OO Exercises in APL   V1.05

Introduction

This document is not meant to be an explanation of Object Oriented programming (OO) or why you would want to use it but rather a series of very simple exercises on HOW to use it in Dyalog APL. It is divided into sections presented in order you are most likely to use them.

As a rule special words are *italicized* and :**keywords** and APL names are **bold**. Function names are surrounded by <angle brackets> and variables by quotes when the context is unclear. Colours are used and *class* names are blue and *instances* are green. Expressions in sentences will be coloured orange. APL expressions and code are on a light grey background.

The first few sections will introduce the OO terms with short examples/problems that can only be used to describe how it works. Later sections will present more practical examples.

Most examples will work under the minimum V11 but V12.1 or later should be preferred.

Assume ⎕IO 1 and ⎕ML 0 unless otherwise specified.

Section 1 - Classes

Classes are models for namespaces like blueprints are for construction work.

To edit a new class use the statement

```
      )ED  ○ className
```

Where 'o' is the APL symbol for PI (Ctrl-O). Only necessary the first time a class is created.

A class definition always starts with the keyword **:Class** and ends with the keyword **:EndClass** on a separate line. The editor will insert these lines for you. You may insert comments on separate lines before the :**class** keyword, but nothing else. You may not add anything after **:endclass**.

Classes contain elements called *members* (data and code).

You can make (construct) namespaces from the classes, they are called *instances*.

When creating an *instance* from a class you can initiate or prepare it, pretty much like []LX can prepare a workspace when it is)LOADed.

To create an instance of a class you use the system function []NEW as in

```
      Instance ← ⎕NEW  class
```

[]NEW will create an instance of the class given as argument. If data is needed to create the instance put right after the class as in

```
      Instance ← ⎕NEW  class  data
```

Members

The *members* of a class are **fields**, **properties, methods** and nested **classes**.

Fields are variables.

Properties are used like variables but implemented by (usually a pair of) functions. You assign them and read them like variables. An analogy for read only properties is a *niladic* function.

If the property only has a <get> function it is *read only*, if it only has a <set> function it is *write only*, if it has both it is *read-write*.

Methods are functions.

To declare a *field* simply precede its name by the term **:Field** as in

```
      :Field  ABC
```

To declare a *field* with a value simply follow its name by the assignment as in

```
      :Field  ABC←expression
```

A *field* may be read only. In this case you add **readonly** between ;*field* and the name as in

```
      :Field   readonly  ABC←expression
```

To declare a property use the term **:Property**, followed by a <**set**> function and/or a <**get**> function and end with the **:endProperty** keyword on a separate line, e.g.

```
:Property   MyProp
      ∇set  x
      ...
      ∇
      ∇r←get
      ...
      ∇
:EndProperty
```

If the <**set**> function is missing the property is read only. It the <**get**> function is missing the property is write only.

To declare a *method*, simply enter its normal APL function definition surrounded by dels (∇) if necessary (D-fns don't need them). Line numbers should be avoided.

Nested *classes* follow the same rules as their parent class. They are found in larger (parent) classes. It is a way of logically group classes that are only used in one place. It increases encapsulation and can lead to more readable and maintainable code.

Solved problems

1. Create class **f3** with 3 fields **X, Y** and **Z**
2. Create class **M1** with a method called <**mono**>
3. Create class **rop** with a read only property called **Time**
4. Create a class **parent** with a nested class

Answers

In each case use the editor (**)ED**) followed by ctrl-O then the name of the class

1.
```
:Class   f3
      :field  X
      :field  Y
      :field  Z
:endclass
```

2.
```
:class   M1
      ∇ Mono
      'This method does not do much'
      ∇
:endclass
```

3.
```
:class   rop
      :property Time
      ∇r←get
      r←⎕TS[4 5 6]
      ∇
      :EndProperty
:endclass
```

4.
```
:class parent
      :class child
      ⍝ This class is empty
      :endclass
:endclass
```

Section 2 - Encapsulation

Classes' members are, by default, invisible outside the instance thus encapsulating the object. By analogy, names in APL functions are made invisible outside the function by localising them in the function header. Members can be made visible and accessible using the keyword **public**. By default, all members are deemed **private** as if that keyword had been used.

To declare a *field* public or private it suffices to insert the keyword **public** or **private** between the keyword :**field** and the name of the field, e.g.

```
      :field   public  ABC←42
```

To declare a *property* public or private you need to use the keyword *:access* followed by either **private** or **public**, after the *:property* statement, e.g.

```
:Property  P
:Access   public
```

Same thing for methods: simply insert the *:Access **public*** statement somewhere in the body of the function. This is only possible with traditional functions, operators or D-functions cannot be made public.

Nested classes are also similar. By default they are private. To make them visible from the outside add the :*access public* statement.

Solved problems

1. Create class **pupr** with one public *field* called **Name** and one *private field* name **Price**
2. Create class **M2** with one public *method* and one *private method* used by the public one
3. Create a class with one readonly *property* called **Time** which uses *public field* **TZ**
4. Create class **mom** which contains *public class* **baby**

Answers

1.
```
:class  pupr
      :field public Name
      :field private Price
:endclass
```

2.
```
:class  M2
      ∇ Mono
      :access public
      Spaceout   'this method prints this msg'
      ∇
      Spaceout ←{((2×⍴,⍵)⍴0 1)\⍵}
:endclass
```

In this last example, <**Spaceout**> is a private method. It can be called inside the class but not directly by the user of the class. Only <**Mono**> is visible and usable externally.

3.

```
:Class TellTime
    :field public TZ← ¯5 0 0  ⍝ EST
    :property Time
    :access public
        ∇ r←get
          r←1↓,'<:>,ZI2'⎕FMT 24 0 0 | TZ + 3↑3↓⎕TS
        ∇
    :endproperty
:EndClass
```

Here field **TZ** is used by read only property **Time** on line 1 of the **get** function.

4.

```
:Class mom
    :Class baby
    :access public
    :EndClass
:EndClass
```

The parent class is always public.

Section 3 – Instances

If classes are models, instances are tangible objects created from them. As shown before, APL provides a way to create a completely independent object modelled on a class through the system function []NEW. If **'MyClass'** is a class then doing

```
      MyCopy←⎕NEW  MyClass
```

will create an instance of it. It is initially identical to the class.

