:EndClass ⍝ Classes with inherited characteristics from base class
```

The base class may be a Dyalog user defined class, a .NET class, an interface or a **Dyalog GUI class**, in which case the object following the second colon in the first line of the class structure must be surrounded by **quotes** as in, for example, `:Class MyGUI : 'Form' …` . Note that GUI objects may be created with ⎕NEW in addition to ⎕WC by again surrounding the class name in quotes, *eg* by ⎕NEW⊂'Form'.

```
      VecCharVec←⎕SRC ClassRefSc      ⍝ Returns a character representation of a class
```

By analogy with

```
      'foo'≡⎕FX ⎕CR'foo' ↳ 1
```

we can write

```
      MyClass≡⎕FIX ⎕SRC MyClass ↳ 1
```

```
      RefSc←⎕THIS                     ⍝ Returns a reference to the current space
```

This system variable may be defined by

```
      ⎕THIS≡⍎''⎕NS'' ↳ 1
```

or

```
      ⎕THIS≡⍎⎕CS'' ↳ 1
```

20.1.1.1 Create some instances of a class and investigate the extent to which they mirror the behaviour of namespace clones.

§§ 20.1.2 The `:Field` Statement

A class is like a namespace, and a field in a class is like an APL variable in the namespace.

```
          ⎡ Private ⎤ ⎡ Instance ⎤
:Field    ⎢         ⎥ ⎢          ⎥ {ReadOnly} MyField ⍝ Defines field called MyField
          ⎣ Public  ⎦ ⎣ Shared   ⎦
```

The keystrings in the field statement may be `Private` (the default if elided) or `Public`, `Instance` (the default if elided) or `Shared`. A public field is visible outside an instance or outside a class (if `Shared`). It may also be defined as `ReadOnly` in which case it may behave rather like an ENUM.

The initial value of the field may be assigned in the `:Field` statement by any APL expression.

```
:Field Public MyField←expr     ⍝ Defines a predefined public field called MyField
```

This is perhaps the most common form of definition of a field. It is visible outside an instance of the class in which it is specified, and it is initialised by an assigned expression at the end of the statement.

If the field statement includes the keystring *`Public`* and also keystring *`Shared`* then the field is visible from outside an instance of the class *and* outside the class itself.

20.1.2.1 Write a class definition such as *`FldClass`* below and attempt to read and assign the values of the fields *`A`*, *`B`*, *`C`* and *`D`* directly from the class, and from an instance of the class. Find the result of `⎕NL -2` in the class and in an instance. Is there any operational difference between field *`A`* and variable *`E`*?

```
:Class FldClass
    :Field A
    :Field Public B
    :Field Public C←1
    :Field Public Shared D←2
     E←3
:EndClass
```

20.1.2.2 Explain the resulting values of fields *`C`* and *`D`* after

```
RefArr←⎕NEW¨3 3⍴FldClass
RefArr.C←3 3⍴⍳6
RefArr.D←3 3⍴⍳6
RefArr.A←3 3⍴⍳6
```

Is *`RefArr.A`* above a field? Set the display form of each instance in *`RefArr`* individually.

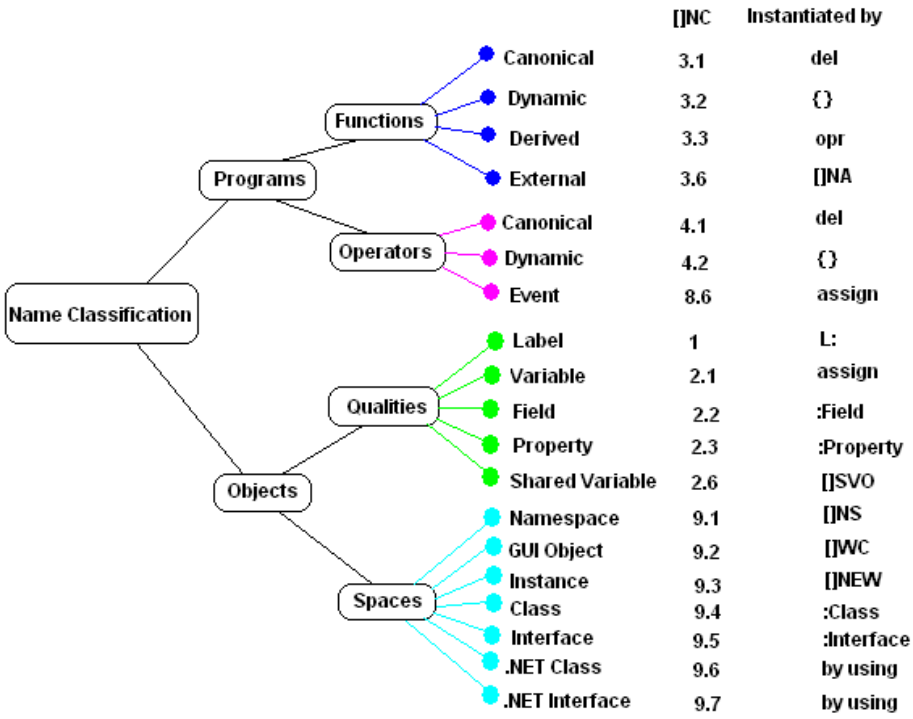
20.1.2.3 In instances of the nested classes below, change the values of both of the fields.

