DYALOG

The tool of thought for software solutions

Object Oriented Programming for APL Programmers

Dyalog Limited

Minchens Court
Minchens Lane
Bramley
Hampshire, RG26 5BH
United Kingdom

tel: +44 (0)1256 830030
fax: +44 (0)1256 830031
email: support@dyalog.com
http://www.dyalog.com

Version 14.0

Revision: 20170718_001

.

Contents

Preface

This guide is intended to be read from one end to another. Our goal has been not only to explain the details of object orientated functionality of Dyalog APL, but also be mildly provocative and entertaining, to convey a flavour of the thinking which is behind the object oriented extensions to APL, and something about how the development team imagines that you might make use of the new features – in the belief that this will make them easier to understand and use.

If you are in a hurry and want to get a quick feel for OO in APL, you might want to start with "A Quick Introduction to OO for (Impatient) APLers", which is distributed in PDF format.

This guide is not a reference manual, although there is a brief reference at the end of each section. It is recommended that you have aversion of Dyalog APL available for experiments as you work through the guide, and use it to verify your understanding of the new features as they are introduced.

If you have an electronic copy of this guide, you should be able to copy and paste code from the guide into the Dyalog APL editor and session (use the Edit|Paste Unicode menu item). Alternatively, the folder `OO4APL`, included with a Dyalog APL/W installation, contains a workspace with the same name as each of the classes and namespaces used in the guide.

The APL code samples in this document assume migration level 0 (`⎕ML←0`).

Good luck!

The APL Development Team at Dyalog Ltd.
September, 2006

CHAPTER 1

Introduction

Version 11.0 of Dyalog APL introduced *Classes* to the APL language. 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).* For example, the following defines a class which implements a simple timeseries analysis module:

```
:Class TimeSeries

    :Field Public Obs      ⍝ Observed values
    :Field Public Degree←3 ⍝ 3rd degree polys by default

    ∇ r←PolyFit x;coeffs
      :Access Public
      ⍝ Use polyfit of observed points to compute f(x)
      coeffs←Obs⌹(⍳⍴Obs)∘.*0,⍳Degree ⍝ Polynomial coeffs
      r←coeffs+.×⍉x∘.*0,⍳Degree      ⍝ Compute fitted f(x)
    ∇

    ∇ make args
      :Access Public
      :Implements Constructor
      Obs←args
    ∇

:EndClass ⍝ Class TimeSeries
```

The above description declares that instances of `TimeSeries` will have four *Members*: two *Public Fields* called `Obs`and `Degree` (the latter having a default value of 3), a *Public Method* called `PolyFit` (header plus 4 lines of code) and a *Constructor*, which is implemented by the function `make` (header plus 3 lines of code). Note that methods (or functions) begin and end with a `∇`. The term *Public* means that the methods are for "public consumption" by all users of the class.

The system function `⎕NEW` is used to create new instances using the class definition. The first element of the right argument to `⎕NEW` must be a class, the second element contains *instance parameters*, which are passed to the constructor:

```
      ts1←⎕NEW TimeSeries (1 2 2.5 3 6)
```

During *instantiation*, the constructor function `make` is called (note that constructors must be public), and this function initialises the instance by storing its argument in the public field `Obs`. We can now use the instance `ts1` in much the same way as if it were a namespace:

```
      ts1.Obs
1 2 2.5 3 6

      ts1.PolyFit ⍳5
0.9714285714 2.114285714 2.328571429 3.114285714 5.97142857

      1↓⍉ts1.(Obs-PolyFit ⍳⍴Obs)
 0.0 ¯0.1 0.2 ¯0.1 0.0

      ts1.⎕nl ¯2 ⍝ Roughly equivalent to "PropList"
 Degree  Obs
      ts1.⎕nl ¯3 ⍝ NameClass ¯3 is "MethodList"
 PolyFit
```

`⎕NL` accepts negative arguments, in which case it returns a vector of names rather than a matrix – and reports names exposed by underlying class definitions in addition to those in the namespace where it is executed. The `Degree` field allows us to decide the degree of the polynomial function used when fitting the curve. The following example generates a straight line:

```
      ts1.Degree←1
      ts1.PolyFit ⍳5
0.7 1.8 2.9 4 5.1
```

Arrays of instances are treated in much the same way as arrays of namespaces:

```
      ⎕rl←16807 ⍝ So we get the same random numbers
      tss←{⎕NEW TimeSeries ((10×⍳5)+?5⍴5)}¨⍳4

      ↑tss.Obs
11 24 33 43 52
11 24 34 45 52
13 25 31 41 53
14 21 32 41 53

      tss.Degree←2 3 2 1

      1⍕↑tss.PolyFit ⊂⍳10
 11.4 23.0 33.6 43.2 51.8 59.4 66.0 71.6 76.2  79.8
 11.1 23.5 34.8 44.5 52.1 57.2 59.2 57.6 52.0  41.8
 14.0 22.7 32.0 41.9 52.4 63.4 75.0 87.2 99.9 113.2
 12.6 22.4 32.2 42.0 51.8 61.6 71.4 81.2 91.0 100.8
```

APL developers have often used naming standards, and in recent versions of Dyalog APL namespaces, to collect related functionality into modules. Users of Dyalog APL will recognise that an instance is very similar to a *namespace*. One of the advantages of classes is that they make it possible to "clone" a namespace and create multiple data "contexts", without copying the code. This saves space, but more importantly it means that you are less likely to lose track of where the source code is.

Imagine that we have prototyped our way through to a "classical" solution to fitting multiple polynomials. Looking back, we are able to copy the following expressions from our session:

```
      data←(4⍴⊂10×⍳5)+?4⍴⊂5⍴5       ⍝ Generate test data
      exp←0,¨⍳¨2 3 2 1    ⍝ Our polynomials vary by degree
      coeffs←data⌹¨(⊂⍳5)∘.*¨exp
      1⍕↑coeffs+.×¨⍉¨(⊂⍳10)∘.*¨exp
 14.2 23.0 32.8 43.6 55.4 …etc…
```

After thinking a bit more about it, we might identify a couple of potential functions. If we use dfns, we can refactor our solution as follows:

```
      polyfit←{ω⌹(⍳⍴ω)∘.*0,⍳⍺}
      polycalc←{⍺+.×⍉ω∘.*¯1+⍳⍴⍺}
      1⍕↑(2 3 2 1 polyfit¨data) polycalc¨⊂⍳10
 14.2 23.0 32.8 43.6 55.4 …etc…
```

In a traditional APL system, we could now create a workspace called `POLY` with these functions inside, write some documentation explaining how to call them, and then store that documentation in a variable in the workspace, or in a separate document. Anyone wanting to use the functions would have to find the relevant documentation and make sure that he did not already have any functions with these names in his application (or variables with the same name as the documentation). A namespace could be used to isolate the names from our application code.

Classes make it possible for the developer to encapsulate functionality in a way which keeps related code and data together, avoids name conflicts **and** provides some degree of documentation which suggests - and can limit - how the solution is used. This makes the module easier to learn to use, while the control over how the module can be used makes it easier to maintain.

On the other hand, it is also clear that the simple functions `polyfit` and `polycalc` are more *generally* useful than the `PolyFit` method of the class `TimeSeries`, which exposes a specific form of polynomial fitting. The encapsulation of data within instances can make it harder, slower, and sometimes virtually impossible to go "across the grain" and use the properties and methods in a way which is different from that which was intended by the class designer. OO fans will argue that object orientation will help you think more carefully about how things will be used and this is to your advantage. However, APL is often used in problem areas where requirements change *very* unexpectedly: APL is as often used to explore a design as it is for performing an implementation from a complete specification. Providing a flexible solution with OO design is as much of an art, and requires the same insight into where the solution might be heading, as any other technique.

A key design goal for Dyalog APL has been to make it as easy as possible to blend the array and functional paradigms which already make APL so productive, with the object oriented view of data, a Tool of Thought in its own right.

This is one of the reasons why, if you have a namespace `POLY` which contains the two dfns we developed above, you can add a line which says:

```
:Include #.POLY
```

… at the beginning of `:Class TimeSeries`, and subsequently write `PolyFit` as:

```
∇ r←PolyFit x;coeffs
  :Access Public
  ⍝ Use polyfit on observations to compute values at x

  coeffs←Degree polyfit Obs ⍝ Find polynomial coeffs
  r←coeffs polycalc x       ⍝ Compute fitted f(x)
∇
```

This *Object Oriented Programming for APL Programmers* will attempt to illuminate the issues and put the reader in a better position to decide when and how to combine array, functional and object thinking. In order to achieve this, we will:

- First, briefly explore the thinking which lead to the emergence of OO, to get an idea about the type of problems which OO is likely to help us solve.
- Introduce the fundamentals of OO programming using a number of examples written in Dyalog APL.
- Illustrate how the new OO functionality in Dyalog APL makes it easier than ever before to implement components which can be “consumed” by other development tools.
- Where possible, try to remember to discuss alternative solutions, and present some guidelines on how to choose between the various techniques which are available. Given that the temperament and environment of the developer, the department and the company will weigh heavily on any choice of technique, it is clear from the outset that there will be no universal answers.

CHAPTER 2

Origins

Although OO feels like a recent invention to many of us, the first OO language saw the light of day around the same time as the first APL interpreter. SIMULA (SIMUlation LAnguage) was designed and implemented by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Centre between 1962 and 1967, based on ideas which Nygaard had developed during the 1950's[1]. At the same time that Ken Iverson was working on new ways to conceptualise algebra and computation involving large groups of numbers, Nygaard was searching for ways to think about a different type of systems using symbolic notation[2].

The focus of the NCC work was on the simulation of complex systems. Nygaard explained the rationale behind SIMULA as follows:

SIMULA represents an effort to meet this need with regard to discrete-event networks, that is, where the flow may be thought of as being composed of discrete units demanding service at discrete service elements, and entering and leaving the elements at definite moments [sic] of time. Examples of such systems are ticket counter systems, production lines, production in development programs, neuron systems, and concurrent processing of programs on computers.

The desire to describe and model so-called discrete-event networks led to object-oriented notation, in which the description of the ways in which the "service elements" interacted with each other is separated from the details of how each service element (or *object*) manages its internal state. As with APL, the language subsequently evolved into a notation which could be executed by computers.

SIMULA turned out to be a powerful notation for simulating complex systems, and other OO languages followed. Initially, OO languages were used for planning and simulation applications (much the same areas as APL has been most successfully applied to), but with the arrival of graphical user interfaces, which are a form of discrete-event network, and concurrent or networked computing systems, OO languages proved that they had more to offer.

---

[1] For more on the fascinating story about SIMULA, see "Compiling Simula" by Jan Rune Holmevik, http://www.ifi.uio.no/~cim/sim_history.html

[2] Ken Iverson was born in Canada in 1920, but Norway also figures prominently in his heritage. Therefore, we should have reason to hope that the paradigms are not incompatible ☺.

As systems and teams used to implement them have grown in size and complexity, OO has grown from a humble start as a specialist modelling technique to become the most popular paradigm for describing computer systems.

With Dyalog APL version 11.0 onwards, Arrays, Functions and Objects are now happily married. It is possible to have arrays of instances, and instances can contain arrays (of more instances, if necessary). The challenge is to pick the best architecture for a given problem!

Queue SIMULAtion

As an illustration, let us take a look at one of the classical examples which the inventors of SIMULA used the new language to model: The *queue*. Customers arrive at random intervals and enter the queue. An algorithm simulates the time required to process each customer. The goal is to run a number of simulations with different parameters and see how long the queue gets and how long customers have to wait. With luck, we will discover how the queue or system of queues can be optimized, and measure the effect of improving the system without having to perform expensive experiments. In the post-modern age, these systems are probably being used to see how much *longer* the queues will get if the Post Office spends *less* money. Assuming that the planning department has not already been "made redundant".

The following is a simple `Queue` class[3].

[3] For a typical SIMULA solution, see Example 2 on page http://staff.um.edu.mt/jskl1/talk.html#Simulation

```
:Class Queue

    :Field Public Instance History←⍬
    Customers←⍬

    ∇ r←Length
      :Access Public
       r←1⊃⍴Customers
    ∇

    ∇ Join Customer
      :Access Public
      Customers←Customers,⊂Customer,(3⊃⎕AI),1+⍴Customers
      :If 1=⍴Customers ⋄ Serve&⍬ ⋄ :EndIf
    ∇

    ∇ Serve dummy;t;elapsed
    ⍝ Serve queue in new thread when length increases to 1
    ⍝ Stop when queue is empty.

      :Repeat
          ⎕DL 9+?11 ⍝ Processing takes 10-20 seconds
          elapsed←(3⊃⎕AI)-1 2⊃Customers
          ⍝ Since customer entered queue
          History,←⊂(elapsed,1⊃Customers)[2 1 4]
          ⍝ (Cust#, Total Time, Initial Queue Length)
          Customers←1↓Customers ⍝ Customer has gone
      :Until 0=⍴Customers
    ∇

:EndClass ⍝ Class Queue
```

The class has two *public methods* called `Join` and `Length`, which are used to add customers to the end of the queue and report the current length of the queue, respectively. There is a *public field* `History` which contains a record of customers who passed through the system. Note that *members* of a class are *private* unless they are declared to be public. Private members cannot be referenced from outside the class.