Initialization

Instances can be set up or *constructed* when they are created. To do so at least one of the methods must be called (automatically) when the instance is created. To identify a method as a **constructor** the statement

```
:implements  constructor
```

Must appear somewhere in the function used to construct the instance, e.g.

```
:class  time
      :Field  Time ⍝ uninitialized - will be done in the constructor
      ∇ new
      :implements constructor
      :access public
      Time←3↑3↓⎕TS
      ∇
:endclass
```

Because the method must be accessible externally the statement **:access public** must also appear.

A *constructor* is akin to []LX which initializes a workspace: it initializes the instance.

Arguments to constructors

It is possible to supply arguments to the constructor of the class.

[]NEW also accepts a two element vector as argument, the second element being the argument to the constructor. So, given the class

```
:class  e4
      ∇ ctor  arg
       :implements constructor
       :access public
       Date←arg
      ∇
:endclass
```

Doing

```
      dins←⎕NEW  e4  '2008/10/12'
```

will set field **Date**[1] to the string ‘2008/10/12’ in the instance ‘**dins’**.

In this case, if no argument is supplied APL will complain with a LENGTH error because the constructor requires an argument.

It is also possible to create a class that may or may not take an argument. In this case there should be 2 constructors: one that takes an argument and one that does not. APL will use the appropriate constructor depending on the number of arguments.

Solved problems

1. Create a class named **stats** with 1 private field called **Obs** and one method used to set Obs with the ravel of []NEW’s 2nd argument when creating an instance
2. Change the class to have a property returning the average of **Obs**
3. Create an instance with the values 1 to 99 and find their average
4. Given a list of numeric vectors **LNV**, create a vector of instances each constructed with the values of each vector in LNV
5. Find the average of Obs in each instance of the vector resulting in 4.
6. Create a class ‘date’ that takes an argument to initialize its DATE field as a 3 element vector. If none is supplied it uses today’s date.

Answers

1.
```
:class stats
      :Field private Obs
      ∇ setobsto  arg
      :implements constructor
      :access public
      Obs←,arg
      ∇
:endclass
```

2.
```
…
:property  AVG
:access public
 ∇r←get
  r←(+/Obs) ÷ ⍴Obs
 ∇
:EndProperty
…
```

3.
```
      istats  ← ⎕NEW  stats  (⍳99)
      istats.AVG
```

[1] Although **Date** has not been declared a Field and is a variable, it is considered a *private field* for OO purposes

4.
```
      vistats ← {⎕NEW  stats  ⍵}¨LNV
```

5.
```
      vistats.AVG
```

6. This class requires 2 constructors: one for the case with an argument and one for the case with no argument:

```
:class  date
      :field public DATE
      ∇ set1date arg
      :implements constructor
      :access public
      DATE ← arg
      ∇

      ∇ setnodate
      :implements constructor
      :access public
      DATE ← 3↑⎕TS
      ∇
:endclass
```

Finding what is accessible in an instance from outside

The system function []NL can be used to ‘see’ which members are available.

[]NL ¯2 returns the list of **public** *fields* and *properties* (variables) and []NL ¯3 returns the list of **public** *methods* (functions), both in list of character vectors format (VTVs). It is not possible to have public operators and []NL ¯4 will always return an empty list. []NL ¯9 will return the list of *nested* public classes, if any.

Including common code

Very often you have code that is common to many classes. Instead of rewriting these functions for every class you may put them in a namespace, scripted or not, and include the namespace in all the classes where the code is needed by simply adding this statement in the body of the class

:*include* common

For example, if you have a namespace **Tools** containing only the *public* functions[2] **sqr**, **root** and **avg** and the following class:

```
:class  etools
      :include  Tools
      :field  public  F
```

[2] Functions can be declared public using the :Access public statement in a regular namespace as long as they are not D functions

```
      ∇ r←correct stats
      :access public
      r←root  avg  sqr  stats
      ∇
:endclass
```

You would see

```
      iet ←⎕NEW etools
      iet. ⎕NL-2 3
avg   correct    root   sqr   F
```

Solved problems

1. Create a class named **SALE** with 3 public fields named **ID, Price** and **Quantity** and one method used to initialize them
2. Change the class to have a property returning the total of the sale
3. Create 3 instances using the values ‘PC’ 200 2000, ‘APL/p’ 150 50 and ‘APL/S’ 30 1000
4. List the **ID** and **Total** of those instances in table form (1 row per instance).
5. Adjust the **Price** of each sale by +10%
6. Adjust the **Price** of each sale whose total exceeds 100,000 by -15%
7. Create a class doing simple statistical analysis. It will use general utilities in a separate namespace.

Answers

1.

```
:class  SALE
      :Field public ID
      :Field public Price
      :Field public Quantity

      ∇ setdata  arg
      :implements constructor
      :access public
      (ID   Quantity   Price) ← arg
      ∇
:endclass
```

2. Here the total is given as Price x Quantity. It is read-only since there is no set> function.

```
…
:property  Total
:access public
 ∇r←get
  r←Price × Quantity
 ∇
:EndProperty
…
```

3.
```
   s←{⎕NEW SALE ⍵}¨(‘PC’ 200 2000)( ‘APL/p’ 150 50)(‘APL/S’ 30 1000)
```

4.
```
      ↑ s.(ID Total)
```

5. This is merely multiplying every Price by 1.1:
```
      s.Price ×← 1.1
```

6. This is multiplying by 0.85 the price of those sales whose total>100000:
```
      ((s.Total>100000)/s).(Price ×← .85)
```

Note that when there are no instances to ‘dot’ into (as in 0/s above) APL must still build one instance from the class to find what it would look like. If the class requires an argument to construct an instance then APL is unable to comply. For this reason, here, the expression will fail if there are no Total>100000 because APL will then need to build an instance of ‘SALE’ to find what ‘Price’ would look like. In this case APL must create an instance that requires an argument to its constructor and none can be provided.

On the other hand, had there been no constructor or a constructor NOT requiring an argument then an instance would have been generated and a Price would have been created from which a prototype would have been made.