```
:Class MyClass1
    :Field Public MyField1←1
    :Class MyClass2
        :Field Public MyField2←2
    :EndClass
:EndClass
```

Make the fields `Shared` and explore the difference. Is the value of a field in an instance always the same as that in the corresponding class?

§§ 20.1.3 Name sub Classifications of ⎕NC

In Version 11, names are *sub classified*. Below is a vain attempt to categorise these sub classifications.



The above view of the new classification scheme is not authoritative but may be helpful in gaining a feel for the relative meanings of the new categories introduced in 3rd and 4th generation APLs. In particular events are categorised under operators as they essentially take an operand of a user defined (callback) function. Nameclass 2.6 also applies to external properties, *eg* properties of `OLEClient` objects or .NET instances. (Note that external properties have to be used or accessed first before they become visible.)

As well as by some script editor, categories 3.1 and 4.1 may also be created using `⎕FX`, and categories 9.1 and 9.4 may also be created using `⎕FIX`.

20.1.3.1 Examine the differences in, for example, ws ..\samples\OO4APL\Chapter9.dws, between `(⎕NC ⎕NL ⍳9)` and `(⎕NC ⎕NL -⍳9)`.

Namespaces can be defined in script files too via the `:Namespace` structure. We first came across this structure in a .apl script file in §19. This structure can contain all the usual elements of namespaces including classes and other namespaces.

```
:Namespace MyNamespace¶...¶:EndNamespace          ⍝ Basic namespace script structure
```

Conversely, classes may include namespaces by means of the `:Include` keyword in a class definition.

```
    :Include ñ                                    ⍝ Makes contents of ñ accessible within a class
```

Classes that invoke .NET classes may incorporate the `:Using` keyword as an alternative to the system variable `⎕USING` inside a class definition.

```
      :Using ñ{,ass}                              ⍝ ⎕USING←ñ{,ass}
```

§ 20.2 Methods and Properties in Classes

§§ 20.2.1 The ∇ (Method) Structure

An APL function is usually defined inside a class definition between del (∇) symbols as in APL 1. It is also possible to define dynamic functions using braces, or fix a function definition from ⎕FX.

```
∇MyFunctionHeader⍝...⍝∇          ⍝ Basic function definition structure
```

```
:Access {Private|Public} {Instance|Shared}    ⍝ Declares the access attributes of a function
```

A method is a non-dyadic function with public access. Only Public methods can be called from an instance (and directly from a class if :Access Public Shared).

```
∇MyFunctionHeader⍝:Access Public⍝...⍝∇      ⍝ Basic method definition structure
```

20.2.1.1 Show that it is possible to call a Public Shared method such as MyMethod1 below from instances or directly from the class. Add a field and include it in the method, perhaps as the left argument of iota (⍳).

```
:Class MyClass1
    ∇ R←MyMethod1 Int
      :Access Public Shared
      R←⍳Int
    ∇
:EndClass
```

```
      MyClass1.MyMethod1 9 ↳ 1 2 3 4 5 6 7 8 9
      (⎕NEW MyClass1).MyMethod1 8 ↳ 1 2 3 4 5 6 7 8
```

Classes may be fixed from vectors of character vectors. Controlwords and keystrings are not case sensitive.