While the `TimeSeries` class in the previous chapter only had public members, `Queue` has a *private field* called `Customers`, and a *private method* called `Serve`, which is launched in a background thread when there are customers in the queue (thread-savvy readers are requested to ignore the potential race condition if `Server` drops the last element of `Customers` at the same time as `Join` adds one).

If we have a workspace containing the above class, we can experiment with it from the session:

```
      aQueue←⎕NEW Queue
      aQueue.Join¨1 2 3  ⍝ 3 customers stormed in together
      aQueue.Length      ⍝ Depends on how quickly you type
2
      ⎕←⎕DL 60
60.022
      aQueue.Length
0
```

The variable `History` contains a log of the customers who passed through the queue, the time they spent in line and the length of the queue when they entered it. Since `History` was declared as a public field, we can refer to it from "outside":

```
      ↑aQueue.History        ⍝ Complete history
1 11032  1
2 26800  2
3 41833  3

      ⌈/aQueue.(¯2↑¨History) ⍝ Longest wait, max length
 41833 3
```

Encapsulation

In our `Queue` class, we have decided that the `Customer` field, which contains the list of customers in the queue, is *private*. We do not want users of our class to reference it. We have provided a public method `Length` which makes it possible to determine how long it is.

Why have we not simply made the variable public and allowed the user to inspect `aQueue.Customers` using ⍴ or other primitive functions, rather than doing extra work to implement a method ourselves? We would typically do this if we want to reserve the right to change this part of the implementation in the future, or if we do not wish to take responsibility for the potential bugs resulting from the use of these members ("Warranty void if seal broken").

If we had exposed `Customers` directly, users would have the right to expect that we would continue to have a one-dimensional vector called `Customers` with similar characteristics in the future. It would be virtually impossible for us (as class designers) to estimate the impact of a change to this variable, without reading all the application code to see *exactly* how it was being used. And even then, we would probably get it wrong. The user might also feel tempted to modify the variable, which might cause bugs to be reported to us, even though there were no errors in our code.

If our class evolved into a more general tool where the contents of the queue were not necessarily customers, we could not rename it – so we would end up with a system where variable names were misleading[4]. We cannot `Customers` into a 2-dimensional matrix, store it on file, or make any other architectural changes which might be convenient as requirements evolve and we need to store more or different types of information about the queue.

We have thought ahead a little bit and decided that it will always be reasonable for users of the `Queue` class to ask what the current length of the queue is, and therefore we have exposed a public method for this purpose.

Information hiding is one of the cornerstones of OO. The ability to decide which members of a class are visible to the users on the one hand and the developer of the class on the other is seen as the key to reduced workload, improved reliability and maintainability of the code on *both* sides of the divide. Dyalog APL takes a strict view of encapsulation. It is *not* possible to reference private members (a `VALUE ERROR` will be signalled if you try).

```
      aQueue.Customers
VALUE ERROR
      aQueue.Customers
      ^
```

Clearly, if the designer of a class you are using has decided to hide information which you really need before 9am tomorrow, this can be frustrating. Before you get too concerned, the good news is that there are a variety of techniques for getting past the gatekeepers in an emergency, or in a "prototyping session". We will discuss a couple of these in the following chapters. However, to enjoy the full benefits of OO, it is important to have the discipline to use such tricks only when required, and *re-factor* the class in question at the first available opportunity.

The variable `History` is obviously susceptible to the same problems as `Customers`, and seasoned OO designers would probably consider it to be bad form to expose it. This is perfectly true, exposing all the result data in this form in order to make it easy to analyze is a result of "traditional APL thinking". We will investigate a number of alternatives which we could have used to expose this data in subsequent chapters.

[4] One-letter names tell no lies, of course ☺.

CHAPTER 3

Working with Classes

In order to help you successfully experiment with APL as we explore more OO functionality, let us take a closer look at the practical details of functionality which has been introduced in the first two chapters, and get comfortable with actually working with classes and instances.

The easiest way to create a class is probably to use the editor. Start an `)ED` session, prefixing the name of your new class by a circle (ctrl+O on most keyboards). We're going to use a class for generating random numbers to illustrate some important issues:

```
      )ed ⍥Random
```

This will present you with an empty class, which only contains the `:Class` and `:EndClass` statements. Insert a few lines to create the following simple class:

```
:Class Random

    InitialSeed←42
    ⎕←'Default Initial Seed is: ',⍕InitialSeed

    ∇ make args
      :Implements Constructor
      :Access Public
      :If 0≠⍴args ⋄ InitialSeed←args ⋄ :EndIf
      Reset
    ∇

    ∇ Reset
      :Access Public
      ⎕RL←InitialSeed
      'Random Generator Initialised, Seed is ', ⍕⎕RL
    ∇

    ∇ r←Flat x
      :Access Public
      ⍝ Return x random numbers in range [0,1>, flat
distribution
      r←(¯1+?x⍴100)÷100
    ∇

:EndClass ⍝ Class Random
```

This class can be used to generate sequences of random numbers with a “flat” distribution between 0 and 1 (with only 2 digits, to allow us to easily recognize them in the examples). One advantage of encapsulating it in a class is that it can manage its own seed (⎕RL) completely separately from the rest of the system. We can generate repeatable or suitably randomized sequences according to the requirements of our application.

As you exit from the editor, APL evaluates the class definition from top to bottom. Most of the script consists of function definitions, but in our class there is one APL expression, which is executed as the script is fixed in the workspace:

```
    InitialSeed←42
    ⎕←'Default Initial Seed is: ',⍕InitialSeed
```

As a result, you should see one line written to the session as you leave the editor. If there are errors in any of the executable lines in your script, you will see one or more error messages in the status window, and the class will not be fixed in the workspace.

Let’s perform some experiments with our new class:

```
      rand1←⎕NEW Random 0
Random Generator Initialised, Seed is 42
```

The constructor also has an output statement, which shows us which initial seed was selected for the instance.

```
      rand1.Flat 6
0 0.52 0.73 0.26 0.37 0.19

      rand2←⎕NEW Random 0
Random Generator Initialised, Seed is 42

      rand2.Flat 3
0 0.52 0.73

      ?6 6 6
1 5 3

      rand2.Flat 3
0.26 0.37 0.19
```

As can be seen above, each instance of the generator produces the same sequence of numbers, if the same initial seed is used. The sequence is unaffected by the use of **?** in the root, or indeed anywhere else in the application.

```
      rand3←⎕NEW Random 7913
Random Generator Initialised, Seed is 7913

      rand3.Flat 6
0.06 0.85 0.1 0.29 0.78 0.62

      rand3.Reset
Random Generator Initialised, Seed is 7913

      rand3.Flat 6
0.06 0.85 0.1 0.29 0.78 0.62
```

We can create a generator with a non-default initial seed, and we can reset the sequence. If you are accustomed to using namespaces, the above behaviour will not come as a surprise, as you will be accustomed to each namespace having a separate set of system variables. However, the encapsulation provided by an instance is even stronger, as illustrated by the following example:

```
      rand4←⎕NEW Random 0
Random Generator Initialised, Seed is 42

      rand4.Flat 3
0 0.52 0.73

      rand4.(?6)
4

      rand4.Flat 3
0.26 0.37 0.19
```

If `rand4` had been a namespace rather than an instance, the call to `?` inside `rand4` would have modified ⎕RL in the namespace, and the subsequent call to `Flat` would have continued from a different point in the sequence. However, APL expressions executed on an instance from outside are executed in an "APL execution space", which is *separate* from the space in which the class members run.

In effect, when an instance of a class is created, APL encapsulates it within a namespace. This has always been the case for instances of COM or DotNet classes, and as a result, Dyalog APL also allows the use of APL expressions in parenthesis following the dot after the name of one of these instances. Such APL expressions have access to all the public members of the instance, but are (obviously, since Excel cannot run APL expressions) executed outside the instance itself, as in the following example:

```
      'XL' ⎕WC 'OLECLIENT' 'Excel.Application'
      XL.(Version,'/',OperatingSystem)
11.0/Windows (32-bit) NT 5.01
```

In this example (which would work the same way in Dyalog APL versions 9 or 10), there is an APL expression which references the public properties `Version` and `OperatingSystem` and catenates them together. For consistency, the same approach is used for instances of APL-based classes, rather than simply running the expressions as if the instance was a namespace. Thus, the behaviour of an APL class will not change if it is exported to a DotNet Assembly or a COM DLL and subsequently used from APL.

When an instance method such as `Flat` is referenced in one of these expressions, ***it*** runs in the instance environment. For example, the example on the previous page could have been written:

```
      rand4←⎕NEW Random 0
Random Generator Initialised, Seed is 42
      rand4.Flat 6
0 0.52 0.73 0.26 0.37 0.19

      rand4.Reset
Random Generator Initialised, Seed is 42
      rand4.(⍵(Flat 3) (?6) (Flat 3))
 0 0.52 0.73  4  0.26 0.37 0.19
```

The reverse is required because the rightmost call to `Flat` happens first. The important point is that the call to `(?6)` in the middle of the expression executes in and uses the ⎕RL in the *APL space* and does not modify the value of `⎕RL` in the *instance space*. If you get a different value for the middle element `(4)` when you try the above, see the following comment.

Note: There are a couple of small potential surprises which are worth mentioning:

First, the APL space inherits the values of `⎕IO`, `⎕ML` and other system variables when the object is instantiated – *not* where it is being used. Secondly, if you mistype the name of a property or field in an assignment, this will create a variable in the APL space. For example:

```
      ts1←⎕NEW TimeSeries (1 3 2 4 1)
      ts1
#.[TimeSeries]
      ts1.Obs
1 3 2 4 1

      ts1.obs←1 3 2 4 2 ⍝ Lower "o", note no error message!
      ts1.⎕nl 2         ⍝ obs is in surrounding namespace
obs
      ts1.⎕nl -2        ⍝ List visible fields & variables
 obs  Degree  Obs
```

This is the same behaviour as you would get if you made a spelling error in APL, but might come as a bit of a surprise in an OO setting. However it is desirable to allow a user to introduce own names into the *APL space* and combine them with members of the class for analytical purposes. For example, if `iPlan` is an instance of some object which exposes properties named `Actual` and `Budget`, it may be very useful to introduce a new property:

```
      iPlan.(Variance←Actual-Budget)
```

It is possible that a future version of Dyalog APL will allow the class designer or the user to place restrictions on the introduction of new names into the APL space.

CHAPTER 4

Some Useful Debugging Tricks

The *strict encapsulation* described in the previous chapter may be a bit disconcerting to APL developers, who are accustomed to having access to data on a "want to know" rather than a "need to know" basis. What if we want to know what value ⎕RL or InitialSeed currently have in the instance, because the instance seems to be misbehaving?

The first thing which is important to realize is that if you set a stop in a method, or if you trace into a function call, the internal environment where the method is running is available to you while the method is on the stack. To experience this first hand, create a new instance of Random, trace into a call to Reset or Flat and examine the value of ⎕RL while one of these functions is suspended.

It is also important to realize that classes and instances are *dynamic* in APL (as you would expect)! If you edit a class and fix it, all existing instances will be updated to include the new definition. You can inject temporary methods into a class for debugging purposes. Type)ED Random and add a public method to the class:

```
  ∇ r←RL x
    :Access Public
    r←⎕RL
    :If x≠0 ⋄ ⎕RL←x ⋄ :EndIf
  ∇
```

Using our new method RL, we can now query and set ⎕RL in the instance as follows:

```
      rand1.RL 77
42
      rand1.RL 0
77
```

The system function ⎕CLASS returns the class of an instance. This can be useful in a debugging situation where you are faced with a misbehaving instance of unknown pedigree and need to know which class to edit. You *could* just display the instance, the default display will often tell you the class name, but as we will learn a bit later, it is possible to change this – so it is not a reliable way to determine the class.

```
      ⎕←rand1
#.[Random]

      ⎕CLASS rand4
 #.Random
```

In one of the following chapters, we will show how it is possible to define a *derived* class. A derived class *extends* an existing class by *inheriting* its definition and adding to it. For an instance of a derived class, the result of ⎕CLASS will have more than one element, and document the entire class *hierarchy*. The first element always starts with a reference to the class which was used to create the instance.