7. The namespace contains some math functions like square, square root, etc.

```
:namespace mathUtils
    square←*∘2
    cube←{⍵*3}
    ∇ r←sqrroot x
      :Access public
      r←x*0.5
    ∇
:endnamespace
```

The class uses them through the :*include* statement:

```
:class statUtils
    :include mathUtils
    ∇ r←average list
      :Access public
      r←(+/list)÷⍴,list
    ∇
    ∇ r←stddev list   ⍝ standard deviation
      :Access public
      r←sqrroot average square list-average list
    ∇
:endclass
```

Note that only the public functions are listed with []NL, including those declared public in the included namespace:

```
      s←⎕NEW statUtils
      s.stddev  4 1 0 2 1 5 2 3 4 3
1.5
      s.⎕nl ¯3
 average  sqrroot  stddev
```

Section 4 – Shared vs. Instance

Sometimes members are better kept in the class. For example, a count or a description of each instance created can only be kept in the class. Each instance can have access to it but none can be the bearer as instances do not "see" each other. We say that such members are ***shared*** or that they are ***class*** members. The following class is an example:

```
:Class  tally
    :field shared Count←0
    ∇ init
      :Access public
      :Implements constructor
      Count+←1
     ...
    ∇
:EndClass
```

When an *instance* is created the Count is increased by one but it remains in the *class*. It is visible from within all *instances* but unless public it remains visible only within the group *class-instances*. If *public* it can also be accessed directly through the *class*. Here tally.Count would do it if declared *public*.

Likewise, an *instance* method is a function that can be called **only** from an instance. A class method (one which is *shared*) can **also** be called directly from the class but it can only refer to *shared* fields.

Useful Tools

Namespaces display form

All namespaces, whether a class, an instance or a namespace, all have a display form which is usually the path of the namespace or, for *instances*, the path of their *class* surrounded by brackets.

A *class* C is displayed as #.C but its *instance* will be displayed as #.[C].

There is a way to change the display form by using the ⎕DF system function. For example, to change the display of # to 'ws' you could use #.[]DF 'ws'.

Finding a reference to itself

One way to find in which space we are is by using the system variable ⎕THIS. You can also use ⎕CS ''. ⎕THIS returns a reference whereas ⎕CS '' returns a character string equivalent to ⍕⎕THIS.

Finding the instances of a class

It is possible to find all the *instances* of a *class* thru the system function ⎕INSTANCES. This returns a series of references to *instances*. They will be displayed as per their display format.

One way a *class* can find a list of references to all its *instances* is by doing ⎕INSTANCES ⎕THIS.

Solved problems

1. Create a class named **SALE** with 3 public fields named **ID, Price** and **Quantity** and one method used to initialize them. Each instance will have a display form equal to its ID.

2. Create a method to determine the average total for ALL the instances of the SALE class

Answers

1. We can reuse the class in problem 1 of the previous exercise on page 10. All we need to do is modify the constructor to change the display form:

```
:class  SALE
      :Field public ID
      :Field public Price
      :Field public Quantity

      ∇ setdata  iqp
      :implements constructor
      :access public
      (ID   Quantity   Price) ← iqp
      ⎕DF ‘SALE ‘,ID
      ∇
:endclass
```

The display form will apply to the instance. It is set at creation time, in the constructor.

2. Here we can either use the property Total as before or add a new field **Total**

```
      :Field public Total
```

and add a line to compute the value in the constructor:

```
      Total ← Price × Quantity
```

either way, the method <**AvgTtl**> can be defined like this:

```
    ∇ r←AvgTtl
      :Access public shared
      r←{(+/⍵)÷⍴⍵}(⎕INSTANCES ⎕THIS).Total
    ∇
```

The method must be run from within the class and therefore be *shared*. It must also be *public* to be accessible from outside the class. []THIS is a reference to the class itself and []INSTANCES returns a reference to ALL the instances in the workspace of the class. From each one of them the ‘Total’ is returned and the average computed.

Note that <**AvgTtl**> is niladic and a property (of the same name) could have been used instead. It’s up to the programmer to make these kinds of decisions.

Section 5 – Inheritance

Inheritance is a way to avoid rewriting code by writing general purpose classes and classes that *derive* from them and *inherit* their *members*.

This helps achieve *reusability*, a cornerstone of OO, by avoiding rewriting code and using what already exists.

We say that a (*derived*) class is *based* on another one.

All members in the base can be inherited from the derived class but unless the base class provides a way to access its private members only the public ones will be accessible. All inherited members can be redefined.

When an *instance* is created from a class *based* on another one it inherits all its members automatically.

Here is an example.

Class A has 4 members, 3 public and 1 private:

```
:Class A
:Field public  F1 ←1
:Field public  F2 ←2
:Field private F3 ←3
    ∇ r←M1
      r←F3  ⍝ provide access to F3 through M1
      :Access public
    ∇
:endclass
```

Class B is based on A and has also 3 members, 2 public and 1 private:

```
:Class B: A
:Field public  F2 ←11
:Field public  F4 ←12
:Field private F6 ←13
:endclass
```

Instances of class B will have 4 visible (*public*) fields: **F1** (from A), **F2** (from B which overrides the definition from A), **F4** (only in B) and <**M1**> (from A). Thus

```
      in ←⎕NEW B
      in. ⎕NL-2 3
F1  F2  F4  M1
      in.M1       ⍝ access to private F3 in A thru M1
3
```

Constructions

When an instance is created from a *derived class* the *base class* is used first as model then the *derived class*' members are added. If the *derived class* uses a constructor to initialize the instance it is then called. If the *base class* also has a constructor it will be called automatically through the :**implement constructor** statement. If it requires an argument you must supply it using :*based* as in

```
:implements constructor :based  argument
```

If a *derived* class does not have a constructor but its *base* class does then it is called automatically. In that case the *base* class' constructor must be *niladic*.

If several classes are based one on another their constructors' call are made in the same fashion.

Solved Problems

1. Create a class B1 with a public field **BF.** B1 will be used as base for class D1 which will have private field **DF**. Show that an instance of class D1 has both fields **BF** and **DF**.

   Answer: define B1 and D1 as follows:

```
:Class B1
    :field public BF
:EndClass

:Class D1: B1
    :field public DF
:EndClass
```

   Then create the instance and ask for its fields:

```
      i1←⎕new D1
      i1.⎕nl-2
 BF   DF
```

   Note that fields **BF** and **DF** are defined even though they do not have a value yet. This is similar to what happens when you share a name (using []SVO) before giving it a value: its class (2) is set but referencing it will yield a VALUE error until it is assigned.

2. Create a class D2 based on B2. B2 has one field **BF2** with value 90. D2 has a constructor which sets its own field **DF2** to **BF2** plus a random number from 1 to 10. Create an instance and verify the value of each.

