Module2: Methods and Events

Objects can be brought to life by the built-in functionality that is intrinsic to the object (methods and default event processing) or by arbitrary user defined functions (callbacks) which are assigned to events in such a way as to run whenever the particular event occurs as a result of some user interaction with the object.

§ 2.1 Object Methods

A method is essentially a type of event that can only be generated under program control. A method may perform an operation and may return a result.

§§ 2.1.1 Enqueuing Object Methods with ⎕NQ

```
      2 ⎕NQ CVec_Arr                    ⍝ Enqueues method given by Arr for object CVec
```

⎕NQ adds the method specified in Arr to the end of the Windows *event queue*. The method will subsequently be processed when it reaches the front of the queue. If the method returns a result then it is returned by ⎕NQ as a shy result.

2.1.1.1 Investigate the methods associated with a Form by way of the MethodList property or the)METHODS system command, and try

```
      2 ⎕NQ(⊂'MyForm'),⊂'ChooseFont'
```

§§ 2.1.2 Invoking Event default Action using ⎕NQ

The default action of an event can be invoked by calling the event as a method.

2.1.2.1 Investigate the events associated with a Form by way of the EventList property or the)EVENTS system command and try

```
      2 ⎕NQ'MyForm','Configure' 10 10 20 20
```

This has the same effect as moving the Form to Posn 10 10 and resizing it to Size 20 20 but without invoking any associated callback function on the Configure event of the Form.

The unnecessary (through lenience) catenations and enclosures in the above right arguments for ⎕NQ are included to emphasise the separation of the object name and method calling information. The forgiving nature of ⎕NQ syntax (like that of ⎕WC) can obscure the essential components of the arguments. At this point DISPLAY.DWS or varChar.exe might help to clarify some details of strand notation. Also note that it can be useful to put system commands onto programmable function keys with ⎕PFKEY.

§§ 2.1.3 Method Functions

When ⎕WX is set to 1 then not only are property names exposed and become immediately accessible as pseudo-variables, but also method and event names are exposed and become pseudo-functions. Thus

```
      ⎕CS'MyForm'⎕WC'Form'
      Configure 10 10 20 20
```

will configure the Form as above.

2.1.3.1 Experiment with mouse events on a Button.

```
      ⎕CS'MyButton'⎕WC'Button'
      MouseDown 50 50 1 0
```

```
      MouseUp 50 50 1 0
```

Notice that when you are inside MyForm the `Close` event does make the `Form` disappear, but the remaining vanilla namespace called MyForm does not disappear until you exit the space.

§ 2.2 Object Events and callback Functions

An event may occur as a consequence of a user action. The user action triggers a default behaviour or a programmer-defined process.

§§ 2.2.1 Firing Events by User Actions

2.2.1.1 Create a `Form` with a `Calendar` on it.

```
      'F'⎕WC'Form' ⋄ 'F.C'⎕WC'Calendar'
```

Hit the right and left cursor keys to move the highlit day back and forth. Given that

```
      ⎕KL'RC'↳'Right'
```

experiment in the F namespace with

```
      2 ⎕NQ'C' 'KeyPress' 'RC'
      ⎕NQ'C' 22 'RC'
```

and in the C namespace with

```
      KeyPress 'RC'
```

On the one hand the event may be triggered by user action, on the other hand it may be generated under program control.

2.2.1.2 Use the `GetEnvironment` property of the Root object (or alternatively use REGEDIT.EXE) to discover the name of the APL input (.DIN) file. Open that file in Notepad and view the key labels section. Which of these could apply to the `Calendar`? Look at the `EventList` for the `Calendar` and guess which events are triggered by which keystrokes. Check whether some do what you expect.

§§ 2.2.2 Attaching callback Functions to Events

Functions may be associated with events by way of the `Event` property. The `Event` property is very flexible and tolerant (lenient) in its assignment. We shall focus on the simplest forms of assignment and leave the more complicated forms aside.