The Mother of All Workarounds

The ultimate workaround or back door to break encapsulation is of course the introduction of a public method with a name like `Execute`, which allows you to execute any APL expression you like in the instance space. We can use the `:Include` keyword to embed a namespace containing suitable development tools in a classe. An example namespace called `OOTools` can be found in the workspace of the same name in the `OO4APL` folder. Copy it into the active workspace and edit the `Random` class to add the following statement:

```
:Include OOTools
```

The namespace includes a number of functions which may be useful during development. Functions with names beginning with **`i`** will execute in the instance space, those beginning with **`s`** will run in the shared space (more about the shared space later):

```
      rand1←⎕NEW Random 0
Random Generator Initialised, Seed is 42

      rand1.xxx←'Bingo'   ⍝ New variable in the APL space
      rand1.⎕NL 2    ⍝ Vars in the APL space
xxx
      rand1.iNL 2    ⍝ Vars/Fields/Props in Instance Space
InitialSeed
      rand1.iNC 'Flat'    ⍝ Flat is an instance Function
3
```

And:

```
      rand1.⎕RL←77
      rand1.⎕RL              ⍝ No surprise!
77

      rand1.iExec '⎕RL'      ⍝ Unchanged in the instance
42
      rand1.iExec '⎕RL←99'   ⍝ But now we can change it!
99
```

When you release the application for testing, you should remove the `:Include` statement, or simply the `OOTools` namespace to ensure that your application code does not use any of these "forbidden" methods.

Summary of Chapters 1-4

In the first four chapters, we have discussed the extensions to Dyalog APL which are summarized in the following table.

`)ED ⍥MyClass`	Edits the class `MyClass`
`Instance←⎕NEW MyClass Args`	Create a new instance of `MyClass`, passing `Args` to the constructor.
`:Class MyClass` `:EndClass`	Statements which begin and end the class script for `MyClass`.
`:Field MyField` `:Field Private MyField`	A private field (can only be used by code defined in the class)
`:Field Public MyField`	A public field (visible from the outside)
`:Field … MyField←expression`	A field with a default value
`∇MyFn` `∇`	Beginning and end of a function or "method"
`:Access Public`	Declares the current function to be public
`:Access Private`	Function is private (the default)
`:Implements Constructor`	Identifies a method as the constructor, which is used to initialize the contents of a new instance
`:Include NameSpace`	Makes all functions in `NameSpace` private members of the current class, unless this is overridden by `:Access` statements in the included code
`⎕CLASS Instance`	Returns the class hierarchy for an instance.
`⎕NL`	Negative arguments return a vector of names. Further extensions to `⎕NC` and `⎕NL` will be introduced shortly.

CHAPTER 5

Properties

With a bit of luck, we are now able to find our way around simple classes and instances, and are ready to explore a few more pieces of OO functionality: A *Property* is a member which is *used* in the same way as a field, but *implemented* as a one or more functions: There is usually a pair of functions, which are commonly referred to as the *getter* and the *setter*. To explore properties, we are going to modify our `TimeSeries` class so that an instance represents a year of monthly data:

```
:Class Monthly

    :Field Public Obs      ⍝ Observed values
    :Field Public X        ⍝ X values for known Months

    :Field Public Degree←3 ⍝ 3rd degree polys by default

    ∇ r←PolyFit x;coeffs
      :Access Public
      ⍝ Use cubic fit to compute values at x

      coeffs←Obs⌹X∘.*0,⍳Degree   ⍝ Find polynomial coeffs
      r←coeffs+.×⍉x∘.*0,⍳Degree  ⍝ Compute fitted f(x)
    ∇

    ∇ make args
      :Implements Constructor
      :Access Public
      ⍝ args: Obs [X]
      args←,(⊂⍣(1=≡args))args    ⍝ Enclose if simple[5]
      Obs X←2↑args,⊂⍳⍴⊃args
      'Invalid Month Numbers' ⎕SIGNAL (∧/X∈⍳12)↓11
    ∇

:EndClass ⍝ Class Monthly
```

[5] ⍣ is the power operator, which takes a function (enclose) on the left, number of applications (1 if simple, else 0) on the right. When right operator argument is boolean, power can be read “if”, in this case “enclose *if* simple”.

Darker shading highlights the changes: We have added a new field called `X` , which will contain the `X` co-ordinates (month numbers) for our observations. The `PolyFit` function has been enhanced to use `X` rather than `(⍳⍴Obs)` in constructing the right argument to matrix division. The constructor `make` has been enhanced so that it generates `X` if it was not provided:

```
      (⎕NEW Monthly (1 4 7 9 10)).X
1 2 3 4 5

      m1←⎕NEW Monthly ((1 4 7 9 10)(1 3 5 7 9))
      m1.X
1 3 5 7 9
```

Because we think we might extend our class to handle very long timeseries at some point in the future, and since the matrix division algorithm only uses the actual observations anyway, we have decided to use a "sparse" data structure: We only store the points for which we have values.

We will experiment with defining a variety of properties which can present the sparse data in ways which may be more convenient to the user.

:Property Simple

A reporting application is likely to want to see the data as a 12-element vector, so it can be tabled with other series and easily summarized. We can support this requirement using the following property:

```
    :Property Simple FullYear
    :Access Public
        ∇ r←Get
          r←(Obs,0)[X⍳⍳12]
        ∇
    :EndProperty
```

Simple is the default type of property, so the keyword Simple is not actually required above. Although `FullYear` is implemented using a function, it looks, tastes and feels[6] like a field (or a variable) to the *user* of the class:

[6] However, if the access function crashes, it will not smell like one.

```
      ts1←⎕NEW Monthly ((1 4 2 7 3) (1 2 3 7 8))
      ts2←⎕NEW Monthly ((1 3 2 4 1) (1 3 5 7 9))
      ts1.FullYear[2] ⍝ February
4
      ↑(ts1 ts2).FullYear
1 4 2 0 0 0 7 3 0 0 0 0
1 0 3 0 2 0 4 0 1 0 0 0

      ts1.FullYear←⍳12
SYNTAX ERROR…
```

The last statement above illustrates that `FullYear` is a read only property, which is due to our not having written the corresponding `Set` function – we'll get to that in a minute. APL is (of course!) able to perform indexing on the data returned by a property, as can be seen in the expression where we extracted data for February. However, the `Get` function *did* generate all 12 elements of data, which might have been extremely inefficient if we had more data.

The `:Access Public` statement makes the property visible from outside the instance. Without this, the property could only be used by methods belonging to the class.

:Property Numbered

Numbered properties are designed to allow APL to perform selections and structural operations on the property, deferring the call to your get or set function until APL has worked out which elements actually need to be retrieved or changed:

```
    :Property Numbered ObsByNum
    :Access Public
        ∇ r←Shape
          r←12
        ∇
        ∇ r←Get args
          r←(Obs,0)[X⍳args.Indexers]
        ∇
        ∇ Set args;i
          :If (⍴X)<i←X⍳args.Indexers ⍝ New X: Add to list
              X←X,args.Indexers ⋄ Obs←Obs,args.NewValue
          :Else
              Obs[i]←args.NewValue   ⍝ Simply update
          :EndIf
        ∇
    :EndProperty
```

In order for APL to perform indexing and other structural operations, it needs to know the `Shape` of the property. With this knowledge, APL is able to perform the same

indexing and structural operations which would be allowed on the left side of the assignment arrow in a selective specification expression. The getter is called when APL encounters a function which actually needs the data in order to proceed.

To investigate how this works, edit the `Get` function for the property `ObsByNum` in the class `Monthly` from the workspace `OO4APL\Monthly`, so that it outputs the contents of `args.Indexers` to the session on each call – and experiment with different operations on the property – for example:

```
      ts1←⎕NEW Monthly ((1 4 2 7 3) (1 2 3 7 12))
      ts1.ObsByNum[2]
 Get:  2
4

      ¯2↑ts1.ObsByNum
 Get:  11
 Get:  12
0 3

      1↑⌽ts1.ObsByNum
 Get:  12
3

      ts1.ObsByNum
 Get:  1
 Get:  2
…etc… 3…11
 Get:  12
1 4 2 0 0 0 7 0 0 0 0 3
```

As can be seen above, the number of function calls is equal to the number of elements which are accessed. In a future release of Dyalog APL, it may become possible to declare the *rank* of the getter and setter, so they can work on several indices in a single call.

When APL calls a getter or setter for a numbered property, the argument is an instance of type `PropertyArguments`, which will contain the following variables:

`Indexers`	An integer vector of the same length as the rank of the property (as implied by the result of the `Shape` function).
`Name`	The name of the property, which is useful when one function is handling several properties (we'll show how to do that later).
`NewValue`	Within a setter, this is an array containing the new value for the selected element of the property.

Note that `Indexers` will always contain indices in the index origin of the *class*. This means that the access functions do *not* need to adapt to the index origin of the calling environment – APL adjusts the indices as appropriate. Conversely, the user of a class

does not need to know the index origin used within the class. This is a good example of *encapsulation* or *information hiding*, important principles of object orientation which make it easier to share and use code without needing to understand details of the implementation.

:Property Keyed

It is common for object oriented languages to allow indexing using names or other *keys*, which are not necessarily numeric indices into an array. For example, the collection of sheets in an Excel workbook can be indexed by sheet name using expressions like:

```
      XL.ActiveWorkbook.Sheets[⊂'Sheet2'].Index
```

See the chapter titled External Classes for more information about indexing OLEClient and DotNet objects. *Keyed* properties make it possible to support indexing of properties using arbitrary keys. We create vector of month names to use as keys:

```
    Months←'Jan' 'Feb' 'Mar' 'Apr' 'May' … 'Nov' 'Dec'
```

… or alternatively, if we have the Microsoft .NET framework available on our machine, we can extract the names from the environment:

```
    ⎕using←'System.Globalization' ⍝ DotNet "Globalization
Namespace"
    Months←12↑(DateTimeFormatInfo.New
⍬).AbbreviatedMonthNames
```

This will define a 12-element vector[7] of 3-letter month names in the local language. We can now define a property to do indexing using local month names:

```
:Property Keyed ObsByName
:Access Public
    ∇ r←get args;i
      ⎕SIGNAL((⍴Months)∨.<i←Months⍳1⊃args.Indexers)/3
      r←(Obs,0)[X⍳i]
    ∇
    ∇ Set args;i;m
      ⎕SIGNAL((⍴Months)∨.<i←Months⍳1⊃args.Indexers)/3
      m←~i∈X ⍝ New months?
      X←X,m/i ⋄ Obs←Obs,m/args.NewValue ⍝ Add new months
      Obs[X⍳i]←args.NewValue
    ∇
:EndProperty
```

[7] DateTimeInfo.AbbreviatedMonthNames has 13 elements, according to a source on the Microsoft CLR team this is in order to "discourage developers from making assumptions about the length of calendars".

The `ObsByName` property can be used as follows:

```
      ts1.ObsByName[⊂'Jan']
1
      ts1.ObsByName['Oct' 'Nov' 'Dec']←4 5 6
```

The entire index and array of new values (when provided) is passed in a single call to a *keyed* getter or setter. APL will verify that the shape of the sub-array identified by the `Indexers` conforms to the shape of the result of get, and in the case of set, the new values. This is why `'Jan'` must be enclosed in the first example above.

Unlike a field, a property does not map directly to an underlying array. The property provides a "virtual" array, the elements of which can be generated on demand – and created when updated. As an extreme example of this, the following class presents the APL interpreter as an infinitely large keyed property, with all possible APL expressions as keys, and the results of the expressions as the corresponding values:

```
:Class APL
    :Property Keyed ValueOf
    :Access Public
        ∇ r←Get args
          r←⍎¨1⊃args.Indexers
        ∇
    :EndProperty
:EndClass ⍝ APL
```

Allowing:

```
      iAPL←⎕NEW APL
      iAPL.ValueOf['2+2' '+/⍳5']
 4 15
```

Numeric Keyed Properties

Note that you can use *keyed* properties with integer indices, and use them in the same way as in a *numbered* property. You might do this if you were worried that a numbered property would be inefficient due to the number of calls to your access functions, or if a simple property would cause too much data to be generated, The drawback of this approach is that the only direct selection operation which is possible on a keyed property is indexing, and – since APL will be passing the indices unchanged - the indices will have to be provided in the index origin of the class.

For readers familiar with the concept of function rank, it may be helpful to think of a numbered access function as having rank zero, while a keyed access function has infinite rank. Future versions of Dyalog APL may allow you to control the rank of your access functions.

Default Properties

The `Default` option identifies a property as the *default property* for a class. If a class has a default property, square bracket indexing can be applied directly to a reference to the instance. If we edit the definition of the `ObsByNum` property and identify it as `:Property Default Numbered ObsByNum`, we can apply indexing as follows:

```
      ts1←⎕NEW Monthly ((1 4 2 7 3) (1 2 3 7 8))

      ts1[2]          ⍝ Shorthand for the following
4
      ts1.ObsByNum[2] ⍝ Equivalent to the above
4

      ⍴ts1            ⍝ But: ts1 is a scalar

      2↑ts1
LENGTH ERROR
```

The last two expressions expose a problem with indexing default properties: An instance (or rather: a reference to an instance) *is* a scalar. The APL standard allows square bracket indexing on instance as a *conforming* extension, since indexing the instance would give a `RANK ERROR`. If you feel uncomfortable about indexing a scalar, reference the default property explicitly.

```
      ts1.ObsByNum[2]
```

Although indexing can be applied directly to an instance, all other selection or structural operations must view the instance as a scalar. For example, the expression `(2↑ts1)` should return `ts1` followed by a prototypical instance of the `Monthly` class, but fails because none has been defined (we'll look at how to define class prototypes when we take a closer look at *constructors*).