   Answer: to be able to verify the values of the fields we make them public and we define D2 and B2 as follows:

```
:Class B2
    :field public BF2←90
:EndClass
```

```
:Class D2: B2
    :field public DF2
    ∇ cons
      :Implements constructor
      :Access public
      DF2←BF2+?10
    ∇
:EndClass
```

Then create an instance and verify the values of each field:

```
      i2a←⎕NEW D2 ⋄ i2a.(BF2 DF2)
90 95
```

Note that **DF2** has the same value in each new instance. The random function returns the same result because the []RL is the same[3]:

```
      i2b←⎕NEW D2 ⋄ i2b.(BF2 DF2)
90 95
```

3. Create a class D3 based on B3. B3 has a field **TS** and a constructor which sets it to the current date/time. Verify that instances of D3 have **TS** set by including a public method in D3 that displays **TS**.

   Answer: define B3 and D3 as follows:

```
:Class B3
    :field public TS
    ∇ cons
      :Implements constructor
      :Access public
      TS←⎕TS
    ∇
:EndClass

:Class D3: B3
    ∇ when
      :Access public
      'this instance was created on' TS
    ∇
:EndClass
```

Creating an instance of D3 involves B3's constructor. The instance created will have a **TS** variable.

[3] []RL, like most system variables, has namespace scope. The values of these variables are acquired at definition time from the parent namespace, just like regular functions inherit values from the namespace they're defined in whether they're localized or not.

```
      i3←⎕new D3
      i3.when
 this instance was created on  2009 2 3 21 25 24 96
```

4. Create a class D4 based on B4. B4 has 2 *shared* fields: **LastID** and **TIME** which is set by its constructor to the current time. D4 is to have a field **ID** that will be set by its constructor which also sets the shared field **LastID** in B4 to the same value. That value is specified as an argument at creation time.
   Every time an instance will be created the class will hold the value of the (last) instance's ID and the time when it was created. Create 2 instances and verify that the class holds the latest values.

   Answer: B4 and D4 should look like this:

```
:Class B4
    :field shared public TIME
    :field shared public LastID
    ∇ cons
      :Implements constructor
      :Access public
      TIME←3↑3↓⎕TS
    ∇
:EndClass

:Class D4: B4
    :field ID
    ∇ cons id
      :Access public
      LastID←ID←id
      :Implements constructor
    ∇
:EndClass
```

Creating an instance involves both constructors. First, D4's constructor's is called and the statements are executed until the **:implements constructor** statement at which point B4's constructor is called.

The instances created will "see" the shared fields.

```
      i1←⎕NEW D4 'abc'
      i1.⎕nl-2 3
 LastID  TIME
     i1.(LastID  TIME )
 abc   21 8 0
      i2←⎕NEW D4 'def'
      i2.(LastID  TIME )
 def   21 8 5
```

Just like the base class:

```
      B4.(LastID  TIME )
 def   21 8 5
```

Note that because the field **ID** in D4 has been declared *private* there is no way to access it externally and verify that it has, indeed, been set properly. We will see debugging tricks later to circumvent that problem.

5. Create a class *Animal* which has 2 public fields: **Legs** (an integer) and **Sounds** (a VTV[4]). The constructor will initialize these fields. A method will show the sound it makes.

   Create also a function, <**Zoo**>, which will create a list of different animals and create them all talk at once.

   Answer: here's Animal:

```
:Class Animal
    :field public Sounds
    :field public Legs

    ∇ make(legs sounds)
      :Implements constructor
      :Access public
      Sounds←,⊂⍣(1≡≡sounds)⊢sounds ⍝ turn into a VTV if necessary
      Legs←⌊legs
    ∇
    ∇ r←talk;s
      r←(?⍴s)⊃s←Sounds,(0∊⍴Sounds)/⊂'' ⍝ pick a sound at random
      :Access public
    ∇
:EndClass
```

To create an instance of an *Animal* we MUST supply 2 elements: the number of legs and a series of sounds (strings) the animal emits. Any other argument will be refused.

The function <**Zoo**> could then look like this:

```
     ∇ list←Zoo;tiger;monkey;snake;fish
      ⍝ This fn builds a list of animal instances
       tiger ←⎕NEW Animal(4,⊂'Grrr' 'WOOAOH')
       monkey←⎕NEW Animal(2,⊂'ooooo' 'waaaa' 'ee-ee-hee')
       snake ←⎕NEW Animal(0 'sssss')
       fish  ←⎕NEW Animal(0,⊂0/⊂'') ⍝ list of sounds is empty

       list←tiger,fish,snake,monkey
     ∇
```

We can assign the result to a variable and make the animals all "talk" at once:

[4] VTV: Vector of Text Vectors

```
      ⎕←an← Zoo
#.[Animal]  #.[Animal]  #.[Animal]  #.[Animal]
      an.talk
 Grrr    sssss  waaaa
      ρ⎕←an.talk
 WOOAOH    sssss  ee-ee-hee
4
     ⎕nc 'an'
2
     an.Legs
4 0 0 2
```

‘**an’** is a variable holding 4 instances.

6. Create a *Dog* class based on *Animal* that needs an argument representing the type of dog to build an instance of. The type will be stored in a field.

   Create an instance of a ‘canis familiaris’ and verify it has all its limbs.

   Answer:

```
:Class Dog: Animal
    :field public Type

    ∇ make typ
      :Access public
      :Implements constructor :base   4 ('woof' 'wa-wa-wa')
      Type←typ
    ∇
:EndClass
```

Tracing through the following statement you can see that the *Dog* constructor is called first. Then, on the 2nd line, the *Animal* constructor is called with the 2 values after the :**Base** keyword. No other argument can be supplied after :**base** as this is the only acceptable argument the *Animal* class will accept.

```
      d1←⎕new Dog 'canis familiaris'
```

And of course we get the expected behaviour, we can make it ‘**talk’**, it has **Legs**, **Sounds** and **Type**:

```
      d1.⎕nl-2 3
 talk  Legs  Sounds  Type
      d1.talk
woof
      d1.(Legs Type) ⍝ how many legs? what type?
4  canis familiaris
```

7. Create a dog breed Collie based on the Dog class which will require a name then create 2 pets named ‘Rex’ and ‘Fido’ from it.

   Answer:

```
:Class Collie: Dog
    :field public Name

    ∇ adopt me
      :Implements constructor :base 'canis familiaris'
      :Access public
      Name←me
    ∇
:EndClass
```

   We can create the instances in 1 or 2 statements. In 1 it looks like this:

```
      pets←{⎕new Collie ⍵}¨'Rex' 'Fido'
      pets.Name
 Rex   Fido
      pets.talk
 woof   woof
     pets.⎕nl-2 3
  Legs   Name   Sounds   Type     talk
```

As in problem 5, **pets** is an array, this time of Collies. Each instance has inherited all the fields from all the underlying base classes.