```
      ⎕FIX':class c1' ':field public shared var←0' '∇r←foo w' 'r←w*2' '∇' ':endclass'
```

If a method is to be called from a language other than Dyalog APL then it is necessary to define precisely the dataType of the arguments and result. This is achieved by the :Signature statement which makes use of the .NET dataTypes as outlined in §0.

```
      :Signature FunctionSyntax          ⍝ Signature declaration statement
```

An example of this is given below. Notice that a :Using statement is required in order to access the System.Int32 object from .NET.

```
:class c2
:using System
    :field public shared var←0
    ∇ r←foo w
      :Access public shared
      :Signature Int32←foo Int32
      r←w*2
    ∇
:endclass
```

When looking into ASP.NET in §19.3, with Dyalog APL as the scripting language, we came across a web service script, eg1.asmx, that contained a class based on the .NET `System.Web.Services.WebService`. The details of this script, in particular the `:Signature` statement, should now be clear.

20.2.1.2 Load workspace **..\Samples\asp.net\tutorial\fruit.dws** and examine the `FruitSelection` class.

```
:Class FruitSelection: Page
  :Using System.Web.UI,System.Web.dll
  :Using
  :Access Public
    ∇ Page_Load
      :Access Public
      :Signature System.Void←Page_Load
      :If 0=IsPostBack
        list.Items.Add⊂'Raspberries'
        list.Items.Add⊂'Blackberries'
        list.Items.Add⊂'Grapes'
        list.Items.Add⊂'Mangoes'
      :EndIf
    ∇
    ∇ Select args
      :Access Public
      :Signature System.Void←Select  System.Object obj, System.EventArgs e
      out.Text←'You selected ',list.SelectedItem.Text
    ∇
:EndClass
```

Navigate to http://82.111.24.53/tutorial.net/frintro6.htm. This tutorial example is based on a version of file ..\Samples\asp.net\tutorial**intro6.aspx** which invokes the above class.

```
<%@Page Language="Dyalog"  Inherits="FruitSelection"  Src="Fruit.dws" %>
<html>..</html>
```

Run the tutorial online and study the explanations given. Notice that the class inherits from the .NET class `System.Web.UI.Page`. Hence the need for the first `:Using` statement. The second `:Using` statement is needed so that the `:Signature` statements can locate all dataTypes derived from `System.Object` class.

§§ 20.2.2 The `:Implements` Statement

The `Page_Load` function is rather special in that, if it exists in the class definition, then it is run automatically every time the class is instantiated.

Generally, in a user defined class one must declare a Public method to be a *constructor* function in order to have the function run on creation of an instance.

```
      :Implements Constructor      ⍝ Statement declares a method to be run on instantiation
```

A function which is declared to be a constructor function can not return a result and must be `Public`. The function may be niladic as in the case of `Page_Load` above, or monadic as in the case of `MyClass2` below.

```
:Class MyClass2
    :Field Public MyField2
    ∇ MyMethod2 Int
      :Access Public
      :Implements Constructor
      MyField2←⍳Int
    ∇
:EndClass
```

When instantiated with an argument of 5, say, then `MyMethod2` sets the value of `MyField2` to `⍳5`.

```
      (⎕NEW MyClass2 5).MyField2 ↳ 1 2 3 4 5
```

It is possible to employ the iota symbol as *index generator* with a numeric vector argument (in which case the argument will match the shape of the result).