Dyalog APL also includes the *squad indexing* primitive `⌷`, a dyadic primitive function which provides a functional alternative to square bracket indexing:

```
      4⌷'ABCD'               ⍝ 'ABCD'[4]
D
      2 (1 2)⌷2 2⍴'ABCD'     ⍝ (2 2⍴'ABCD')[2;1 2]
CD
```

Monadic `⌷` returns the entire right argument, as if the elided left argument had selected everything along all dimensions. If the right argument is an instance, monadic `⌷` returns the entire contents of the default property.

```
      ⍴⍬ts1
12
      2↑⍬ts1
1 4
      ⍬ts1
1 4 2 0 0 0 7 3 0 0 0 0
```

Note that the 2nd expression above only called the numbered getter twice, once for each element of the complete selection operation `2↑⍬`. If we had picked `ObsByName` as the default property, we would be able to index the instance using month names:

```
      ts1←⎕NEW Monthly ((1 4 2 7 3) (1 2 3 7 8))
      ts1['Jan' 'Jul']
1 7
```

(Obviously), there can only be one default property.

Note: The workspace `MonthlyAfter5` in the `OO4APL` folder contains a version of the Monthly class using all of the enhancements discussed in this chapter.

Triggers

If you want the "lightweight" characteristics of a *field*, where APL handles data access directly without your having to write access functions, but you would like to be able to react to changes to the field, you can use a *trigger*. A trigger is a function which is called when one or more fields (or other variables) are modified. For example, we could update our `Monthly` class so that it calculates the coefficients when one of the variables involved in the calculations change, rather than doing it on every call to `PolyFit`. `PolyFit` can be modified to use the pre-calculated coefficients:

```
    ∇ PreCalc args
      :Implements Trigger X,Obs,Degree
      ⎕←'PreCalc: ',args.Name
      :If OK←((⍴Obs)=⍴X)∧Degree<⍴Obs
          Coeffs←Obs⌹X∘.*0,⍳Degree ⍝ Find polynomial coeffs
      :EndIf
    ∇

    ∇ r←PolyFit x;coeffs
      :Access Public
      ⍝ Use polynomial to compute values at x

      'Unable to Fit Polynomial' ⎕SIGNAL OK↓5
      r←Coeffs+.×⍉x∘.*0,⍳Degree  ⍝ Compute fitted f(x)
    ∇
```

`PreCalc` writes to the session when it is called, so we can keep an eye on things:

```
      ts1←⎕NEW Monthly ((1 4 2 7 3) (1 2 3 7 8))
PreCalc: Obs
PreCalc: X
      1⍕ts1.PolyFit 1 2 3 4
 1.7 2.0 3.5 5.3

      ts1.Degree←2
PreCalc: Degree
      1⍕ts1.PolyFit 1 2 3 4
 1.0 2.7 4.0 4.8

      ts1.X←1 2 3 7
Precalc: X
      1⍕ts1.PolyFit 1 2 3 4
Unable to Fit Polynomial
```

The trigger mechanism is not a purely object oriented feature. It can be used to track any variable in a workspace. For example:

```
∇ foo args
  :Implements Trigger A
  'A ←'A'at',1↓⎕SI{α,'[',(⍕ω),']'}¨⎕LC
∇
∇ goo
  A←2
  A←'Hello'
∇
```

If we now run `goo`, we will see the following output:

```
A ←  2  at goo[2]
A ←  Hello  at
```

Warning: Applications should not depend on the exact timing of, or the number or calls to, a trigger, beyond relying on each trigger function to be called at least once following the completion of the primitive function (usually assignment) which set the trigger variable. As can be seen in the above example, the trigger functions were called *after* the completion of the entire line of code containing the trigger event. In the second case, `goo` was no longer on the stack at the time when `foo` was called.

CHAPTER 6

Constructors and Destructors

As we have seen, a *constructor* is a function which is called to initialize the contents of an instance, immediately *after* the instance has been created – but *before* it can be used. In the examples we have seen so far, we have used it to:

Set an initial seed in our `Random` number generator class
Initialise data for a new instances of `Monthly` and `TimeSeries`

A constructor is used when it makes sense to provide initial values for fields, or allocate external resources associated with the object. For example, a constructor might open a component file, or connect to a database and execute a query.

A *destructor* is called when the instance is about to disappear. In a destructor, you can close files and database connections, or free up any other resources which will no longer be required.

The following is a simple class which provides access to Excel Workbooks, using Excel as an OLE Server:

```
:Class ExcelWorkBook

    :Field Public Application
    :Field Public Workbook
    :Field Public Sheets
    :Field Private Opened←1 ⍝ Did we open it?

    ∇ make book;i;books
      :Implements Constructor
      Application←⎕NEW
          'OleClient' (⊂'ClassName' 'Excel.Application')

      :If 0≠Application.Workbooks.Count ⍝ Open books
          books←⎕Application.Workbooks
      :AndIf (⍴books)≥i←books.FullName⍳⊂book ⍝ Open?
          Workbook←i⊃books
          Opened←0 ⍝ No we did not
      :Else
          :If book∧.=' ' ⍝ No book named => Create one
              Workbook←Application.Workbooks.Add ⍬
          :Else          ⍝ Open the requested book
              Workbook←Application.Workbooks.Open ⊂book
          :EndIf
      :EndIf

      Sheets←⎕Workbook.Sheets
    ∇

    ∇ close
      :Implements Destructor
      :If Opened  ⍝ Close it if we opened it
          :Trap 0 ⍝ If workbook somehow damaged
              ⎕←'Closed workbook ',Workbook.FullName
              Workbook.Saved←1
              Workbook.Close ⍬
          :EndTrap
      :EndIf
    ∇
:EndClass ⍝ Class ExcelWorkBook
```

The constructor takes the name of a workbook as its argument, and ensures that the book is open. If the name is blank, a new workbook is created. The three public fields of this class are:

`Application`	A handle to Excel.Application, so we can do things like make Excel visible and tell it not to issue prompts.
`Workbook`	Handle to the Excel workbook instance corresponding to our workbook.
`Sheets`	A vector containing all the sheets in the book, so we don't have to repeat `⎕Workbook.Sheets`

The following illustrates the use of the first two fields.

```
      wb1←⎕NEW ExcelWorkBook 'c:\temp\book1.xls'
      wb1.Application.Visible←1
      wb1.Workbook.FullName
C:\temp\Book1.xls
```

The following examples show how we can read data from the sheets, make a change, save the workbook, and close the object:

```
      wb1.Sheets.Name
Sheet1  Sheet2  Sheet3

      wb1.Sheets[1].(Name UsedRange.Value2)
 Sheet1  1  2  3  4  5  6  7  8
         9 10 11 12 13 14 15 16

      wb1.Sheets[1].Range[⊂'A1:B2'].Value2←2 2⍴'Kilroy'
'was' 'here' '!'

      wb1.Workbook.SaveAs⊂'c:\temp\book2'

      ⎕ex 'wb1'
Closed workbook: C:\temp\Book1.xls
```

Note that the destructor is called when the *last* reference to an instance disappears. If you expunge a name and the destructor is not called as you expected, look for a leaked local variable or a temporary global created by hand, containing a reference that you had forgotten about. The system function `⎕INSTANCES` will return a list of refs to existing instances:

```
      wb1←⎕NEW ExcelWorkBook 'c:\temp\book1.xls'
      wb2←⎕NEW ExcelWorkBook 'c:\temp\book2.xls'
      books←wb1 wb2

      ⎕instances ExcelWorkBook
 #.[ExcelWorkBook]  #.[ExcelWorkBook]
      (⎕instances ExcelWorkBook).Workbook.Name
 Book1.xls  Book2.xls

      )erase wb1 wb2
      ⍴⎕instances ExcelWorkBook
2
```

There are still 2 instances.

Display Form

The function ⎕DF (Display Form) can be used to set the "display form", of an instance (or namespace) – which defines the result returned by monadic format (⍕) of the instance. By default, the display form shows the parent space and the name of the class, as in:

```
#.[Monthly]
```

This tells us that we have an instance of `Monthly` which was created in the root (`#`). ⎕DF is often used in a constructor to set the display form to something which helps to identify the particular instance. For example, if we added the following line to the constructor of `Monthly`:

```
      ⎕DF ⍕,[1.5]'[Timeseries]' (Months[X],'=',[1.5]Obs)
```

Then the display form would be something like:

```
      ⎕←ts1←⎕NEW Monthly ((1 4 2 7 3) (1 2 3 7 8))
 [Timeseries]
  Jan = 1
  Feb = 4
  Mar = 2
  Jul = 7
  Aug = 3
```

We could also update the trigger function to keep the display form up-to-date.

Niladic Constructors and the Class Prototype

All the constructor methods we have written so far have taken an argument which has been used to initialise each new instance. If the constructor is a niladic function, this means that the class does not need any instantiation parameters. It is possible for a Dyalog APL class to have several constructors: One or more which take a right argument, and one (or zero) which does not. A constructor with no argument is known as the *default constructor*. If `⎕NEW` is called with no parameters following the class, the default constructor will be called. For example, we could have defined a niladic constructor for the `Random` class, as an alternative to testing whether the argument is zero in the monadic constructor:

```
∇ make args
  :Implements Constructor
  :Access Public
  InitialSeed←args
  Reset
∇

∇ make0
  :Implements Constructor
  :Access Public
  Reset
∇
```

With the above constructors and the `RL` function from the beginning of Chapter 4, the `Random` class could be used as follows:

```
      r1←⎕NEW Random ⍝ Nothing strange about this…
Random Generator Initialised, Seed is 42

      r1.RL 1234   ⍝ Change ⎕RL to 1234 (returns old value)
42
      (3↑r1).RL 0  ⍝ Two default instances created
Random Generator Initialised, Seed is 42
Random Generator Initialised, Seed is 42
1234 42 42

      r0←0/r1      ⍝ APL remembers the Class

      (1↑r0).RL 0  ⍝ 1↑ causes creation of a new prototype
Random Generator Initialised, Seed is 42
42

      1↑r0.RL 0
Random Generator Initialised, Seed is 42
0
```

In the last example, APL determines the type of the result by creating a prototype, calling the function `RL` in it, and doing a `0⍴` of that result (`42`). `1↑` on this gives the result.

CHAPTER 7

Shared Members

Almost all the members we have used up to this point have been *instance* members. An *instance field* has a distinct value for each instance. The code in *instance methods*, or associated with *instance properties,* generally refer to instance data. *Shared* fields have the same value in *all* instances. Shared methods and property accessor functions access shared data.

The meaning of the terms *public* and *private* is the same for shared members as it is for instance members. Public members are visible from outside, private members can only be seen by code which is defined and running inside the class.

The `Months` field, which contains abbreviated month names within the `Monthly` class, is an example of a field which might as well be shared – it is going to be the same in all instances. We can add:

```
:Field Shared Public Months
```

… to the beginning of our `Monthly` class, and subsequently verify that the property is in fact shared:

```
      ts1←⎕NEW Monthly ((1 2 3 4)(5 6 7 8))
      ts2←⎕NEW Monthly ((3 4 5 6)(5 6 7 8))
      ts1.Months[3]←⊂'Mch'

      ts2.Months
 Jan  Feb  Mch  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec

      )erase ts1 ts2
      Monthly.Months
 Jan  Feb  Mch  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
```

As the last expression illustrates, public shared properties can be referenced through the class itself, without using an instance. In this case, it is probably a good idea to declare the field as “read only”:

```
    :Field Public Shared ReadOnly Months←12↑
          (DateTimeFormatInfo.New 0).AbbreviatedMonthNames
```

Most of the time, shared fields are likely to be either read only or private - or hidden behind some kind of property with careful validation in the setter.

Shared members are available to code which defines instance methods and properties. As we have seen above, they can be used as if they were members of all instances. On the other hand, instance data is *not* visible to shared code, since this would presume the selection of a particular instance. Note that code which is executed when the a class script is fixed, is shared code, executing within the class. Thus, it will not be able to see the value of instance fields, except to define default values for instance fields in :Field statements.