8. Create a class *Species* based on animal that takes as argument the scientific name of the animal, the number of legs and the sound it makes and stores the scientific name permanently (read only) in the new instance. A property, Sname, will return this value.

   We will store the name in a variable in the instance and the property will simply return it. The legs and sounds will be passed no to the *Animal* class for building.

   Answer:

```
:Class Species: Animal

    ∇ new (sn legs sounds)
      :Access public
      :Implements constructor :base legs sounds
      SN←sn
    ∇

    :property Sname
    :access public
        ∇ r←get
          r←SN
        ∇
    :endproperty
:EndClass

      duck←⎕NEW Species('Anas' 2 'qwack-qwack')
      duck.⎕nl-2
 Legs  Sname  Sounds
     duck.Sname
Anas
     duck.Sname←'anatis'
SYNTAX ERROR
     duck.Sname←'anatis'
     ^
```

As seen above, the name cannot be changed, it is set permanently.

Accessing shadowed base members

Occasionally a public member will be hidden from view because it has been redefined by the derived class. If the member was public there is still a way to get at it.

The first way is externally using the dyadic form of []CLASS recasting the instance to the base class to access the member.

The second way is internally, using the system function []BASE to reference the member from within the instance.

These are two totally independent things.

Let’s see both. For this we will use the following class:

```
:Class xbase
    ∇ r←lengthyCalc x
      :Access public
      r←'complex process here...'{⍵}x*2
    ∇
:EndClass
```

Dyadic []CLASS

This system function allows us to reach a base member from an instance.

Consider a class based on **xbase** with a slightly different method of the same name

```
:Class xderived: xbase
    ∇ r←lengthyCalc arg ⍝ overwrite previous method
      :Access public
      r←arg*3
    ∇
:EndClass
```

Let’s create an instance:

```
      xd←⎕new xderived
     (xbase ⎕class xd).lengthyCalc 5 ⍝ the square (not the cube)
25
```

[]BASE

This system variable is a reference to the base class. It allows you to access any public member in it using the dot syntax.

Consider a class based on **xbase** with a slightly different method of the same name

```
:Class xderived2: xbase
    ∇ r←lengthyCalc arg ⍝ overwrite previous method
      :Access public
      r←2×⎕BASE.lengthyCalc arg
    ∇
:EndClass
```

Because the name <**lengthyCalc**> is in use we cannot have direct access to the function and we must use []BASE.

```
      (⎕NEW xderived2).lengthyCalc 3
18
```

It is easier to reuse the code in the base class than to rewrite it, not to mention the fact that the base class' code may get out of sync if you copy the code instead.

Solved Problems

1. Create a class with a public a property **time** returning the current time and another class based on it which has a field **time** and a method making use of the **time** property in the base class

Answer:

```
:Class cw1p
    :property time
    :access public shared
        ∇ r←get
          r←100⊥⎕TS[4 5]
        ∇
    :endproperty
:EndClass

:Class teatime: cw1p
    ∇ time←tt        ⍝ 'time' is local and shadows 'time' in 'cw1p'
      time←'not now.'
      :Access public shared
      :If 1500 1545≠.<⎕BASE.time ⋄ time←'It''s tea time!' ⋄ :EndIf
    ∇
:EndClass
```

2. Create an instance of the derived class and use it to call **time** in the base class

Answer:

```
      tea← ⎕new teatime
      tea.tt
not now.
      (cw1p ⎕class tea).time
1627
```

Section 6 – Inheritance of GUI and .Net classes

Dyalog APL already comes with namespace based GUI objects, effectively classes of their own.

For example a form is a built-in class from which instances are created when doing

```
      ‘frm’  ⎕WC  ‘form’
```

and ‘**frm**’ becomes an *instance* of a **form**.

Dyalog has been extended to use the new []NEW syntax and doing

```
      frm ← ⎕NEW 'form' (⊂'Caption' 'dsa')
```

will work just the same. The syntax is slightly different and more stringent but the end result is the same.

It is possible to create classes based on GUI classes and others like .Net classes. For example OLEClients like Excel ® or PowerPoint are possible as well as .Net classes. Like this:

```
      XL←⎕new 'oleclient' (⊂'classname' 'Excel.application')
      ⎕using←''
      present ← System.DateTime.Now
```

It is necessary to set []USING before attempting to use .Net classes. []USING tells APL where to look when using .Net classes.

Solved Problems

1. Write a class Dialog that creates a form preformatted with 2 buttons, **OK** and **Cancel** to the bottom left and takes arguments to set the caption and the size. The **Close** button will HAVE to be used to exit. Use the class to create an instance that has a specific caption and size.

Answer:

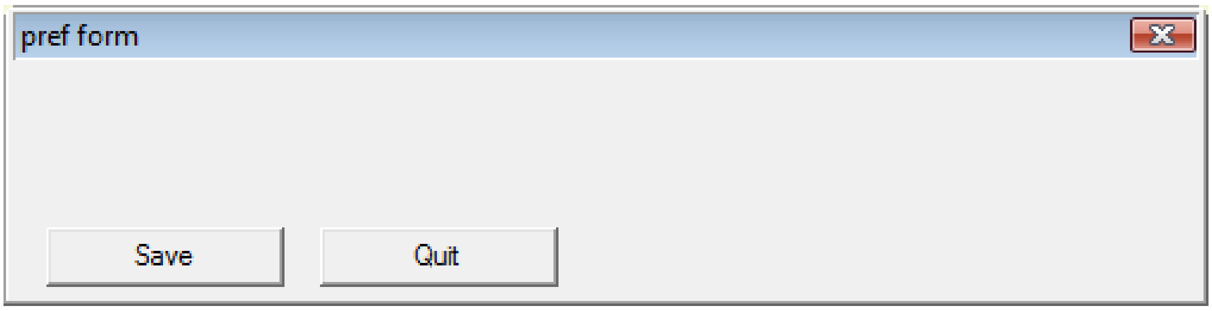
The form will be based on a *form*, some of its properties are defaulted. The close button is disabled and the callback function for the Close button will be made to simply close the form.

```
:Class Dialog : 'form'    ⍝ Standard dialogue form with 2 buttons
      :field public Save
      :field public Quit

    ∇ Init (caption size props) ;dpr;pr;ps;pq
      :Access public

    ⍝ Default values:
      dpr←('coord' 'pixel') ('edgestyle' 'Dialog') ('border' 2)

    ⍝'props' is a list as above or pairs of strings which we enclose
```

```
      dpr,←('caption' caption) ('size' size),⊂⍣(2≡≡props)⊢props

    ⍝ Call form constructor here
      :Implements constructor :base dpr
      onClose←¯1 ⍝ disable Close X title button
      pr←('size' (25 100)) ('attach' (4⍴'bottom' 'left'))
      (ps pq)←(size[1]-30),¨15 130 ⍝ position properly
      Save←⎕NEW 'button' (('caption' 'Save') ('posn' ps),pr)
      Quit←⎕NEW 'button' (('caption' 'Quit') ('posn' pq),pr)
      Quit.onSelect←'doQuit'
    ∇

    ∇ doQuit dum
      Close  ⍝ do anything special here
    ∇
:EndClass
```