```
      (⎕NEW MyClass2 (3 3)).MyField2
 1 1   1 2   1 3
 2 1   2 2   2 3
 3 1   3 2   3 3
```

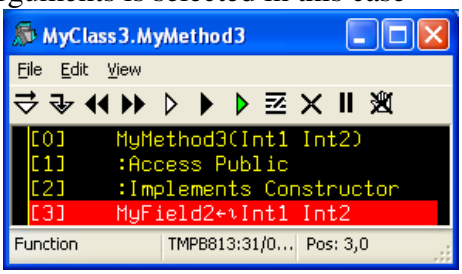
This behaviour is called *overloading* in computerese and in general it requires special treatment such as:

```
:Class MyClass3
    :Field Public MyField2
    ∇ MyMethod2 Int
      :Access Public
      :Implements Constructor
      MyField2←⍳Int
    ∇
    ∇ MyMethod3(Int1 Int2)
      :Access Public
      :Implements Constructor
      MyField2←⍳Int1 Int2
    ∇
:EndClass
```

Tracing the statement

```
      ⍴(⎕NEW MyClass3(5 5)).MyField2
```

shows that the constructor with 2 arguments is selected in this case




```
5 5
```

In this way many monadic constructor functions can be specified in a class definition, each with a different right argument structure. The one that is actually run in any given situation is determined by the structure of the given argument. (This is what must happen under the covers of primitive APL functions, such as the index generator, which encapsulate more than one underlying algorithm.)

20.2.2.1 Write a simple class that has a niladic constructor that initialises the value of a field. Change the constructor function valence to monadic and initialise the field with the argument given to the constructor (the second element of the argument given to ⎕NEW).

```
·····:Implements Constructor :Base expr ⍝ Calls base constructor with argument given by expr
```

The `:Implements Constructor` statement can be supplemented with `:Base` followed by an APL expression. The result of this expression is taken as the argument to the constructor function of the class from which the current class inherits its behaviour (following the colon after the class name in the `:Class` header line, assuming there is one). The constructor of the base class is immediately run with this argument.

```
·····:Implements { Constructor | Destructor | Trigger }   ⍝ Implements Statements
```

A method can contain a `:Implements Destructor` statement in which case the method is run when the last reference to an object is expunged.

A function can contain a `:Implements Trigger Name1,Name2,…` statement in which case the function is executed if any of the variables in the list `Name1, Name2,…` is changed.

20.2.2.2 Write a simple function with a trigger statement and show that this is run when the trigger variable is changed. See Dyalog Version 11 [Help][Latest Enhancements] for an explanation of how to access the old and new values of the variable.

The primitive GUI objects built into Dyalog APL behave very like APL classes except that the name of the object must be placed in quotes when given as an argument to `⎕NEW`, or when referenced as a base class in a `:Class` definition.

20.2.2.3 Create an instance of a `Form` using syntax `⎕NEW 'Form' ''` or `⎕NEW ⊂'Form'`. Hence rewrite `∇makeGrid∇` on page1 of this course to use `⎕NEW` rather than `⎕WC`.

20.2.2.4 Create a class based on a `Form` using syntax `:Class MyClass : 'Form'`. In the constructor function create a black `Static` on the `Form`. Check the hierarchy using new system function `⎕CLASS`.

Hint: `ST←⎕NEW'Static'(⊂'BCol'(0 0 0))`

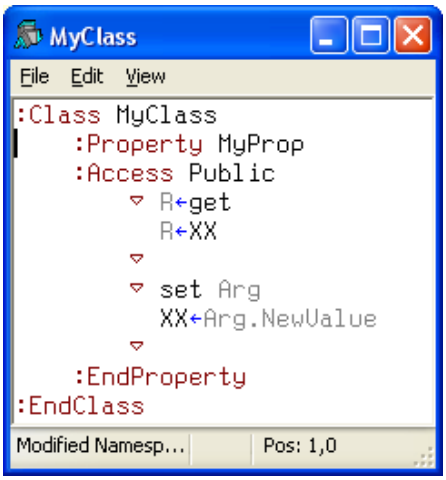
§§ 20.2.3 The `:Property` Structure

A property is like an *adjective* that describes some attribute of an object. (A method is like a *verb* that specifies some action that an object can perform, and a field (and an object itself) is like a *noun*, which has some value.)

`:Property MyProperty ... :EndProperty` A Property Structure

A property of an object is implemented as a `:Property` structure. Inside the structure is a `:Access Public` statement in order to make the property accessible outside of an instance. Also inside the structure may be a case insensitive niladic `∇get∇` function that returns the value of the property, and a case insensitive monadic `∇set∇` function whose (internal instance) argument contains the new property value in the field `NewValue`.

For example, the class definition below bases the simple property `MyProp` on the value of a hidden variable `XX`.