Shared Methods

Shared methods will either be access methods which are used to manipulated shared data, or methods which provide a service which is related to the class but does not require or apply to an instance. For example, if we extend our `ExcelWorkBook` class with a method which lists the workbooks in a folder, we could provide this as a shared method:

```
    ∇ r←List folder;⎕USING
      :Access Public Shared

      ⎕USING←'' ⍝ DotNet Search Path
      :Trap 0
        r←((System.IO.DirectoryInfo.New⊂folder).
          GetFiles⊂'*.xls').Name
      :Else ⋄ r←⍬
      :EndTrap
    ∇
```

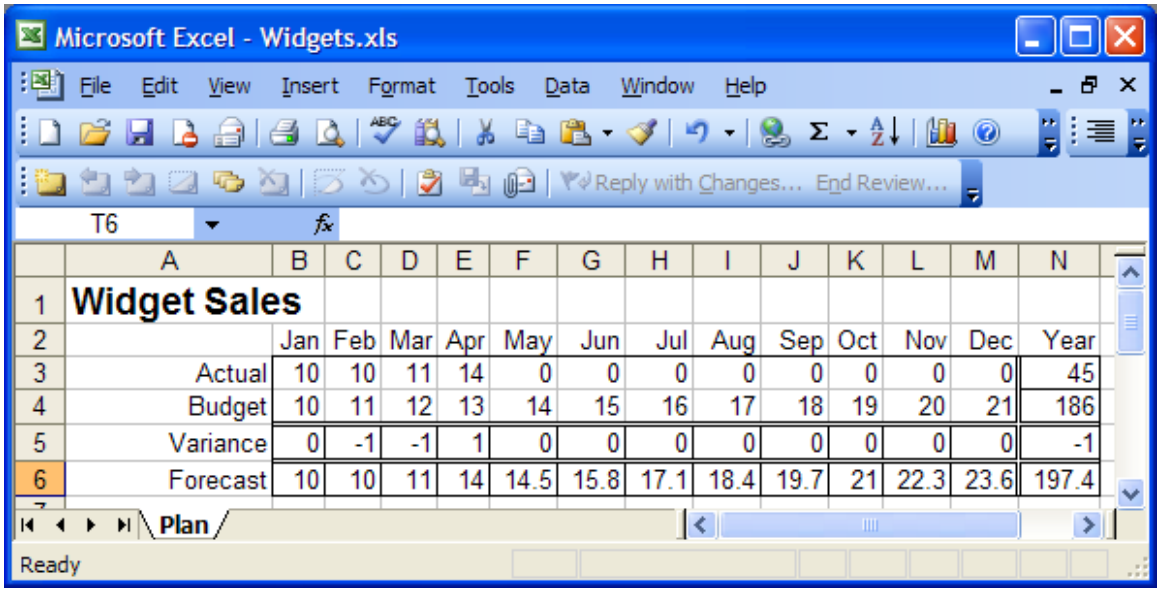
This would allow us to find out which workbooks there are, which is information we might need *before* we open one:

```
      ExcelWorkBook.List 'c:\temp'
Book1.xls Book2.xls  Book3.xls
```

CHAPTER 8

Inheritance

It is possible to define a class which *extends* – or *inherits* the features of – an existing class. *Inheritance* is considered to be one of the most important features of Object Orientation, because inheritance provides a well-defined mechanism for sharing code.



Imagine that we would like to create a simple budgeting and forecasting application which uses Excel spreadsheets like the above for data entry.

Departmental managers will initially enter budget and subsequently actual data and send their workbooks to us at regular intervals for reporting and consolidation. Our plan is to read data from workbooks in the above format, and use code written in APL to calculate the variance to budget for those months where both actual and budget data is available and finally produce a Forecast. Something along the lines of:

```
    ∇ Calculate;real;ts
      :Access Public
      real←+/∨\⌽Actual≠0          ⍝ # of months of real data
      Variance←12↑real↑Actual-Budget ⍝ Where data exists

      :If real>0
          ts←⎕NEW #.Monthly(real↑Actual)
          ts.Degree←1 ⍝ 3rd degree is a bit too exciting
          Forecast←(real↑Actual),real↓ts.PolyFit⍳12
      :EndIf
    ∇
```

We will call this special type of workbook a “PlanBook” (there is an example file called widgets.xls in the `OO4APL` folder). The `PlanBook` class will extend `ExcelWorkBook`, providing additional properties called `Actual`, `Budget`, `Variance` and `Forecast`, which access data via the `Sheets` field which `ExcelWorkBook` makes available.

The constructor in our new `PlanBook` class will examine the contents of the workbook which has been opened and look for signs that it is a well-formed PlanBook. We might start with a class definition like the following:

```
:Class PlanBook : ExcelWorkBook

    :Field Private RowNames←'Actual' 'Budget'
                            'Variance' 'Forecast'
    :Field Private Instance DataRange
                         ⍝ Will point to Excel Data Range

    ∇ make book;z
      :Access Public Instance
      :Implements Constructor :Base book

      :If Sheets.Name≡,⊂'Plan'
      :AndIf 6 14∧.≤⍴z←Sheets[1].UsedRange.Value2
      :AndIf ((⍴RowNames)↑2↓z[;1])≡RowNames
          DataRange←Sheets[1].Range[⊂'B3:M6']
      :Else
          (book,' is not a valid Plan Workbook')⎕SIGNAL 11
      :EndIf
    ∇

:EndClass ⍝ Class Plan
```

A derived class is declared by following the class name with a colon and the name of the class which we wish to extend. The `:Base` keyword in the `:Implements Constructor` causes a call to the base constructor, using the result of the expression following `:Base` as the argument. In this case, the argument to the `PlanBook` constructor, which will contain the name of the workbook, is passed unmodified to the `ExcelWorkBook` constructor.

Once the workbook is open, `make` takes a look at the `Sheets` member, which we have *inherited* from `ExcelWorkBook`, to verify that the workbook has a single sheet named "Plan", that this sheet has at least 6 rows and 14 columns, and that the second column contains the four names which we expect. If all is well, we create a private field called `DataRange`, which gives us easy access to the 4x12 range of cells containing the data.

If the workbook does not look right, we signal an error, which will be reported by `⎕NEW` and prevent the instance from being completely created. In this case, the base destructor is called as the instance disappears and the workbook is closed again.

```
      w←⎕NEW PlanBook'c:\temp\book1.xls'
c:\temp\book1.xls is not a valid Plan Workbook
      w←⎕NEW PlanBook'c:\temp\book1.xls'
     ^
Closed workbook C:\temp\Book1.xls

      w←⎕NEW PlanBook'c:\temp\widgets.xls'
      w.Sheets.Name
 Plan
```

All that remains to make it possible for us to write `Calculate` in the way we envisaged, is to provide the four "timeseries" as simple properties:

```
    :Property Actual,Budget,Variance,Forecast
    :Access Public
        ∇ r←get args
          r←DataRange.Value2[RowNames⍳⊂args.Name;]
        ∇
        ∇ set args
          DataRange.Value2[RowNames⍳⊂args.Name;]←
                                              args.NewValue
        ∇
    :EndProperty
```

If we add the `Calculate` method and `)COPY` the `Monthly` class we can now open the PlanBook and work with it:

```
      w←⎕NEW PlanBook'c:\temp\widgets.xls'
      w.(↑Actual Budget)
10 10 11 14  0  0  0  0  0  0  0  0
10 11 12 13 14 15 16 17 18 19 20 21

      w.Calculate
      ↑w.(Variance Forecast)
 0 ¯1 ¯1  1  0    0    0    0    0    0  0    0
10 10 11 14 14.5 15.8 17.1 18.4 19.7 21 22.3 23.6
```

And the spreadsheet has been updated. We can now:

```
      w.Workbook.Save
      )erase w
Closed workbook C:\docs\sales\Widgets.xls
```

And our work is done.

Inherited Members

We have seen that the `Sheets` member, inherited from `ExcelWorkBook`, is available as a public field of instances of `PlanBook`. Public methods are (of course) also inherited by the derived class:

```
      w.List 'c:\docs\sales'
 Book1.xls  Book2.xls  Widgets.xls
```

However – it might be more useful to have a more specific version of `List`, which only lists valid `PlanBook`s. We can use the base method to get the list of workbooks, and then validate them by opening each one as a `PlanBook`:

```
    ∇ r←List folder;m;i;z
      :Access Shared Public
      ⍝ Extend ExcelWorkBook.List to list only PlanBooks

      r←⎕BASE.List folder
      m←(⍴r)⍴1
      :For i :In ⍳⍴r
          :Trap 0 ⋄ z←⎕NEW PlanBook(folder,'\',i⊃r)
          :Else ⋄ (i⊃m)←0 ⋄ :EndTrap
      :EndFor
      r←m/r
    ∇
```

Which allows:

```
      ExcelWorkBook.List 'c:\temp'
 Book1.xls Book2.xls Widgets.xls

      PlanBook.List 'c:\temp'
Closed workbook C:\temp\Book1.xls
Closed workbook C:\temp\Book2.xls
Closed workbook C:\temp\Widgets.xls
 Widgets.xls
```

It is possible to access the `ExcelWorkBook` version of a method via an instance of `PlanBook`, by using a technique called *casting*. Dyadic `⎕CLASS` allows us to access members of the base class by providing a view of the instance as if it were an instance of that class. You can only *cast* to a class which is in the class hierarchy for the instance:

```
      (ExcelWorkBook ⎕CLASS w).List 'c:\temp'
 Book1.xls Book2.xls Widgets.xls
```

Public members of the base class become public members of the derived class, and are (of course) also accessible to the code in the derived class. We have to refer to the base class version of `List` as `⎕BASE.List` because the derived class has defined its own version of `List`. Note that `⎕BASE` is "special syntax" which searches the class hierarchy for a particular member, and can only be used if it is followed by a dot and a base member name. `⎕BASE` is *not* a function which can be used to return a reference to a base class.

Private members of the base class remain hidden from the outside. As any other user of `ExcelWorkBook`, the implementor of `PlanBook` is insulated from private changes to the implementation of the base class – which helps make it possible for both implementors to get a good nights' sleep. Which is even more important if you are the implementor of both.

Dyalog APL does not implement *protected* members, which are a kind of halfway house; visible to code in derived classes, but invisible from the outside.

Benefits of Inheritance

If we needed to process a new type of workbook, we could create a second (third and fourth) class deriving from `ExcelWorkBook`. The benefits of working this way are:

Easy reuse of the work done in writing the base class, without duplicating code.
Classes are easier to learn to use, due to shared features. For example, the `List` method is available in all classes which derive from `ExcelWorkBook` (unless they redefine it).

The rules of class inheritance guarantee that, so long as the behaviour of public members is not changed, future enhancements to the base class will be immediately available to the derived classes, unless they *decide* to implement different behaviour.

The rules of class inheritance minimize the burden of maintenance, training and documentation – so long as we adhere to some simple rules. Of course, there *are* a few rocks on the road. For one thing, the very attraction of the above benefits can quickly lead to unnecessarily general base classes with a huge collection of members, which can end up being both inefficient and difficult to learn. Huge quantities of documentation are not necessarily a good thing if only 5% of it is relevant to the job at hand.

As requirements change, deciding where in the hierarchy to add new functionality may require much thought. The bad news is that the design of complex systems is always going to require insight into the problems which need to be solved, today and in the future – regardless of the technology or paradigm used. The good news is that OO provides excellent tools both for the redesign and reimplementation of, and easy migration to, a new set of classes. Given that the public interface to a class is so well defined, it is possible to rewrite the implementation at any level in the hierarchy, but keep the old interfaces available as an alternative for those application components which cannot easily be rewritten.

Inheriting from Several Classes

In our `PlanBook` example, although we focused on extending the `ExcelWorkBook` class, we also used functionality from the `TimeSeries` class. In this case, the choice to make `PlanBook` an extension of `ExcelWorkBook` was fairly easy – but in many other situations, it can be difficult to decide which class to extend. Dyalog APL has followed C# in only allowing you to derive from *one* other class. Some OO platforms allow *multiple* inheritance, with rules for how name conflicts are resolved and constructors and destructors cascade. Unfortunately, while all the OO features we have discussed so far in this guide, including *single* inheritance, are supported in much the same manner in all OO systems, there is less agreement on the details of multiple inheritance. This is one of the main reasons why C# has avoided it, and we have followed suit.

Given the way classes work, it may be easier to work around not having multiple inheritance than it would be to understand any particular flavour of it. We had no great difficulty in using the `TimeSeries` class from within `PlanBook`. Even if inheritance was not available at all, we would have been able to provide the important functionality of `PlanBook` quite easily:

We would need to have additional field declarations in PlanBook:

```
      :Field Public WorkBook
      :Field Public Sheets
```

The statement:

```
      :Implements Constructor :Base book
```

Would have to be replaced with:

```
      :Implements Constructor
      WorkBook←⎕NEW #.ExcelWorkBook book
      Sheets←WorkBook.Sheets
```

In the `List` method, we would have to replace:

```
      r←⎕BASE.List folder
```

By:

```
      r←ExcelWorkBook.List folder
```

The big difference between the result of doing this and the original `PlanBook` is that the other public members of `ExcelWorkBook` would not be exposed. Future extensions to `ExcelWorkBook` would not be immediately available to users of `PlanBook`. Whether this is a drawback or a simplification depends on your requirements. In the long term, you might be better off with a class which only exposed the specific `PlanBook` functionality. If you subsequently *did* decide to expose ExcelWorkBook functionality, all you really need to do is to expose the new `WorkBook` variable as a public field, which would allow:

```
      w←⎕NEW PlanBook2 'c:\temp\widgets.xls'

      w.WorkBook.List 'c:\temp'
 Book1.xls  book2.xls  widgets.xls

      w.WorkBook.Application.Version ⍝ Excel Version Number
11.0
```

If we add the `List` method from `PlanBook` to `PlanBook2`, then `w.List` would call the `PlanBook` version of `List`, and `w.WorkBook.List` would allow access to the `ExcelWorkBook` version, so both would be available.