Here is an instance which has the caption 'pref form' and a size of 100 by 500:

```
     f1←⎕new  Dialog ('pref form' (100 500) ⍬)
```



2. Create an EditField class to accept text input based on *edit* that will include a *label* field to its left. The class will need the label's text, the input position and size and any other parameter deemed necessary (e.g. 'multi-line')

Answer:

```
:Class EditField  : 'edit'
    ∇ make (t p s x)
      :Access public
      :Implements constructor :base  ('posn' p)('size' s),x
      Label←##.⎕NEW 'text'(('text' t)('points'(p+3 ¯3))('halign' 2))
    ∇
:EndClass
```

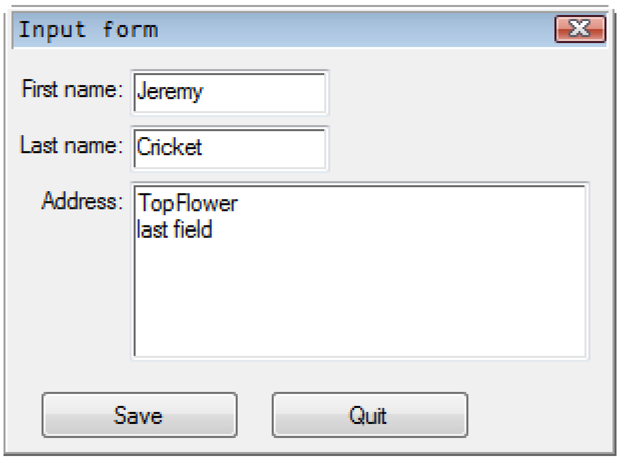
3. Create a form from the Dialog class using 3 instances of the EditField class.

Answer:

```
     f1←⎕new Dialog ('Input form' (200 300) ⍬)
     f1.f←f1.⎕NEW EditField ('First name:'(10 60) (⍬ 100) ⍬)
```

```
      f1.l←f1.⎕new EditField ('Last name:' (38 60) (⍬ 100) ⍬)
      f1.a←f1.⎕new EditField ('Address:'   (66 60) (90 230)
                                                   (⊂'style' 'multi'))
      f1.(f l a).Text←'Jeremy' 'Cricket' ('TopFlower' 'last field')
```

Here's what it looks like:



First name: Jeremy
Last name: Cricket
Address: TopFlower
last field

.Net based classes

Classes can also be based on .Net classes. For example, .Net has classes to manipulate forms and APL can also create forms based on them. A class to create a form similar to the 'Dialog' class above could look like this:

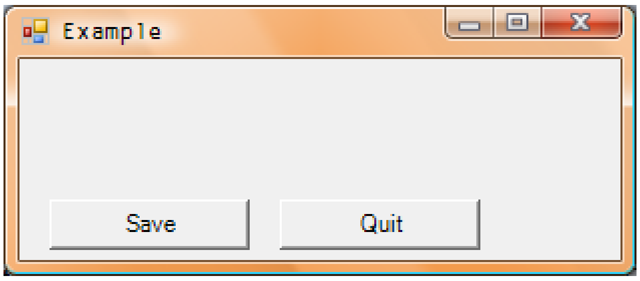
```
:class DNDialog: Form

:USING System.Windows.Forms,system.windows.forms.dll
:USING System.Drawing,system.drawing.dll
:USING ,System.dll

    ∇ make (title size);ps;pq;sq
      :Access public
      :Implements constructor
      ClientSize←⎕NEW System.Drawing.Size size
      Text←title ⋄ Visible←1
      sq←(Save Quit)←⎕NEW¨2⍴Button
      sq.Text←'Save' 'Quit'
      sq.Size←⎕NEW System.Drawing.Size (100 25)
      (ps pq)←15 130,¨ size[2]-30
      Save.Location←⎕NEW System.Drawing.Point ps
      Quit.Location←⎕NEW System.Drawing.Point pq
      Controls.Add¨sq
      Quit.onClick←'doQuit'
    ∇
    ∇ doQuit dum
      Close
    ∇
:endclass
```

Such classes are not only intended for external use, from other .Net languages but also from Dyalog APL itself. To test the class or before exporting it for good you must use the '*export to memory*' menu item. This will prepare the class for .Net and make it usable for testing. But it will only last until the workspace is cleared.

Once done the class can be instantiated like this (no need to assign the result):

```
     ⎕new DNDialog ('Example' (300 100))
```

will produce



Because the .Net framework does not close the form when it "goes out of scope" you don't need to assign the instance but you need to call the Close method to close it (it's exactly the same as in C#).

If Class A needs class B. It should refer to it as #.B but because .Net has noted its presence (through "export to memory") it "knows" where to find it.

If you want to instantiate a .Net derived class you need to either put in a *:signature* statement to describe the function's syntax or enclose the argument because the default for .Net is 'object'.

The *:signature* statement's general syntax is:

```
:Signature [RESTYPE←] fnname [TYPE1 [argname] [, TYPEn [name]]]
```

where RESTYPE is the result's type (default Void) and TYPEx is the corresponding argument's type. The names of the function and arguments names need not be the same and can even be ignored for the arguments. But a comma must separate the arguments.

There are various **.Net** types all residing in *System*. The more interesting ones are **Int32, Double, String** possibly followed by **[ ]**, once per rank.