```
MyInst←⎕NEW MyClass
MyInst.MyProp←⍳9
MyInst.MyProp ↳ 1 2 3 4 5 6 7 8 9
```

Of course, any amount of processing could be included in the `∇get∇` and `∇set∇` functions (whose names are allowed to be postfixed with other characters).

As well as `Simple` properties (the default if elided), there are `Numbered` properties and `Keyed` properties. In the case of a `Numbered` property, the property structure normally has a monadic `∇get∇` function and a niladic or monadic `∇shape∇` function.

```
                  ⎡Simple   ⎤
      ......:Property ⎨Numbered ⎬ Name1{,Name2,…}⋄:Access Public⍝…⍝:EndProperty ⍝
                  ⎣Keyed    ⎦
```

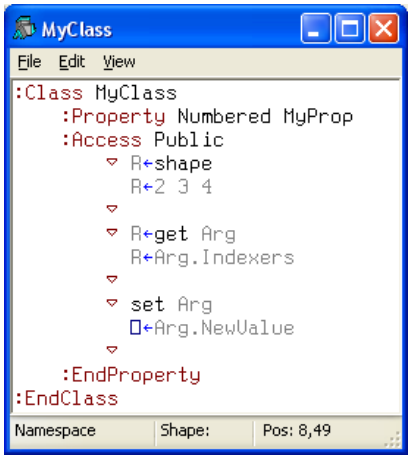
The definitions of these types of properties, and the extra `Default` keystring for `Numbered` properties should be explored in the [Help][Latest Enhancements] or in the glorious *Object Oriented Programming for APL Programmers* to be found in the file **..\manuals\OO for APL'ers, 2006-06-22.pdf**

20.2.3.1 Given a class with the `Numbered` property on the right, trace and interpret the results of expressions:

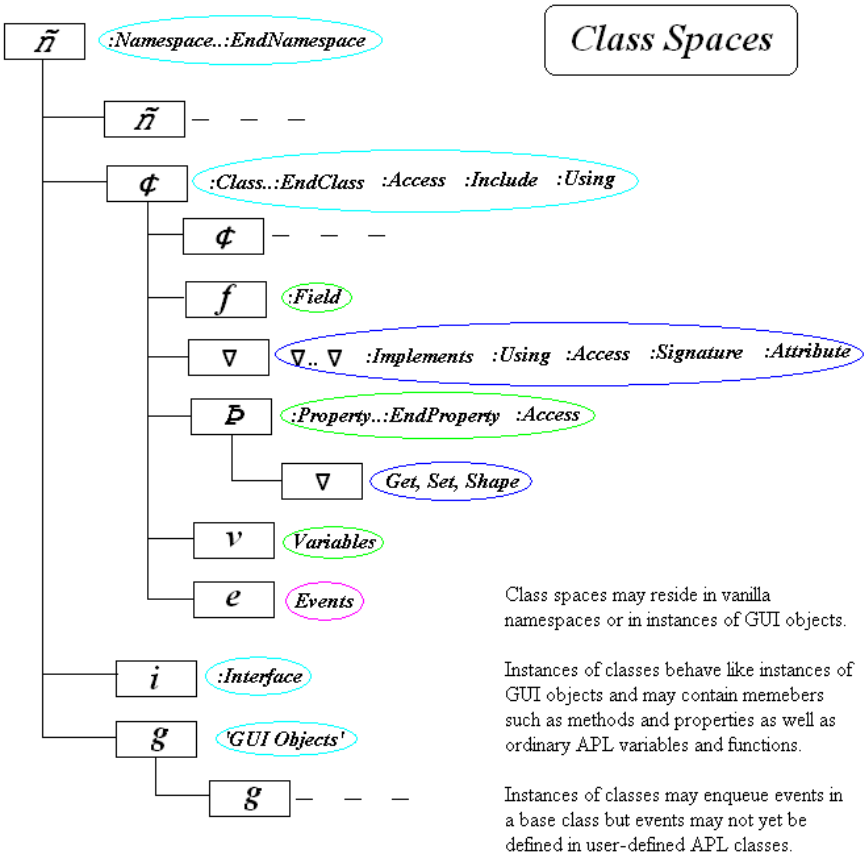
```
(⎕NEW MyClass).MyProp
```

and