Code Reuse with :Include

If you have a set of utility functions or methods which you wish to include in a number of classes, but some of these classes are already derived classes, `:Include` provides another alternative to multiple inheritance. For example, if we decide that the `List` and `Delete` methods of our `ExcelWorkBook` are generally useful, we can create a namespace called `ExcelTools` and define the functions in it:

```
      )cs ExcelTools
      ⎕using←''

     ∇ Delete file
[1]    :Access Public Shared
[2]    (System.IO.FileInfo.New file).Delete
     ∇

     ∇ r←List folder
[1]    :Access Public Shared
[2]
[3]    :Trap 0
[4]        r←((System.IO.DirectoryInfo.New folder).
                                    GetFiles'*.xls').Name
[5]    :Else ⋄ r←⍬
[6]    :EndTrap
     ∇
```

If we remove List from `PlanBook2` and add the line:

```
:Include ExcelTools
```

This will import the functions in ExcelTools, without using inheritance. The difference between this and using inheritance is that the :Included functions actually become methods of the class they are imported to, so they can reference members of this class if required – which would not be the case if they were inherited methods of a base class.

Whether you use “real” inheritance or :Include, the source code is *not* copied, so if you trace into and modify a method, the change will be made in the original namespace or class, and any instance or class which currently contains it will immediately see the new version.

Summary of Chapters 5-8

Features of Dyalog APL which we have introduced while discussing Properties, Constructors, Destructors, Shared members – and Inheritance.

`:Implements Trigger field`	The function will be called shortly after field is modifiied.
`:Implements Constructor`	The function is used to initialize a new instance of the class.
`:Implements Destructor`	Function used to clean up or de-allocate resources an instant before the instance disappears.
`⎕INSTANCES`	Return instances of [or derived from] class
`⎕DF`	Set Display Form
`index⌷array`	Squad indexing
`⌷instance`	Return all of the default property
`:Property [Simple]` `:EndProperty`	A property where getters and setters pass the entire array at once.
`:Property Numbered` `:EndProperty`	Getter and setter work on a single item of the property at a time, identified by numeric indices along each dimension. Requires `Shape` function.
`:Property Keyed` `:EndProperty`	Entire Indexer is passed in a single call to getter and setter. Only square bracket indexing is permitted.
`:Property Default` `:EndProperty`	A property which is referenced if square bracket indexing is performed directly on the instance.

`:Field … Shared …`
`:Property … Shared …`
`:Access … Shared …`
	Fields, Properties and Methods which are shared by all instances, and accessible through the class.

`:Field … ReadOnly`	A read only field.
`:Base …`	Following `:Implements Constructor`, specifies a call to the base constructor.
`⎕BASE.member`	A reference to a public member in the closest class in the hierarchy which exposes it
`:Class name : baseclass`	Class `name` which *derives from* or *extends* `baseclass`.

CHAPTER 9

Deriving from Dyalog GUI Classes

In addition to the classes which you can write in APL, you can work with:

- Dyalog GUI classes, including …
- OLE Servers and and OLE Controls (ActiveX components)
- Microsoft .NET classes

These classes can be used in exactly the same way as instances of classes which have been implemented in APL. You create instances of them with ⎕NEW, and you can manipulate the public properties, methods and events which they expose (but unlike APL classes, you cannot see or modify the implementation). For example:

```
      MyForm←⎕NEW 'Form' (('Caption' 'Hello World')
                                          ('Size'(10 10)))

      XL←⎕NEW 'OleClient' (⊂'ClassName' 'Excel.Application')
      XL.ActiveWorkbook.Sheets[⊂'Sheet2'].Index
2

      ⎕using←'' ⍝ Set the .NET search path
      now←System.DateTime.Parse ⊂'2006-06-06 06:06:06.666'[8]
      ∪now.(Year Month Day Hour Minute Second Millisecond)
2006 6 666
```

Note that the class *name* for built-in GUI classes is provided as a string rather than as a reference to a name in the workspace.

[8] If you are reading this, the world apparently did not come to an end as the clock ticked past 6/6/6.

It is not possible to create instances of OLEControl objects using ⎕NEW. (or classes which derive from OLEControls). This limitation may be lifted in a subsequent release.

A Standard Dialog Box

Dyalog GUI classes provide basic building blocks for GUI applications. Each class, be it a Form, Button, List- or Combo-box, Edit, Grid, Treeview (etc), is a very general component with a large number of properties which can be used to provide a variety of different behaviours depending on your application requirements. In any given application, you are likely to use certain behavioural patterns frequently. If you write an APL class which uses a GUI class as its base class, you can create "custom controls" which are easier to use in your particular application context, than the original Dyalog classes.

Imagine that your company has a corporate standard for the appearance of dialog boxes:

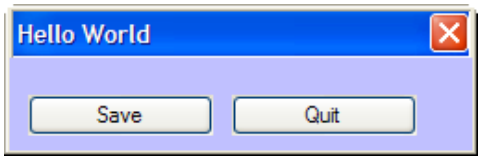
- The form will have standard dialog box appearance (EdgeStyle Dialog, Border 2)
- Each form will have a "Quit" and a "Save" button in the bottom left corner ("attached" to corner if the form is resized)
- The form cannot be closed except by clicking on one of these buttons
- The form Caption and its Size must always be provided when an instance is created.

We can create a suitable class which derives from `'Form'`: (you can find this class in the workspace `DerivedGui` in the `OO4APL` folder):

```
:Class Dialog : 'Form'
⍝ Implement Company Standard Dialog Form
⍝ Has 'Save' and 'Quit' buttons at bottom left
⍝ Coord Pixel, Cannot be Closed except thru Quit or Save
⍝ Usage: ⎕NEW Dialog (Caption Size FormProps)
⍝ Set Save.onSelect to control Save button behaviour

    :Field Public Save
    :Field Public Quit

    ∇ Create(cap size formprops);z
      :Access Public
      z←('Coord' 'Pixel')('EdgeStyle' 'Dialog')('Border' 2)
      z←z,('Caption'cap)('Size'size),formprops
      :Implements Constructor :Base z

      onClose←¯1 ⍝ Disable Close event
      z←('Size'(22 100))('Attach'(4⍴'Bottom' 'Left'))
      Save←⎕NEW'Button'(('Caption' 'Save')
                        ('Posn'((Size[1]-30),10)),z)
      Quit←⎕NEW'Button'(('Caption' 'Quit')
                        ('Posn'((Size[1]-30),120)),z)
      Quit.onSelect←'doQuit'
    ∇

    ∇ doQuit args
      :Access Public
      Close
    ∇

:EndClass ⍝ Dialog
```

Which allows:

```
      f1←⎕NEW Dialog ('Hello World' (50 250) 0)
      f1.BCol←192 192 255 ⍝ Pale Blue
```



Note that the name of the GUI class is quoted when it is used as a base class, in the same way as it would be in the argument to ⎕NEW. Apart from that, all the principles we have discussed so far about the use of the public members of the base class apply in the same way as if the base class had been written in APL. For example, we can set the BCol property to change the background colour.

In the class code, note that we can refer to the onClose property of the form directly in our constructor: It is a public property exposed by our base class and thus immediately available to derived class code – and to any user of the derived class. Our doQuit function can call the Close method of the form directly to close the form.

With the above class in our arsenal, we can start each dialog from a slightly higher level than a raw Dyalog Form object.

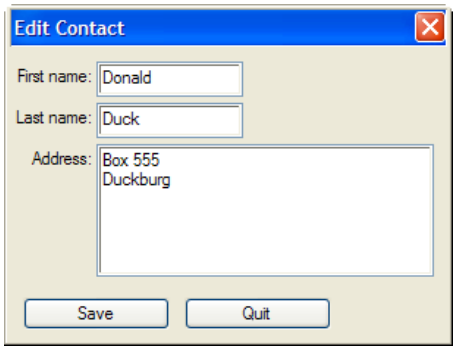
A Labelled Edit Field

One pattern which is often repeated in applications is an edit field with an attached label. It would be nice for application code to be able to treat such a pair as a unit, rather than have to position, size and track them separately. We can achieve this with a class which derives from the built-in Edit class and adds some right-justified text just to the left of the edit box:

```
:Class EditField : 'Edit'
⍝ An Edit field with an associated Label on the left
⍝ Usage example: ⎕NEW EditField
  ('Price:' '10.5' (10 30) (⍬ 50) (⊂'FieldType' 'Numeric'))

    :Field Private Label ⍝ Ref to a Text object

    ∇ Make(label text posn size editprops)
      :Access Public
      :Implements Constructor :Base ('Text' text)
                      ('Posn' posn)('Size' size),editprops
      Label←##.⎕NEW'Text'(('Text' label)
           ('Points'(posn+3 ¯3)) ('Halign' 2))
           ⍝ Created in container (##)
    ∇
:EndClass ⍝ EditField
```

A more sophisticated implementation might have options for the positioning and aligment of the label, but this class illustrates the principle. Creating “company standard” dialog boxes for data entry is now a bit easier than before:

```
      f1←⎕NEW Dialog ('Edit Contact' (200 300) 0)
      f1.First←f1.⎕NEW EditField
                      ('First name:' '' (10 60) (0 100) 0)
      f1.Last←f1.⎕NEW EditField
                       ('Last name:' '' (38 60) (0 100) 0)
      f1.Address←f1.⎕NEW EditField
        ('Address:' '' (66 60) (90 230) (⊂'Style' 'Multi'))
      f1.(First Last Address).Text←'Donald' 'Duck'
                                    ('Box 555' 'Duckburg')
```



Our `Dialog` class exposes the Save button as a public field, so we can collect the data by defining a function and assigning it to the `Select` event:

```
      ∇ Update(button event);form
[1]     form←button.## ⍝ Find the form
[2]     ContactInfo←form.(First Last Address).Text
[3]     form.Close
      ∇

      f1.Save.onSelect←'Update'
```

Finally, we could collect our notes, tidy up and put the whole thing up into a little `Contact` class with an `Edit` method:

```
:Class Contact
    :Field Public FirstName←''
    :Field Public LastName←''
    :Field Public Address←0⍴⊂''

    ∇ Edit;f1
      ⍝ Uses #.Dialog and #.EditField
      :Access Public
      f1←⎕NEW #.Dialog('Edit Contact'(200 300)⍬)
      f1.First←f1.⎕NEW #.EditField
                          ('First name:' ''(10 60)(⍬ 100)⍬)
      f1.Last←f1.⎕NEW #.EditField
                          ('Last name:' ''(38 60)(⍬ 100)⍬)
      f1.Address←f1.⎕NEW #.EditField
         ('Address:' ''(66 60) (90 230)(⊂'Style' 'Multi'))
      f1.(First Last Address).Text←
                                FirstName LastName Address
      f1.Save.onSelect←'Update'
      ⎕DQ'f1'
    ∇

    ∇ Update(button event);form
      ⍝ Private method used by Edit as callback on Save
      form←button.##
      FirstName LastName Address←
                             form.(First Last Address).Text
      form.Close
    ∇
:EndClass ⍝ Contact
```

We have not created a constructor for this class, so new instances will have the values declared at the start of the class definition. Register your first three friends as follows:

```
      Friends←3↑⎕NEW Contact
      Friends.Edit
```

This pops us three modal dialog boxes in a row, one after the other – which is hardly ideal. We'll improve on that in the next chapter.

CHAPTER 10

Interfaces

As we have seen, a *base class* provides core functionality which other classes can build upon. The core functionality is available from each derived class, unless the derived class intentionally overrides all or part of it. This makes it possible for programs which "know about" the base class behaviour to use most, if not all, classes derived from the same base. If necessary, a user of a derived class can *cast* the instance to the base class using dyadic ⎕CLASS. The following expressions calls the List method of `ExcelWorkBook` via an instance of `PlanBook`.

```
      w←⎕NEW PlanBook 'c:\temp\widgets.xls'
      (ExcelWorkBook ⎕CLASS w).List 'c:\temp'
```

There are situations where it is also useful for classes which do *not* derive from a common base class to expose common behaviour – which is referred to as an *interface*. In the same way as with a classes which derive from a common base class, a program written to use a particular interface should be able to use any class which *implements* the interface.