In its simplest form, the `Event` property expects a 2-element vector of character vectors, the first being the name of the particular event and the second being the name of the callback function. (View the default content of this property in varChar.) For example, in a clear workspace,

```
      ⎕CS'F'⎕WC'Form'
      Event←'MouseMove' 'moveForm'
```

where we define the callback function simply as

```
      ∇moveForm Msg
    [1] Posn←Posn+Msg[3 4]÷100∇
```

The Rarg of every callback function is supplied automatically, if required, by APL and its content is determined by the event in question. In every case the first element supplied contains the object in question and the second element contains the event in question. In the case of a `MouseMove` event, for example, the definition of the remaining 4 elements may be found in [Help][GUI Help]. In particular, the third and fourth elements of the event message refer to the coordinates of the position of the mouse.

2.2.2.1 Move the mouse over the `Form` and explain the effect. Put a stop on the first line of the callback and investigate the elements of the incoming right argument.

The `Event` property may be assigned to more than one event. Each assignment only affects the events named in the assignment. All other events retain their current settings.

Event names may be prefixed with "`on`" to give a new set of properties that can obviate the need for the `Event` property, and make it redundant.

```
      onMouseMove←'moveForm'
```

results in behaviour very much like that produced by setting the `Event` property, but with one subtle difference in the message right argument automatically supplied to the callback.

2.2.2.2 Can you determine the difference? Use varChar if necessary.

Callback functions may return a result. If there is no result, or the result is 1, then APL proceeds to process the default action for the event. If the result is 0 the default processing does not take place. If the callback function modifies the incoming argument and returns that modified message as the result of the callback function then APL (leniently) processes default action in accordance with that new message. APL is even surprisingly lenient with a would-be `VALUE ERROR` result, and a niladic callback.

§§ 2.2.3 Bringing Objects to Life

Now we have all the ingredients necessary to breathe life into objects and superobjects built by a superposition of primitive Dyalog GUI objects.

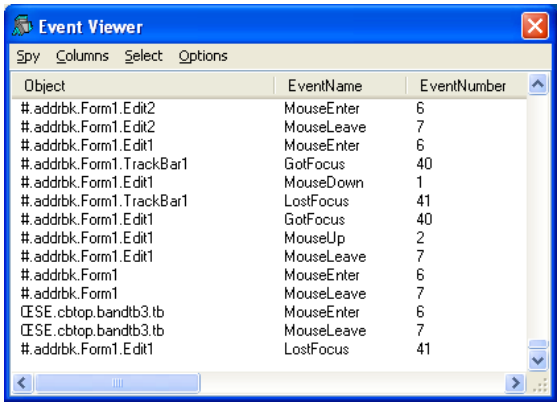
There is a poorly documented but very useful event called `All` that is recognised by every primitive GUI object. This event can be used to associate a callback function with *all* the events relating to an object. Write a trivial callback that just prints its message argument in the Session. Assign it to the `All` event.

```
      ∇ show Msg
[1]     Msg
      ∇
      onAll←'show'
```

Now use the mouse and keyboard to interact with the `Form`. Doing this you will discover many ways in which you can programmatically intervene in the default behaviour of a `Form` using callbacks. This technique is a good way to explore the event structure of any GUI object.

In version 10.1, under the [Tools][Spy] menu, there is a new EventViewer that allows you to spy on events in a similar manner to the above, but with much more control through many new features.

Compare this with the EventViewer in [Control Panel][Administrative Tools].



§ 2.3 The Event Queue

§§ 2.3.1 Dequeuing Events with and without `⎕DQ`

Given the elements above it would be possible to write an APL cover function to control the activities and events associated with collections of GUI objects. However, as the complexities of superobjects grow,

and the numbers of events multiply, and callbacks begin to take noticeable amounts of time to run, the administration of events waiting to be processed becomes a significant problem.

In Immediate Execution Mode (IEM) events are processed as they arrive in the queue and the need for further control might seem unnecessary. However, IEM is *not* available in runtime systems and an alternative then becomes essential.