```
(⎕NEW MyClass).MyProp←2 3 4⍴⍳99
```



20.2.3.2 Create a class having a monadic constructor that takes an argument of a file name and ties the file, creating it if necessary. Introduce a property that returns or sets the value of the first component of the file.



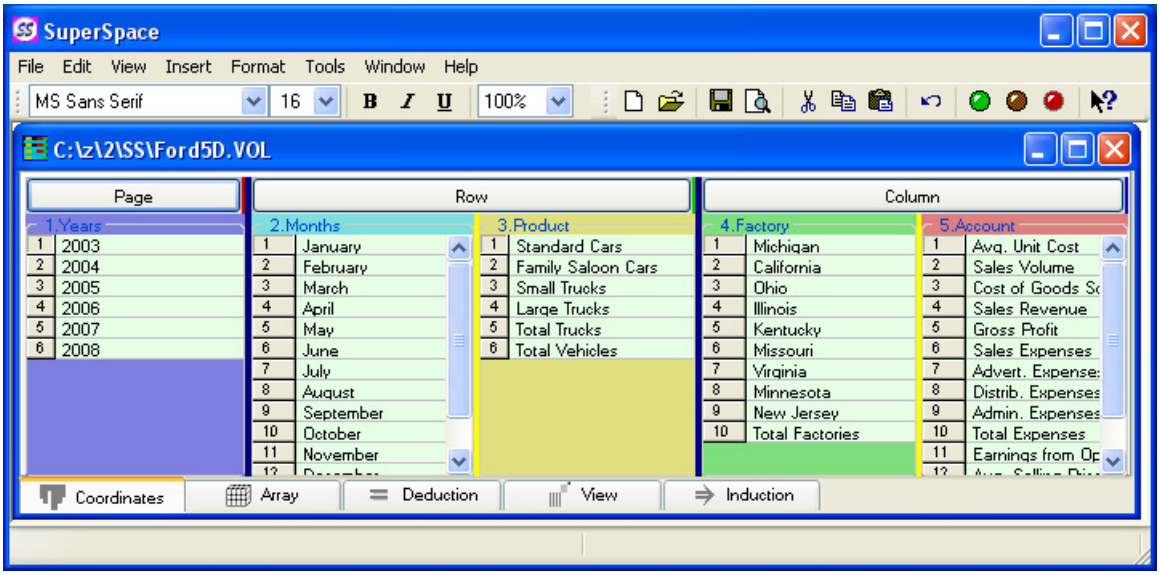
§ 20.3 Architecture with Class Factories

§§ 20.3.1 Designing an Object Model

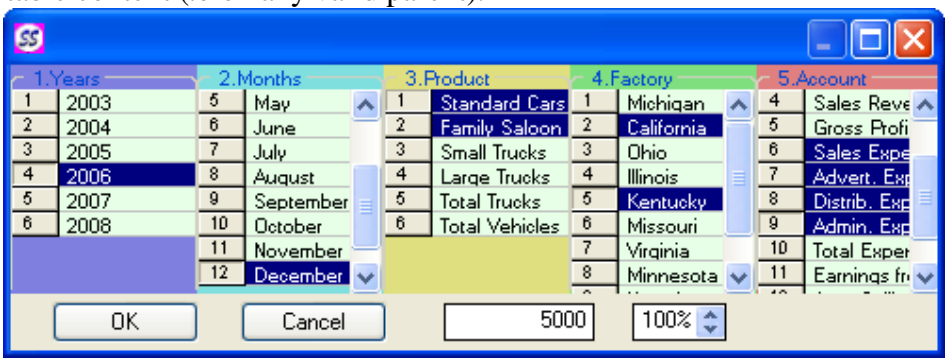
When writing an APL function, it is generally a good idea to carefully consider the header line. What goes in and what comes out and what to call the function and what to call the arguments and result. The first two questions are most important, and determine the functionality completely. The other questions are significant in that they facilitate use; I mean you wouldn't invent a new verb meaning "to tidy up your garage" and call it *xyxy*, and you wouldn't call a new type of garden hose a *ttttt*, would you? Would you?

So when constructing a new application based on classes it is a good idea to carefully consider what objects are needed and how they fit together. A good layout for object-related concepts is given in the *Dyalog Object Reference* manual. Objects are documented with their potential parent and child types as well as their properties, events and methods. Once the basic idea of a GUI object has been grasped, a quick glance at this overview often supplies all the information required to proceed with the application.

For example, let us imagine that we are designing a multi-dimensional application that uses an Index object to group dimensions into those on pages, those on rows and those on columns, in the manner shown on the MDI `SubForm` below.



We wish to use this same Index object in other contexts such as that below, and therefore it makes sense to build an Index class that can build an entire Index object (made up of `SubForm`s, `Button`s, `Splitter`s etc…) in any suitable context (*ie* on any valid parent).



Before diving in to write the APL code (which we love writing when we know what we are supposed to be doing), let us take a little time to specify exactly where we intend to go so that we are less likely to hit unforeseen design faults and other unnecessary limitations. We could do this by filling in the object summary below.

Index Object

Purpose The Index object is a container for three Dimensions collections. Each Dimensions collection contains a number of Dimension objects that specify labels along one axis.

Parents Volume, Variable

Children Dimensions

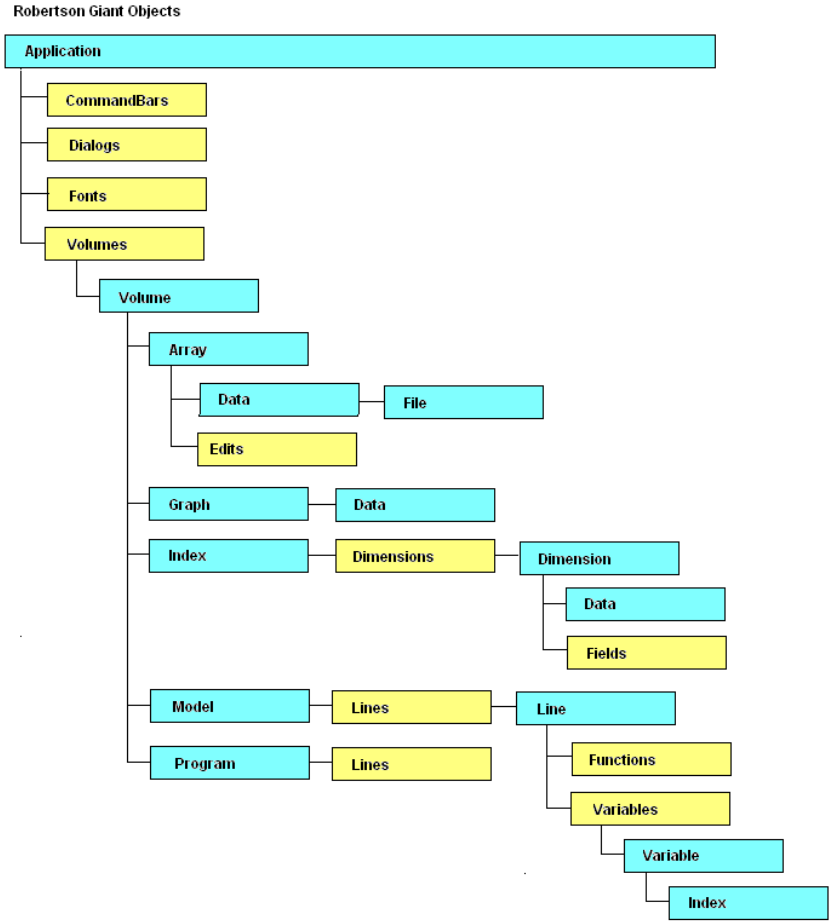
Properties Type, PageRowSplitPosn, RowColumnSplitPosn, Event, SplitterColours, PRC, PRCColours