For example, we can define an interface called `iEditable` which requires an object to be able to:

- Paint itself inside a subform at a given location on a GUI Form and return a reference to the subform which it created for itself
- Pick up new values for its properties when asked to do so

We can then write an editor which was able to edit *any* instance of *any* class which supported this interface. By convention, the names of interfaces begin with a lowercase `i`. Our `iEditable` interface definition might look like this:

```
:Interface iEditable

   ∇ SubForm←Paint(Container Position)
   ⍝ Paint instance in Container at Position, return ref to
SubForm used
   ∇

   ∇ Update
   ⍝ Update properties of instance from Painted controls
   ∇

   ∇ UnPaint
   ⍝ Remove any references to resources created by Paint
   ∇

:EndInterface ⍝ iEditable
```

The interface definition contains no executable code, which will leave many APL programmers wondering what it is for! In most other languages, an interface definition would contain a little more: Declarations of the types of all the arguments and results. APL interface definitions *can* also include type information, if we want to export them for other languages to use – see the final sections on Microsoft .NET for more about this. There *are* other reasons why an interface definition is useful, which we will investigate in a moment.

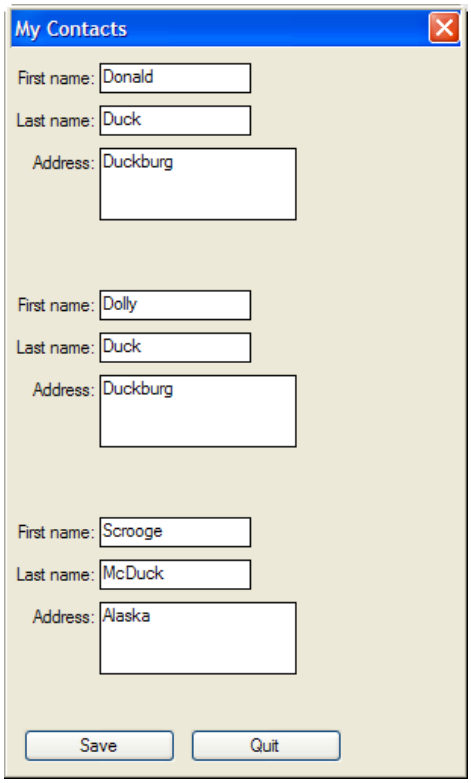
Let us modify the Contacts class we created in the previous chapter, and replace the existing Edit and Update methods with an implementation of iEditable:

```
:Class EditableContact : iEditable
    :Field Public FirstName←''
    :Field Public LastName←''
    :Field Public Address←0⍴⊂''

⍝ --- iEditable implementation

    :Field Private idSubForm ⍝ ref to subform is stored

    ∇ {SubForm}←Paint(container position)
      :Signature SubForm←iEditable.Paint Container,Position

      SubForm←idSubForm←container.⎕NEW 'SubForm'
             ((('Posn'position) ('Size'(120 200))
                                ('BCol'container.BCol))
      SubForm.First←SubForm.⎕NEW #.EditField
             ('First name:'FirstName(10 60)(⍬ 100)⍬)
      SubForm.Last←SubForm.⎕NEW #.EditField
             ('Last name:'LastName(38 60)(⍬ 100)⍬)
      SubForm.Address←SubForm.⎕NEW #.EditField
       ('Address:'Address(66 60)(48 130)(⊂'Style' 'Multi'))
    ∇

    ∇ Update
      :Signature iEditable.Update
      FirstName LastName Address←
                       idSubForm.(First Last Address).Text
    ∇

    ∇ UnPaint
      :Signature iEditable.UnPaint
      idSubForm←⍬
    ∇

:EndClass ⍝ Contact
```

With this class defined, we can create a form and arrange our contacts on it. Note that, in order to access the interface, we have to cast each instance of `EditableContact` to `iEditable`, in the same way that we might cast to a base clase if we wanted to access that. This is in order to make it possible for a class to add an interface without the risk of name conflicts between interface member names and members of the class itself:

```
      contacts←3↑⎕NEW EditableContact
      e_contacts←iEditable ⎕CLASS¨contacts ⍝ Cast each to
iEditable
      e_contacts[1].⎕nl -3 ⍝ Each one exposes 3 methods
 Paint  UnPaint  Update

      f1←⎕NEW Dialog ('My Contacts' (480 300) ⍬)
      e_contacts[1].Paint f1 (0 0)
      e_contacts[2].Paint f1 (150 0)
      e_contacts[3].Paint f1 (300 0)
      ⍝ or:       e_contacts {⍺.Paint f1 ⍵}¨(0 0)(150
0)(300 0)
```

At this point, pause to fill in the form:



When we are ready, we run the update functions:

```
      e_contacts.Update ⍝ Runs iEditable.Update on each
instance
      contacts.(FirstName LastName)
  Donald  Duck    Dolly  Duck    Scrooge  McDuck
```

We could also have defined a function to do the update and connected it to the Save button:

```
      upd←{e_contacts.Update}
      f1.Save.onSelect←'upd'
```

The reason we need an `UnPaint` method can be illustrated by the following sequence. Note that the form does not disappear when expunged. This is due to the references (in the private fields called idSubForm) inside each contact:

```
      ⎕ex 'f1'
      e_contacts.UnPaint
```

Of course, if `contacts` and `e_contacts` were local to the function doing the above work, or if we expunged these, this would also cause the references to disappear.

Avoiding Name Conflicts

Through the ages, many APL developers have “independently discovered” the need for interfaces, and written code which (for example) checks the class of a name like '`Paint`', and if the function is present deduces that “the interface is available”. However, this is dangerous: A class might have a method called `Paint` which has nothing to do with “being `iEditable`”. Even if we are rigorous and check all the names in the interface, there is a risk of confusion with similar interfaces. In addition to this: What it if our class already has a method called `Paint` and we want to add the interface to it?

The important reason for having an interface definition is the avoidance of these name conflicts between interfaces, or between an interface and names used by the class. In the `EditableContact` class, each of the interface functions has a `:Signature` statement, for example:

```
    ∇ {SubForm}←Paint(container position)
… stuff snipped …
      :Signature SubForm←iEditable.Paint Container,Position
```

It is the name `iEditable.Paint` in the `:Signature` statement which declares that the function implements the `Paint` method of the `iEditable` interface. The name of the actual function is not actually used. It makes sense to use the same name as the interface function, but if we needed to avoid name conflicts we could call it `ie_Paint` or `foo` – anything we like.

Conclusion

The chapter on interfaces concludes the introduction to object oriented concepts as implemented in Dyalog APL. The following chapters show how object orientation can be used to integrate APL with other tools which inhabit the Microsoft .NET Framework.

CHAPTER 11

Using The Microsoft .NET Framework

Operating systems are designed to hide the details of the machine hardware from developers, making it possible to write applications without understanding the details of how the chips which manage memory, disk, keyboard, screen and other peripherals are controlled. As operating systems have evolved, they have tended to provided ever higher levels of abstraction. Environments like Microsoft .NET provide object-oriented encapsulations of everything from low level hardware interfaces to high level GUI and Databases.

If you are running Dyalog APL on a machine where the Microsoft .NET Framework is installed, you have access to a vast collection of tools for application building, which are supplied by Microsoft. All of these classes, any classes you have acquired from third parties, plus the classes you write yourself in a language which supports .NET (Visual C#, Dyalog APL, and dozens of other languages), can be used in the same way as you would use the APL classes described in the preceding chapters.

All of the ".NET classes" mentioned above, whether they are something you have written yourself or they were provided by Microsoft, reside in files called *Assemblies*. Microsoft has decided to reuse the extension ".DLL" for these files, so they have the same extension as traditional Dynamic Link Libraries. There can be dozens, or hundreds of assemblies on your machine, so a mechanism is required to name the assemblies that an application would like to use. In Dyalog APL, the system variable `⎕USING` provides this.

`⎕USING` contains a list of assembly names (see the Dyalog APL documentation for details). If one of the elements is an empty vector, this is taken to mean the Microsoft .NET Framework classes contained in the assembly called `mscorlib.dll`. This core library contains the most commonly used classes. To give an impression of the types of services provided by .NET, the remainder of this chapter will explore a (very) small selection.

System.Environment

`System.Environment` is a class which contains a number of useful shared properties and methods which provide information about the platform on which the application is running:

```
      ⎕USING←'' ⍝ This is interpreted as ⎕USING←,⊂''

      System.Environment.⎕nl -2
 CommandLine  CurrentDirectory  ExitCode
    HasShutdownStarted MachineName  NewLine  OSVersion
    ProcessorCount  StackTrace  SystemDirectory  TickCount
    UserDomainName  UserInteractive  UserName  Version
    WorkingSet
```

```
      se←System.Environment
      se.(Version OSVersion)
 2.0.50727.42  Microsoft Windows NT 5.1.2600 Service Pack 2

      se.SpecialFolder.⎕NL-2
 ApplicationData  CommonApplicationData  CommonProgramFiles
    Cookies
    …etc…

      se.(GetFolderPath SpecialFolder.ProgramFiles)
C:\Program Files

      folders←se.SpecialFolder.⎕NL-2
      ↑se.{⍵ (GetFolderPath SpecialFolder⍎⍵)}¨folders
ApplicationData        C:\Documents and Settings\mkrom\...
CommonApplicationData  C:\Documents and Settings\All Users\
…etc…

      ⎕avɪse.NewLine
4 3
```

We could have named the System namespace in `⎕USING`. This would have allowed us to leave the `System` prefix out of our references to classes:

```
      ⎕USING←'System' ⍝ Equivalent to ⎕USING←,⊂'System'
      Environment.Version
 2.0.50727.42
```

The author's personal preference is still to use fully qualified names – I suspect this may change as I (and the rest of the APL community) start using these classes more frequently, and start to recognize Environment as meaning System.Environment.
However: Note that APL will only search for .NET classes if the name which is used would give a `VALUE ERROR` in the APL workspace. The use of so-called "namespace prefixes" in `⎕USING` increases the likelihood of a name conflict between your own names and those in an assembly which you are trying to use (for example, Version is more likely to conflict with a name in the workspace than is System.Version). We'll return to this topic in the next chapter.

System.Globalization.CultureInfo & DateTimeFormatInfo

The DateTimeFormatInfo class contains information about the date and time formats for a given culture. You can get hold of an instance of DateTimeFormatInfo for the current culture, or for a specific one:

```
    current←System.Globalization.CultureInfo.CurrentCulture
      current.(Name EnglishName)
 da-DK  Danish (Denmark)

      dtf←current.DateTimeFormat
      dtf.⎕nl -2
 AbbreviatedDayNames  AbbreviatedMonthGenitiveNames
    AbreviatedMonthNames  AMDesignator  Calendar
    CalendarWeekRule  CurrentInfo  DateSeparator  DayNames
    FirstDayOfWeek  …etc…

     dtf.MonthNames
 januar  februar  marts  april  maj  juni  juli  ...

     dtf.FullDateTimePattern
dddd, dd MMMM yyyy HH:mm:ss

     dtf.GetShortestDayName¨0 1 2 3 4 5 6
 sø  ma  ti  on  to  fr  lø

     dtf.NativeCalendarName
Den gregorianske kalender

     de←(⎕NEW System.Globalization.CultureInfo⊂'de-DE')).
                                          DateTimeFormat
     de.DayNames
 Sonntag  Montag  Dienstag  Mittwoch  Donnerstag  ...
```

`de-DE` means German as spoken (or rather, written…) in Germany,

System.DateTime and System.TimeSpan

DateTime and TimeSpan provide tools for working with timestamps, including doing arithmetic on them:

```
      ⍞←may29←⍞NEW System.DateTime (2006 5 29)
29-05-2006 00:00:00
      ⍞←aday←⍞new System.TimeSpan (24 0 0) ⍝ 24 hours
1.00:00:00
      ⍞←may28←may29-aday
28-05-2006 00:00:00
      +\5⍴aday
 1.00:00:00  2.00:00:00  3.00:00:00  4.00:00:00  5.00:00:00
      (+\5⍴aday).Days
1 2 3 4 5

      nextweek←may28++\5⍴aday
      ,[1.5]nextweek
 29-05-2006 00:00:00
 30-05-2006 00:00:00
 31-05-2006 00:00:00
 01-06-2006 00:00:00
 02-06-2006 00:00:00
      nextweek.(Month Day)
 5 29  5 30  5 31  6 1  6 2
```

```
      nextweek>may29+aday
0 0 1 1 1
      (nextweek-System.DateTime.MinValue).Days
732459 732460 732461 732462 732463
```

System.IO.DirectoryInfo

The DirectoryInfo class contains methods for inspecting the contents of file system folders:

```
      temp←⎕NEW System.IO.DirectoryInfo (⊂'c:\temp')
      temp.CreateSubdirectory ⊂'subdir'
c:\temp\subdir
      ↑(temp.GetDirectories ⊂'*.*').(Name CreationTime)
 DyalogWebSite  30-05-2006 15:35:40
 js             24-02-2006 22:30:06
 subdir         12-06-2006 14:17:14
      (temp.GetDirectories ⊂'subdir').Delete 1

      files←temp.GetFiles ⊂'a*.dws'
      ↑files.(FullName CreationTime LastAccessTime)
 c:\temp\a.DWS     18-01-2006 17:17:01  05-06-2006 21:57:06
 c:\temp\ab.DWS    12-04-2006 15:54:36  05-06-2006 21:57:06
 c:\temp\ado.DWS   17-06-2005 13:22:08  05-06-2006 21:57:06

      ,[1.5] ⎕CLASS files[1]
  System.IO.FileInfo
  System.IO.FileSystemInfo System.Runtime.Serialization.
                                                ISerializable
  System.MarshalByRefObject
  System.Object
```

The final result above shows that each element of the result is an instance of System.IO.FileInfo, which derives from System.IO.FileSystemInfo (which implements the interface System.Runtime.Serialization.Iserializable), which derives from System.MarshalByRefObject. At the end of the day, everything derives from System.Object.