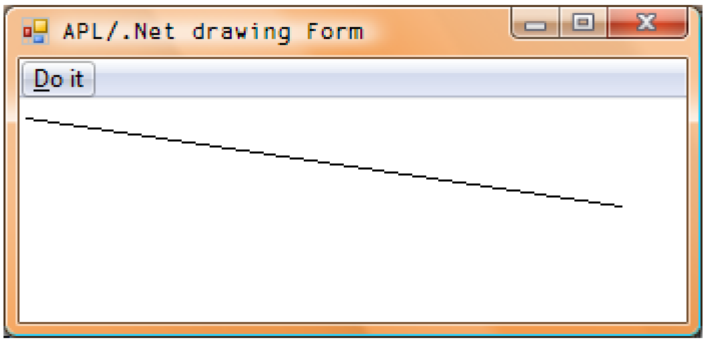
Here is an example. This class creates a drawing form on which a line is drawn when the only menu item is selected. The constructor takes a pair of integers as argument to describe the size of the form. To tell **.Net** external users about this we use the :*signature* statement with **System.Int32 [ ]** which is .Net's definition of a 32 bit integer (the **[ ]**s mean 'array'):

```
:class drawingForm: Form

:USING System.Windows.Forms,system.windows.forms.dll
:USING System.Drawing,system.drawing.dll
:USING ,System.dll

⍝ This class will create a form to display a line using a menu

    ⎕io←0 ⋄ ⎕ml←1 ⋄ ⎕rl←60⊥⌽3↓⎕ts ⋄ bmp←0
    Title←'APL/.Net drawing Form'

    ∇ make size
      :Access public
      :Implements constructor
     :Signature make System.Int32[] size
    ⍝ Prepare the instance
    ⍝ Use a container for the menus
      components←⎕NEW System.ComponentModel.Container
      mainMenu1←⎕NEW MainMenu components
      mnu1←⎕NEW MenuItem

    ⍝ Main menu
      mainMenu1.MenuItems.AddRange⊂,mnu1 ⍝ 1st level
      mnu1.(Text onSelect)←'&Do it' 'mDoit'
      BackColor←Color.White
      ClientSize←⎕NEW System.Drawing.Size size
      Menu←mainMenu1 ⋄ Name←'MainForm' ⋄ Text←Title
      onPaint←'Form_Paint' ⋄ Visible←1
    ∇

⍝ The onClick fns

    ∇ mDoit;sz;t;gScreen;gBitmap
    ⍝ Performs the calculations and displays the graphics
      gScreen←CreateGraphics
    ⍝ Get the form's client size.
      sz←ClientSize.(Width Height)
    ⍝ The shadow bitmap - same size as the current form's area.
      bmp←⎕NEW Bitmap sz
    ⍝ Graphics object for the bitmap.
      gBitmap←Graphics.FromImage(bmp)
      t←Pens.Black,?10 10,sz-10
      gScreen.DrawLine t     ⍝ draw on the screen
      gBitmap.DrawLine t     ⍝ draw on 'bmp'
    ∇

    ∇ Form_Paint (s e)
      :If bmp≢0 ⋄ e.Graphics.DrawImage (bmp 0 0) ⋄ :EndIf
    ∇

:endclass
```

Doing

```
      ⎕NEW drawingForm (⊂333 111)
```

and clicking the only menu item will produce something like this:



.Net types are all in namespace **System**.

:Using is used to tell where to find types. Here the statement

```
:USING ,System.dll
```

tells APL to look at the top level (nothing before the comma) in file <System.dll>.

Solved problems

1. Write a method for .Net external users that takes a floating point number as argument.

   Floating point numbers are **Double**s in .Net.

   An example of a such a method could be

```
      :USING System
    ∇ InitializeComponent argument
      :Signature fnname Double
      ...
```

2. Write a method for .Net external users that takes a matrix of integers and returns a character string.

Example:

```
      :USING System
```

```
    ∇  Z←fnX arg
      :Signature String←fnname Int32[][] arg
      ...
```

Section 7 – Properties

A property is a special class member, intermediate between a *field* and a *method*. It is read and assigned like a *field* but uses a pair of functions to GET and SET its value. The *field*-like syntax is easier to read and write than lots of method calls and this interposition of method calls allows for data validation, active updating and/or read-only *'fields'*

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.

There are 3 kinds of properties *simple*, *numbered* and *keyed*. They differ mainly in the way their indices are treated. To tell APL which one we are using we use a special keyword between :*Property* and its name. For example, to define a *keyed* property KP you would write

```
      :Property  keyed  KP
```

When properties are assigned a value the argument to the <SET> function is a namespace containing a minimum of 2 things: the name of the property (in variable 'Name') and the new value assigned (in variable 'NewValue'). In cases *numbered* and *keyed* they also include the indices (in variable 'Indexers').

Simple properties

By default (if the special keyword *simple* is not supplied) a property is deemed *simple*, that is it is treated as a whole when you ask for it.

It can be indexed just like any other variable. For example:

```
      instance.propertyX[3]
```

APL first gets the property then it extracts the 3rd element. If the property is simple enough this is fine. But if the property needs to do a lot of work this may be a problem. For example, if the property returns ALL the records of a database, fetching every record to keep only the 3rd one isn't very efficient.

Likewise,

```
      instance.propertyX[3] ← 321
```

requires that APL first fetches the entire property using the GET function to find if the assignment makes sense before it can be made. If all goes well, the assigned value is inserted in GET's result and the whole thing is given back to <SET> in the 'NewValue' variable in the namespace given as argument.

Example: record monthly data providing only partial monthly info

```
:Class MonthlyData
    ⎕io←1
    ∇ boa (data months)
      :Implements constructor
      :Access public
      OBS←12⍴0             ⍝ make room for all data
      OBS[months]←data     ⍝ insert data in months’ slots
    ∇
    :property simple AllYear
    :access public
        ∇ r←get
          r←OBS ⍝ retrieve all 12 months
        ∇
        ∇ set x
          OBS←x.NewValue
        ∇
    :endproperty
:EndClass
```

We can use this class this way:

```
      md←⎕new MonthlyData((11 22 33)(2 3 5)) ⍝ data for months 2 3 5
      md.AllYear
0 11 22 0 33 0 0 0 0 0 0 0
      md.AllYear[7]← 123        ⍝ assign value 123 to month 7
      md.AllYear
0 11 22 0 33 0 123 0 0 0 0 0
      (1↑md.AllYear)←21
      md.AllYear
21 11 22 0 33 0 123 0 0 0 0 0
```

As can be seen, structural assignment is also possible using *simple* properties.

