Namespaces: APL/W vs. APL2

Rexford H. Swain
Independent Consultant
8 South Street
Washington, CT 06793
USA
Tel: 203-868-0131
Fax: 203-868-9970
Internet: rswain@ix.netcom.com

0. Abstract

This paper describes and contrasts the implementation of namespaces in two popular combinations of APL dialects and platforms: IBM's APL2 (version 2 release 2) running under VM/CMS, and Dyadic's Dyalog APL/W (version 7.0) running under Microsoft Windows.

In a traditional APL workspace, localization is the only mechanism available to isolate identifier names and values, and it is extremely potent. While a function that localizes a given name is executing, it is impossible to reference any other definition of the same name.

Namespaces are a significant addition to an APL system. They provide a way to "package" a collection of objects (variables, defined functions, etc.), and insulate them from the traditional workspace. A namespace co-exists with the workspace, yet its object names and definitions are completely independent of the workspace. Namespaces can help to organize and hide complexity in workspaces, avoid name conflicts, and share code among applications.

IBM has offered namespaces in APL2 since 1987, whereas Dyadic just recently introduced their facility. Both implementations provide encapsulation and name isolation within a workspace. But they use quite different methods to access objects in a namespace and store their initial definitions. The merits of each approach are discussed. An example is provided to illustrate how namespaces may be used to simplify the implementation and improve the functionality of workspace and function analysis tools.

Keywords: Namespace, localization, package.

APL '95, San Antonio, Texas, USA

1. What's a Namespace?

A traditional APL workspace contains a set of global identifier names and definitions. When localized, a given name may have a different value, but it is impossible to reference the local and global values simultaneously — only one set of names and values is available at any given time. Furthermore, when a function finishes execution, the values of its local variables are lost forever.

If you think of a traditional workspace as being "flat" or "simple", a workspace that contains a namespace could be thought of as "nested". One namespace can contain many objects — and the names and values of those objects are "local" to the namespace.

Another analogy is that objects in a traditional workspace are like PC DOS files before the advent of directories, when every file was in what is now known as the "root" directory, and thus every file name on an entire disk had to be unique. Creating a namespace in a workspace is like creating a sub-directory on a disk — it gives you the ability to organize and isolate sets of APL objects.

Like nested arrays and DOS directories, namespaces may be created within other namespaces, to an arbitrary depth. And the traditional active workspace is *itself* a namespace (which I refer to as the "root" namespace), analogous to the outer-most level of a nested array or the root directory of a disk. As with DOS disks, there is a "current" namespace, and all namespaces except the root namespace have a "parent" namespace.

Suppose you have a workspace containing a namespace `UTIL` which contains, among other objects, a function `FOO`. Everything behaves in the usual way until `FOO` is executed. Then the APL system automatically switches the current namespace from the root namespace to the `UTIL` namespace. Here an entirely different set of names and values becomes visible, and the names in the parent (surrounding) namespace are no longer visible. While `FOO`

is executing, any objects it *and its subroutines* refer to will be retrieved from the now-current `UTIL` namespace. When `FOO` returns to its caller, the current namespace is automatically switched back to the root namespace. But unlike the traditional localization mechanism, values in the namespace are not lost — they are just not visible, and they will appear again if and when the namespace is re-entered.

Namespace implementations also provide mechanisms for functions in one namespace to interact with the names and values of objects in any other namespace. For instance, you can reach "out" from a namespace to retrieve an object from the parent namespace.

(Technically, APL2 namespaces do not have names and do not exist "within" a workspace. And they are not really arranged in a parental hierarchy; IBM uses the term "caller's" namespace rather than "parent". More about this in section 5 below.)

In addition to any variables, functions, and operators that you may save in a namespace, every namespace also contains its own system variables. For example, each namespace has its own `⎕IO` whose value is independent of the parent (or any other) namespace.