Events SelectPage, SelectRow, SelectColumn, DragPageRowSplitter, DragRowColumnSplitter

Methods AutoPosnSplitters

Designing systems based on classes introduces this a new discipline that can assist in specifying and coding applications. This *OOP* tool of thought is analogous to the familiar *arguments and results* design model for functional programming, which can assist with lower level coding.

With these building blocks a top-level model can be constructed that summarises the entire application. Getting this model 'right' is an iterative process.



See the Word and Excel help files such as **VBAWRD9.CHM** and **VBAXL10.CHM** for *real* examples.

§§ 20.3.2 Building with Objects

The Dyalog GUI affords some simple examples of GUI objects built from simpler GUI objects (built from API objects?). Consider for example the `Grid` object, built from `Button` and `Edit` objects.

20.3.2.1 Look at some other Dyalog GUI objects and try to find examples of multi-object constructs. Write a class definition that instantiates a new multi-GUI-object construct.

The question arises, "Where in the workspace should I place my class definitions?" Should they be in namespaces in a hierarchy that mirrors the object model? No, because there is no unique position for objects that are used in more than one context. Should they be on file? Not initially, at least. Then where? Paul Mansour's *flipdb* application (see http://www.flipdb.com) is beautifully designed and suggests that classes are placed at the root level and simply need a `#.` in front of their name in order to invoke them in any part of the application. This seems a very nice simple suggestion – in general classes have no definite hierarchy until instantiated, therefore put them all at the root level.

§§ 20.3.3 Encapsulating, Inheriting and Morphing

Encapsulation, Inheritance and Polymorphism are said to be the three pillars of Object Oriented Programming (OOP). APL 1 (core APL) already had significant examples of encapsulation and polymorphism, and APL 3 (GUI) has fine examples of inheritance. Classes in Dyalog version 11 bring explicit examples of OOP that are immediately recognisable to C++ and VB programmers.

Encapsulation is the practice of hiding internal workings from external scrutiny and revealing only those aspects that have been chosen as relevant to the outside world. This concept is most explicit in OOP where an object reveals its characteristics and behaviour through a set of well-defined 'members'. The concept is, however, already well understood in APL 1 where good function definition encourages localisation of all variables that are irrelevant to the intended use. Even the simple plus sign (+) in any digital computer language hides the internal binary processes, which usually obscure rather than enlighten.