Note that APL will convert the indices if the index origin (⎕IO) in the class is different than the calling environment thus freeing you from having to worry or even know which IO the class is using.

Solved problems

1. Create a class similar to ‘MonthlyData’ that will record **sparse**[5] yearly data for years 2001 to 2015.

Answer: To minimise space a sparse structure is used. We separate data from indices.

```
:Class YearlyData
    ⎕io←1

    ∇ plan (data years)
      :Implements constructor
      :Access public
```

[5] Sparse data is data for which a high percentage is unknown or has a specific value like 0. In that case tricks are used to save space, for example by keeping the indices paired with the data. For the sake of the example we keep the number of years small but in practice it could be for millions of observations using similar principles.

```
      (OBS YEARS)←,¨data years
    ∇

    :property AllYears[6]
    :access public
        ∇ r←get
          r←(OBS,0)[YEARSι2000+ι15] ⍝ retrieve all years
        ∇
    :endproperty

    ∇ year YearIs value
      :Access public
      YEARS∪←year             ⍝ add missing years
      OBS←(ρYEARS)↑OBS        ⍝ add missing space
      OBS[YEARSιyear]←value   ⍝ (re)set years' data
    ∇
:EndClass:EndClass
```

Using indices is now meaningless and we must manage setting values differently. For this we use a pair of values: data and years.

```
      md←⎕new YearlyData ((21 32 54)(2002 2003 2015))
      md.AllYears
0 21 32 0 0 0 0 0 0 0 0 0 0 0 54
      md.AllYears[5]←99
SYNTAX ERROR …
```

There is no <SET> function and the index is meaningless anyway. We use the <YearIs> method to set years instead:

```
      2010 md.YearIs 99
      md.AllYears
0 21 32 0 0 0 0 0 0 99 0 0 0 0 54
```

2. Create 2 instances of ‘YearlyData’ as a vector and use the ‘AllYears’ property to display all the data at once.

Answer: We first create the vector using 2 sets of data:

```
      data←((21 33 12)(2002 2003 2005)) ((98 34)(2001 2011))
       vi←{⎕NEW YearlyData ω}¨data
```

we then use disclose (⎕ML>1) to form a 2 x 12 matrix:

```
      ⊃ vi.AllYears
```

[6] Note we did not write the ‘*simple*’ keyword, this is the default

```
 0 21 33 0 12 0 0 0 0 0  0 0 0 0 0
98  0  0 0  0 0 0 0 0 0 34 0 0 0 0
```

Numbered properties

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, and only those, that need to be retrieved or changed.
Let’s revisit the **MonthlyData** class, making it sparse, like **YearlyData**:

```
:Class NumMonthlyData
    ⎕io←1

    ∇ boa (data months)
      :Implements constructor
      :Access public
      (OBS MONTHS)←,¨data months
    ∇

    :property numbered AllYear
    :access public
        ∇ r←shape
          r←12
        ∇
        ∇ r←get x
          r←(OBS,0)[MONTHS⍳x.Indexers] ⍝ retrieve a specific month
        ∇
        ∇ set x;m
          MONTHS∪←m←x.Indexers
          OBS←(⍴MONTHS)↑OBS
          OBS[MONTHS⍳m]←x.NewValue
        ∇
    :endproperty
:EndClass
```

In order for APL to know the property’s rank and index limits, a function, SHAPE, must be defined accordingly. Here it returns a constant, 12, but in practice it could be a variable result like the limits of a component file. With this information in hand APL is in a position to do any structural assignment.

The difference, here, is that APL makes a call to the GET or SET functions once per index. In cases where the property is big and used mostly with indices this could be a time and space saver. Tracing through the following expressions will show more clearly what happens:

```
       nm←⎕NEW NumMonthlyData ((321 234 543 456)(2 3 5 7))
       nm.AllYear ⍝ GET function called 12 times
0 321 234 0 543 0 456 0 0 0 0 0
       (3↑nm.AllYear)←444 555 666 ⍝ SET function called 3 times
```

```
      nm.AllYear
444 555 666 0 543 0 456 0 0 0 0 0
```

This concept of *numbered* property is only in Dyalog APL, no other OO language supports it yet.

Solved problems

1. Modify the GET function in `NumMonthlyData` above to output the Indexer value each time it is called and verify it is in the range 1-12.

   Answer:

   ```
   ∇ r←get x
     ⎕signal 3⍴⍨0 12=.≥⎕←x.Indexers ⍝ verify range and show index
     r←(OBS,0)[MONTHS⍳x.Indexers]   ⍝ at every iteration
   ∇
   ```

Keyed properties

Keyed properties always use a key index which can be anything. Often this key is a string as with Excel's Sheet property which uses either an integer or a string as key to identify the sheet desired:

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

Here `⊂'Sheet2'` identifies exactly ONE sheet (it is a scalar). Had it been not enclosed, each of the 6 characters would have denoted a different sheet and would have likely failed.

When the property's **GET/SET** functions are called their argument contains **Indexers** which is a vector of N enclosed vectors, one per rank as specified between the indexing brackets. For example, if you enter *inst.prop[A;B]*, **Indexers** will be (A B), whatever A and B may be. APL will only ensure that the result or the assigned value has the shape (⍴A),⍴B.

To better understand this let's have another look at 'MonthlyData':

```
:Class KeyedMonthlyData
    OBS←12⍴0
    MONTHS←↓12 3⍴'JanFebMarAprMayJunJulAugSepOctNovDec'
    ∇ boa (data months)
      :Implements constructor
      :Access public
      OBS[MONTHS⍳months]←data
    ∇
    :property keyed AllYear
    :access public
        ∇ r←get x
          r←OBS[MONTHS⍳⎕IO⊃x.Indexers] ⍝ retrieve specific months
        ∇
        ∇ set x
```

```
          OBS[MONTHS⍳⎕IO⊃x.Indexers]←x.NewValue
        ∇
    :endproperty
:EndClass
```

This version deals with names instead of indices. To use it you could do

```
      z←⎕new KeyedMonthlyData ((11 22 33) ('Feb' 'Mar' 'May'))
      z.AllYear['Feb' 'Mar' 'Jan']
11 22 0
      z.AllYear['Aug' 'Jan']←99 101
      z.AllYear[⊂'Jan']
101
```

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 passes the indices unchanged - the indices will have to be provided in the index origin of the class.

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 ref, to the instance.

(This document is work in progress, other sections will eventually follow)