Happily, namespaces are fast! Basically, the APL interpreter maintains more than one symbol table in a workspace — one for each namespace. When the system encounters an object in different namespace, it's just a matter of switching a pointer to the appropriate symbol table.

2. Why Use Namespaces?

Namespaces address the inconvenient "flatness" of the traditional APL workspace, as well as the scope and duration of object names and definitions. Some potential uses and benefits are:

- Organize workspaces:

  Workspaces themselves serve to organize programs and data. But as workspaces grew larger over the years, it became clear to implementers that something more was needed to organize objects within a workspace. Various features have been added to many APL dialects, starting with the APL\360 `)GROUP` command (now largely abandoned). APL2 allows the use of "indirect" lists with certain commands, such as `)COPY`. But these grouping facilities typically only work with system commands. Namespaces provide a more powerful and dynamic way to impose order on increasingly large and overpopulated workspaces.

- Hide complexity:

  Namespaces allow a workspace to be constructed so that there are relatively few global names. The intricacy of the structure of objects in a namespace is hidden from direct view, and only certain high-level objects need be visible or known to the surrounding application. (You no longer have to see all 87 of your user-interface subroutines every time you do `)FNS`.) This can be particularly helpful in situations where semi-skilled APL'ers work in an application "shell", doing some of their own programming but frequently invoking more sophisticated subroutines provided by an expert support staff.

- Share programs across applications:

  Many applications use common code for user interface, database access, and so on. Namespaces can be used to "package" and segregate collections of such utilities, allowing uniform and convenient access across workspaces. Central maintenance and distribution of utilities can be vastly simplified. Code that has been packaged into a namespace is easy to reuse, providing some of the benefits of "object-oriented" programming.

- Share programs among users:

  In APL2, code in a single namespace can be installed and shared by many users. And multiple namespaces can overlay each other in storage while they are all shared among many users, which can significantly reduce overall system storage requirements.

- Avoid name conflicts:

  Some utilities that are designed to be used in many workspaces must go to obscene lengths to avoid possible name conflicts if they use global variables and/or subroutines. Namespaces alleviate this problem by providing straightforward name isolation.

- Avoid shadowing:

  A function that wants to examine other objects in the active workspace is hampered by the possibility that its local variables may "shadow" some of the global objects that it tries to examine. Namespaces allow a function an unobstructed view of the world outside itself.

- Retain persistent local values:

  Functions and variables in a namespace are "local" to the parent namespace, but their definitions and values are static (lasting) rather than dynamic (fleeting) — they do

not disappear as execution leaves the namespace, and changes may be)SAVEd across sessions.

- Assist graphical user interface applications:

  One of Dyadic's prime motivations for implementing namespaces was to bolster their Windows GUI support. In a typical GUI application, there is a high degree of autonomy between the various GUI objects (form, menu, scrollbar, button, etc.) which comprise the user interface, and it is important that each object have its own domain in which to operate. APL/W GUI objects are now also namespaces, so each can encapsulate all its processing detail internally. For example, when a button is pressed, the system switches to the button's namespace in order to execute the code and possibly update any status information associated with the event. Namespaces help to alleviate "name pollution", which can otherwise become quite severe in this context. Previously, status values which need to be maintained between successive invocations of a callback function might have been kept in numerous global variables, but now each can be isolated in the appropriate GUI object/namespace.

- Implement and distribute run-only applications:

  IBM's namespaces are well-suited to the construction of run-only applications because the stored namespace is basically read-only, and access may be restricted to certain objects, so entry points to the application may be limited. And because APL2 namespaces reside in text decks (or load module libraries), they can be delivered in a way that allows system programmers to install them using the same operating system facilities and tools used for applications written in other languages. The APL2 Application Environment, IBM's run-only interpreter, can be used to process such namespaces.