Inheritance is a concept indicating that certain characteristics at one level are passed on to the next level. We have seen how classes may be based on other classes and acquire their members; for example `System.Int32` is based on `System.Object`. Even in APL 1 we can surmise that matrix divide (⌹) is based on simple division – even the symbol face has a family resemblance! Inheritance gives a new dimension to encapsulation where functionality of ancestors may be employed by future generations without the need to replicate the functions behind the behaviour or the data behind the family attributes.

Polymorphism means 'having many forms'. Dogs (*Canis familiaris*), for example, take many forms, but they are all dogs. So a class may be *overloaded* with many different possible arguments (the second parameter in `⎕NEW`) to produce different objects with related, but not identical, attributes and behaviour. This is one example of polymorphism in OOP. Even in APL 1 primitive functions such as replicate (`/`) and expand (`\`) may take numeric or character right arguments and give a related, though different, form of result. This is polymorphism of a sort. It is called *operator overloading* in C++, but we have inherited profound operator 'overloading' in APL notation `+.×  ∧.=  ⌈.⌈  …  ⍴  :-)`

As systems grow more complex, computer science borrows more terms from biology. The analogy between the human brain and computer systems has always been close. Now deeper analogies with life forms in general are being forged and there seems no end to the abstract correspondence which computing in general can achieve. APL is a powerful *tool of thought* with a star-studded history. The APL language still leads all the best *executable notations* and continues to *solve real problems*.

20.3.3.1 Take control of your computing needs with Dyalog APL ☺.

F E E D B A C K F O R M

Name . . . . . eMail . . . . . .

Date . . . . . Location . . . . . .

Course . . . . . Instructor . . . . . .

Please indicate your assessment of the following:

poor 1 2 3 4 5 excellent

☐ ☐ ☐ ☐ ☐	Location & Facilities
☐ ☐ ☐ ☐ ☐	Course Content (by module if possible?)
☐ ☐ ☐ ☐ ☐	Course Material (by module if possible?)
☐ ☐ ☐ ☐ ☐	Instructor's Knowledge (by module?)
☐ ☐ ☐ ☐ ☐	How useful was the course to your role?

Please suggest improvements to the course.

Any other critical comments are welcome.

Please give this form to your tutor, or send it to **ROBERTSON (Publishing)**.