`⎕DQ CVec`	⍝ Dequeues events associated with object `CVec`

`⎕DQ` takes the name of a top-level object as its argument and administers its events plus any events from subobjects (child objects). (`⎕DQ` may also take a vector of top-level names.)

```
'F'⎕WC'Form'
⎕DQ'F'
```

2.3.1.1 Write a callback on the `Configure` event of `Form` F which resizes the `Calendar` to fit the `Form` exactly whenever the `Form` is resized. What is the difference between running in IEM and under `⎕DQ`?

The default value for an event is zero. So, if the `Close` event for a `Form` has not been set to a callback function or any other action code then `onClose↳0`. Zero means handle the event normally. If the event action code is set to `¯1` then the event is entirely ignored.

2.3.1.2 Investigate the consequence of

```
onClose←¯1
```

or, alternatively,

```
Event←'Close' ¯1
```

both in IEM and under `⎕DQ`.

Note that the way to terminate `⎕DQ` as a programmer developing a system is either by the keyboard **Ctrl+Break**, or by the Session menu [Action][Interrupt], or via the SysTray APL icon - select [Weak Interrupt] or [Strong Interrupt].

Within the program itself, one way to terminate `⎕DQ` is via action code `1`.

```
onMouseMove←1
```

This causes `⎕DQ` to terminate when the mouse is moved over the `Form`.

§§ 2.3.2 Tracing `⎕DQ`

`⎕DQ'F'` is just like the "immediate" default processing of the Session with one very significant difference - **YOU CAN TRACE INTO IT!** It is possible to put a `⎕STOP` on a callback that causes it to suspend when run, but tracing into `⎕DQ` is a much more powerful means of debugging applications. (I like [Options][Configure][Trace/Edit] with Classic Dyalog mode and Independent trace stack checked.)

2.3.2.1 In the Root space, write a function (as in §§ 2.2.3) called `∇show∇` which simply displays its right argument. Then create a `Form` with some event or events set to call `∇show∇`. For example,

```
'F'⎕WC'Form' ('Event' 'All' 'show')
```

Then trace `⎕DQ` by typing

```
⎕DQ'F'
```

and hitting **Ctrl+Enter**. Tracing `⎕DQ'F'` allows you to see all the callback code that is running - this is an important ingredient in debugging GUI applications.

`{Msg}←⎕DQ CVec`	⍝ Dequeues events of `CVec` and returns a message

It is important to know how to terminate ⎕DQ under program control. This is especially important for time-consuming processes that the user might want to terminate early. Controlled interruption can be achieved in a two-step arrangement. First set some arbitrary event on the object to action code 1.

```
'F'⎕WS'Event' 501 1
```

⎕NQing an event with action code 1 just before a ⎕DQ can allow you to pass through a ⎕DQ in a loop while waiting for some other event. Then analysing the result of ⎕DQ allows one to continue, or not.

2.3.2.2 By placing line

```
⎕NQ'F' 501
```

in a callback function, show that this will terminate ⎕DQ as soon as the enqueued event reaches the top of the stack of events to be dequeued. (Tracing is not usually fast enough to both enqueue and dequeue.)

2.3.2.3 Show that it is possible to monitor some arbitrarily complex looping process in a function such as that below while still displaying a GUI via ⎕DQ

```
      ∇ process;I;Sink
[1]     'F'⎕WC'Form'('Event' 'Close' 1)('Event' 501 1)
[2]     I←0
[3]    Loop:→End×⍳10=I←I+1
[4]     ⎕NQ'F' 501          ⍝ Enqueue event 501
[5]     ⎕←R←⎕DQ'F'          ⍝ Dequeue event at top of queue
[6]     →(501≠2⊃R)/0        ⍝ Was that event a 501 or a Close?
[7]     Sink←⊞⊞⊞⊞⊞⊞⊞⊞⊞?200 200⍴10000 ⍝ Good on a 3.19GHz machine
[8]                         ⍝ ProgressBar object in here?
[9]     →Loop
[10]   End:
      ∇
```

Why does tracing through line [4] not have the desired effect? (Instead put a stop on line [6] and run.)