3. Workspace Analysis Tools

My favorite use of namespaces is to avoid name conflicts and shadowing in workspace and function analysis tools. In traditional APL systems, you cannot bring a tool into a workspace without fear of displacing at least one existing object. And when the tool is executed, its local variables might shadow some of the very global objects that it is trying to examine. (An annoying corollary of Heisenberg's uncertainty principle: the process of measuring something inevitably distorts the results.)

For instance, a traditional *WSDOC* tool cannot possibly show *everything* in a workspace because the name of the tool itself might conflict with an object already in the workspace. (What if you want to use your *WSDOC* to look at my *WSDOC*?) And once the tool starts executing, the probability is even greater that its local variables will obscure some global objects.[1] This has lead many programmers to many strange and inconvenient practices, especially the use of extremely unusual local variable names.[2] And, of course, no matter how bizarre the naming convention, there is no guarantee that a conflict will not arise.

As you will see, namespaces provide solutions to these problems. But first, a look at the different ways that Dyadic and IBM have implemented namespaces.

4. Dyalog APL/W Namespaces

In APL/W, you use □*NS* to create a namespace and put objects into it. For example:

```
      )CLEAR
clear ws
      SCALE←2
      □FX 'R←FOO A' 'R←A×SCALE'
      N←128
      'UTIL' □NS 'FOO' 'SCALE' 'N'
#.UTIL
      (□NL 2 3 9),□NC □NL 2 3 9
FOO   3
N     2
SCALE 2
UTIL  9
```

□*NS* creates a namespace named *UTIL* (left argument) and stores a list of objects (right argument) into it. Note that the objects stored in the namespace are copied from the *active* workspace, and that only *specifically named* user-defined objects are included in the namespace. (*System* functions and variables, such as □*IO*, are automatically included.) The namespace now exists *in the active workspace* as a *single object* with name class 9. You can manipulate *UTIL* with many of the usual facilities, such as)*COPY* and □*EX*.

□*NS* returns the fully-qualified name of the namespace. APL/W namespace hierarchies are very similar to DOS directories, except that you use . as a separator, # to indicate the root namespace, and ## to refer to the parent of the current namespace:

---

[1] Some systems provide facilities for *detecting* these situations (e.g., □*IDLOC* in APL*PLUS), but generally all you can do is announce the problem to the user — you still can't actually do what you wanted.

[2] I would be delighted to *never* see another function header like ...;⍙⍙*A*;⍙⍙*B*;⍙⍙*C*;...! And, ironically, to stay sane while maintaining such tools, you need *another* tool to automate the renaming of locals while converting a development version into a production version!

Relative to...	Namespace	DOS Directory
Root	`#.UTIL`	`\UTIL`
Current	`UTIL`	`UTIL`
Parent	`##.UTIL`	`..\UTIL`

To refer to an object in a namespace, you simply prefix an object name with the name of the desired namespace, again using the period as a separator:

Relative to...	Object	DOS File
Root	`#.UTIL.FOO`	`\UTIL\FN.EXT`
Current	`UTIL.FOO`	`UTIL\FN.EXT`
Parent	`##.UTIL.FOO`	`..\UTIL\FN.EXT`

You can also alter existing objects and create new objects in the namespace by simply assigning them:

```
      ⎕FX 'R←FOO A' 'R←A+SCALE'
      N←32
      SCALE←1
      FOO N
33
      UTIL.FOO N
64
      FOO UTIL.N
129
      UTIL.FOO UTIL.N
256
      UTIL.SCALE←10 100
      UTIL.N←9
      UTIL.FOO N
320 3200
      UTIL.FOO UTIL.N
90 900
```

System functions and variables may be used in the same way:

```
      UTIL.⎕FX 'R←IOTA A' 'R←⍳A'
      UTIL.⎕IO←0
      ⎕IO
1
      UTIL.IOTA 5
0 1 2 3 4
      M←UTIL.⎕NL 2 3 9
      (M,UTIL.⎕NC M),⎕NC M
FOO    3 3
IOTA   3 0
N      2 2
SCALE  2 2
```