System.IO.FileInfo

As we have seen above, the GetFiles method of DirectoryInfo returns instances of System.IO.FileInfo, which is a companion class for DirectoryInfo:

```
      (a←files[1]).⎕nl -2
Attributes  CreationTime  CreationTimeUtc  Directory
      DirectoryName  Exists  Extension  FullName
      IsReadOnly  LastAccessTime  LastAccessTimeUtc
      LastWriteTime  LastWriteTimeUtc  Length  Name

      a.(Name Exists IsReadOnly)
a.DWS  1 0

      z←a.(CopyTo⊂'c:\temp\z.dws')
      z.(FullName CreationTime)
c:\temp\z.dws  14-06-2006 16:28:30
      z.Delete
```

Summary

The above examples represent a very small subset of the classes provided by the Microsoft .NET framework. There are classes for handling web requests in HTTP and FTP format, for compressing files, sending and receiving e-mail, GUI, database access, graphics and printing, the list is almost endless.

In addition to providing the framework itself, the .NET environment specifies calling conventions which mean that classes implemented in all .NET languages – including Dyalog APL - are fully interoperable. A class written in any language can use, derive from, be called, and be used as a base class by any other .NET language. This allows us to integrate APL with other languages and tools more easily than ever before.

CHAPTER 12

Using APL Classes from .NET

In the workspace called `DotNet` in the `OO4APL` folder, there is a class with two "mathematical" methods in it:

```
      )copy DotNet
Dotnet saved … etc …
      Maths.Round (○1) 2
3.14
      Maths.Fib 10
55
```

The definition of the class is:

```
:Class Maths

    ∇ r←Round(n decimals);base
      :Access Public Shared
      base←10*decimals
      r←(⌊0.5+n×base)÷base
    ∇

      fibonacci←{ ⍝ Tail-recursive Fibonacci from ws "dfns"
          α←0 1
          ⍵=0:⊖⍴α
          (1↓α,+/α)∇ ⍵-1
      }

    ∇ r←Fib n
      :Access Public Shared
      r←fibonacci n
    ∇

:EndClass ⍝ Math
```

We can make this class available to all Microsoft .NET applications if we copy it into a workspace and subsequently export the workspace as as .NET *assembly*. As a service to users of typed languages like C#, we probably want to add type declarations of the .NET types of the parameters and results of the public methods before we do the export. This is not strictly necessary, but without it all types will default to System.Object.

The result of this will be that most C# users will have to *cast* data to or from System.Object in order to use our class. Our assembly will be more pleasant to use if we can declare everything using the simplest or closest .NET type.

To make the declarations, we must first add a :Using statement which allows us to find .NETdata types (this is usually done at the top of the class script):

```
:Using System
```

Next, we must add one line to each of our methods (a class containing these declarations exists in the workspace `DotNet` under the name `MathsX`). Note that dfns cannot be used as public methods in a class, but must be "covered" by a traditional function which can contain declarative statements.

```
    ∇ r←Round(n decimals);base
      :Signature Double←Round Double N, Int32 Decimals

    ∇ r←Fib n
      :Signature Double←Fib Int32 N
```

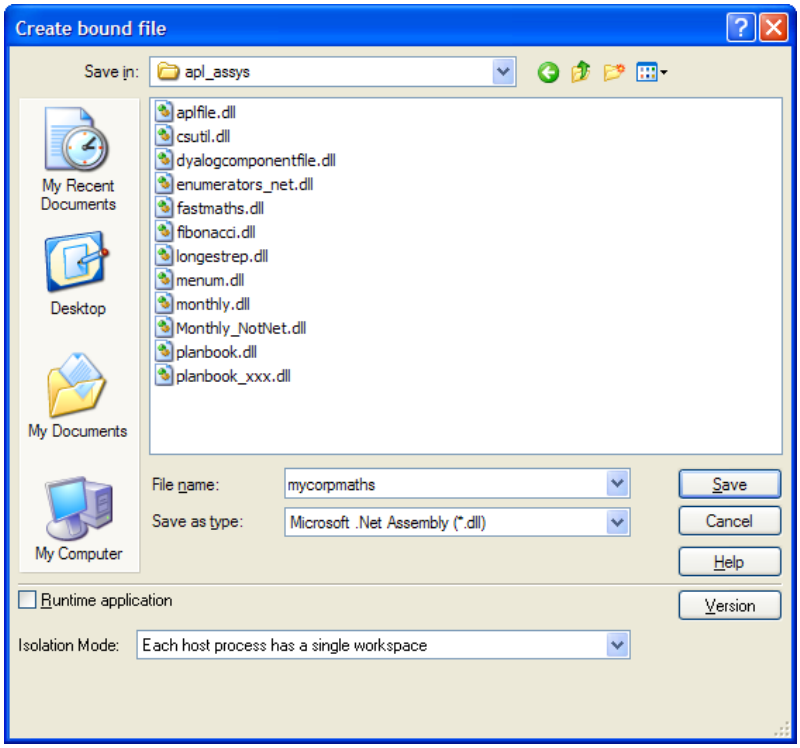
The first `:Signature` declares that `Round` returns a result of type `Double` (which means a double-precision floating-point number), and takes two parameters. The first is a `Double` which is called `N` (most .NET development tools will make this name visible to a developer using our class), and the second parameter is a 32-bit integer called `Decimals`.

After adding the signatures, we are ready to export our class:

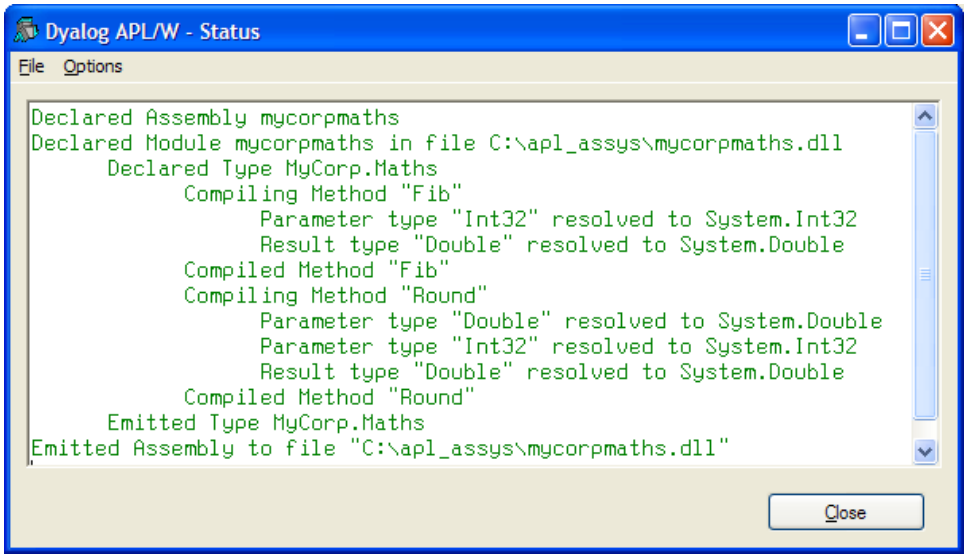
```
      )clear
clear ws
      )NS MyCorp
#.MyCorp
      )CS MyCorp
#.MyCorp
      )copy dotnet Maths
…dotnet saved Mon Jun 19 14:01:18 2006
```

In the above example, we created a namespace `MyCorp` and copied `Maths` into it. This will export our class inside a .NET namespace called `MyCorp`, so that the class can be referred to as `MyCorp.Maths`. It is customary to embed classes within at least one level of namespaces – often two levels, typically a company name followed by a product name - in order to organize classes in applications which use classes from a number of different sources.

Select *File|Export* from the session menu, select "Microsoft .NET Assembly" as the file type and probably uncheck "Runtime Application":



When you press *Save*, the following text should appear in the Status window:



The output allows us to check that the two methods `Fib` & `Round` were exported, with the expected parameter and result types. We can now write a C# program which uses our class, for example:

```
public class Test
{
    static void Main()
    {
        System.Console.Write("Round(2/3,2) = ");

System.Console.WriteLine(MyCorp.Maths.Round(0.6666,2));
        System.Console.Write("Fib(10) = ");
        System.Console.WriteLine(MyCorp.Maths.Fib(10));
    }
}
```

The workspace DotNet in the OO4APL folder contains a function called `CSC` (for C Sharp Compiler), which can be used to call the C# compiler. If we save the above program in a file called `c:\apl_assys\testmaths.cs` (or copy the file by this name from the OO4APL folder), we can compile it by calling:

```
    'winexe' CSC 'c:\apl_assys\testmaths'
                              'c:\apl_assys\mycorpmaths.dll'
0  Microsoft (R) Visual C# .NET Compiler version 7.10…
   for Microsoft (R) .NET Framework version 1.1.4322
   Copyright (C) Microsoft Corporation 2001-2002…
```

The left argument of `'winexe'` instructs the C# compiler to build a Windows Executable (the default would be to make an assembly). The right argument contains the name of the source file (without the .cs extension), optionally followed by any assemblies which it needs. The result is a file with the same name as the source file, but with an extension of .exe. We call it from APL using `⎕CMD`:

```
      ⎕CMD 'c:\apl_assys\testmaths.exe'
Round(2/3,2) = 0,67
Fib(10) = 55
```

It is important to note that our assembly (mycorpmaths.dll) can be used from ANY language which can use Microsoft .NET. The list includes APL – see the web page http://www.gotdotnet.com/team/lang/. In fact, we can use the assembly from APL, in the same way that we would use any other .NET assembly:

```
      )clear
clear ws
       ⎕using←',c:\apl_assys\mycorpmaths.dll'
       MyCorp.Maths.Round (○1) 3
3.142
```

CHAPTER 13

Inheriting from a .NET Base Class

In the same way that you can inherit from one of the built-in Dyalog GUI classes, you can write a class which derives from a .NET base class. You can derive from classes which you write yourself in a .NET language (including Dyalog APL), classes purchased from a 3rd party tool vendor, or from Microsoft .NET Framework classes. Very many of the Framework classes are *sealed* for security and performance reasons, which means that they can not be used as base classes, but quite a few are intended for use as base classes.

We will now look at writing our own class in C#, as an optimization of our fibonacci function. The Fibonacci series is generated using a recursive algorithm which does a VERY small amount of processing for each level of recursion. Even though tail-recursive dfns are highly efficient, a compiled, strongly typed language like C# is goint to be able to execute this particular type of algorithm significantly faster than APL can do it.

If we build our Maths class on a base class written in C#, we can easily substitute some of the methods by methods written in C#. For example, we can write the following (ugly but fast) C# class:

```
using System;

namespace OO4APL
{ public class Maths
    { public static double Fib(int n)
        {   double n1 = 0, n2 = 1, r = 0;
            int i=1;
                    while (i<n)
                    {   i++;  r = n1 + n2;
                        n1 = n2; n2 = r;   }
                    return n2;
        }
    }
}
```

If we save this code in a file called `fibonacci.cs` (which can be copied from the OO4APL folder), and compile this class using our `CSC` function:

```
      CSC 'c:\apl_assys\fibonacci'
0  Microsoft (R) Visual C# .NET Compiler version…
   …etc…
```

We can now write a new class `FastMaths`, which derives from `OO4APL.Maths`. It contains the `Round` method which are still written in APL, while `Fib` will be inherited from the base class:

```
:Class FastMaths : OO4APL.Maths
:Using ,c:\apl_assys\fibonacci.dll

    ∇ r←Round(n decimals);base
      :Access Public Shared
      base←10*decimals
      r←(⌊0.5+n×base)÷base
    ∇

:EndClass ⍝ Math
```

A class which derives from a .NET class must be exported as a .NET class before it can be used. This restriction may be relaxed in a future release. If you want to test the class quickly, without creating a .NET assembly as a .dll, it is sufficient to export the class to memory using the menu item *File|Export to memory*, after which it can be called:

```
      FastMaths.Round(○1)3
3.142
      FastMaths.Fib 10
55
```

The first method runs in APL, and the second in C#. From the users point of view, there is no discernible difference between this class and the one which was written entirely in APL (apart from the speed).

Conclusion

This concludes the Introduction to Object Oriented Programming for APL Programmers. If you want to learn more, it is time to read the Release Notes and the Language Reference, both of which contain a number of additional examples.