2.3.2.4 Trace ⎕DQ'⎕SE'.

Immediate Execution Mode makes most objects alive and usable - but a few objects actually need ⎕DQ to give them life. For example, a MsgBox requires ⎕DQ to make it visible.

```
'MBX'⎕WC'MsgBox'('Style' 'Info')('Caption' 'Info')('Text' 'Msg')
```


```
⎕DQ'MBX'
```

2.3.2.5 In [Help][GUI Help], investigate the Wait method and compare the objects to which it applies.

§§ 2.3.3 Defining complex Behaviour

You now have all the ingredients necessary to write complex GUI applications that call arbitrary APL functions as a result of user actions. These functions can modify the GUI itself or create new GUI objects and pass user control from one object to another. Essentially, all the visible GUIs and functionality to be found in Microsoft Office applications can now be reproduced in Dyalog APL!

It is quite a conceptual leap from traditional linear programming to object-oriented programming, but you might not have that baggage...

A GUI object on your screen is bristling with hotspots that are ready to spring into action and might potentially change the virtual or indeed the real world almost beyond recognition.

2.3.3.1 Below is a function that creates a *Form* with a number of controls on it, and associates a number of (undefined) callback functions with these controls. Read the function and consider various alternative ways in which this function might be written. Discuss with your partner good practices and what the callback functions might do in a completed application.

```
      ∇ MAKE_Form
[1]     ⎕CS'Form1'⎕WC'Form' 'Address Book'(60 268)(266 238)↵
      ('Coord' 'Pixel')('Event'('Close' 1))¶
[2]     'Label2'⎕WC'Label' 'Name:'(0 8)(24 32)↵
      ('Attach'('Top' 'Left' 'Top' 'Left'))¶
[3]     'Edit1'⎕WC'Edit' ''(0 48)(24 184)↵
      ('Attach'('Top' 'Left' 'Top' 'Right'))('FCol'(0 0 192))↵
      ('Event'('KeyPress' 'EditKeyPress'))¶
[4]     'Label3'⎕WC'Label' 'E-mail:'(24 8)(24 32)↵
      ('Attach'('Top' 'Left' 'Top' 'Left'))¶
[5]     'Edit2'⎕WC'Edit' ''(24 48)(24 184)  ↵
      ('Attach'('Top' 'Left' 'Top' 'Right'))↵
      ('Event'('KeyPress' 'EditKeyPress'))  ¶
[6]     'EditBox1'⎕WC'Edit' ''(48 0)(184 232)↵
      ('Attach'('Top' 'Left' 'Bottom' 'Right'))('HScroll' ¯1)↵
      ('VScroll' ¯1)('Style' 'Multi')('Event'↵
      ('KeyPress' 'EditBoxKeyPress'))¶
[7]     'TrackBar1'⎕WC'TrackBar'↵
      ('Attach'('Bottom' 'Left' 'Bottom' 'Right'))('Limits'(1 1))↵
      ('Posn'(232 0))('Size'(32 184))('TickAlign' 'Top')↵
      ('Event'('KeyPress' 'TrackBarKeyPress')('Scroll' 'TrackBarEvent')↵
      ('ThumbDrag' 'TrackBarEvent'))¶
[8]     'Push1'⎕WC'Button' 'Close'(240 184)(24 48)    ↵
      ('Attach'('Bottom' 'Left' 'Bottom' 'Right'))('Data'(1))('Defa↵
ult' 1)('Event'('Select' 1))∇¶
```

Notice that extra spaces introduced at the front of the last line above *do* matter, whereas spaces introduced at the front of the second last line displayed above *do not* matter - through expected APL syntax lenience.

One conceptual leap involved in object oriented programming (OOP) relates to a new mental model of the place of a program. Evolving from a linear sequence of instructions with occasional embedded jumps, OOP conceives a model of a hierarchy of objects each bristling with callback functions that can arbitrarily modify existing objects and create with impunity other new bristling object hierarchies.

2.3.3.2 Ask for the next module on **dot syntax**. How did you get on with Module 2? Was it clear? ☺