Note that the namespace object must exist in the active workspace before it can be used:

```
      )SAVE TEMP
TEMP saved Sun Oct 23 12:17:54 1994
      )CLEAR
clear ws
      UTIL.FOO 99
VALUE ERROR
      UTIL.FOO 99
           ^
      )COPY TEMP UTIL
TEMP saved Sun Oct 23 12:17:54 1994
      (⎕NL 2 3 9),⎕NC ⎕NL 2 3 9
UTIL 9
      UTIL.FOO 5
50 500
```

You may change the current namespace with the system command `)CS`, like the DOS CD/CHDIR command. This gives you the convenience of working in immediate execution mode within a namespace:

```
      )LOAD TEMP
TEMP saved Sun Oct 23 12:17:54 1994
      )FNS
FOO
      )CS UTIL
#.UTIL
      )FNS
FOO      IOTA
```

There is also a system command `)NS` that creates a new namespace (without putting anything into it), like the DOS MD/MKDIR command.[3] The fully-qualified name of the namespace is reported:

```
      )NS DEEPER
#.UTIL.DEEPER
```

`⎕NS` can do this too — with a namespace specified in its left argument and an empty right argument, it creates the namespace without copying anything into it. Instead of the `)NS` command above, we could have executed:

```
      'DEEPER' ⎕NS ''
#.UTIL.DEEPER
```

4.1. Create/Change Objects In a Namespace

As you have seen, `⎕NS` may be used to create a new namespace and copy objects from the current namespace into it. You may also directly create, re-define, and erase objects in a namespace:

```
      )CS #
#
      UTIL.N←0              ⍝ Change var
      UTIL.B←52             ⍝ New var
      UTIL.⎕FX 'R←AVG A' 'R←(+/A)÷⍴A'
```

[3] The `)CS` and `)NS` commands are inconsistent with their analogous DOS commands when used with no argument. `)CS` *changes* the current namespace to the root namespace, whereas CD *queries* the current directory; and `)NS` queries the current namespace, whereas MD fails.

```
      UTIL.⎕EX 'SCALE'   ⍝ Erase var
      UTIL.⎕NL¯2 3 9
B  AVG   DEEPER
N  FOO
   IOTA
      UTIL.⎕EX 'DEEPER' ⍝ Erase ns
```

This changes *N*, creates *B* and *AVG*, and erases *SCALE* and *DEEPER* in the *UTIL* namespace — without affecting the current namespace.

4.2. Query Current Namespace

When both ⎕*NS* arguments are empty, it queries the current namespace:

```
      '' ⎕NS ''
#
      )CS UTIL
#.UTIL
      '' ⎕NS ''
#.UTIL
```

4.3. WSDOC Using APL/W

Let's take a look at how you might use APL/W to write and then invoke a simplified workspace listing tool. The easiest approach is to write the code so that it refers to its parent namespace, and then store the main function and all required subroutines into a namespace. Before defining the functions, we use)*NS* to create a new namespace, and then)*CS* make it the current namespace:

```
      )CLEAR
clear ws
      )NS DOC
#.DOC
      )CS DOC
#.DOC

    ∇ WSDOC;M;V
[1]   ⎕←'Workspace: ',⎕WSID
[2]   ⎕←'Namespace: ','' ##.⎕NS ''
[3]   M←##.⎕NL 3 4    ⍝ Names matrix
[4]   V←↓M            ⍝ Nested vector
[5]   WSDISP¨V        ⍝ Display each
    ∇

    ∇ WSDISP N
[1]   ⎕←' '           ⍝ Blank line
[2]   ⎕←'∇ ',N        ⍝ Show name
[3]   ⎕←##.⎕CR N      ⍝ Definition
    ∇

      )SAVE TOOLS
TOOLS saved Mon Oct 24 12:06:31 1994
```

To use *WSDOC*, you must first copy the *DOC* namespace into the target workspace, potentially displacing one existing object (e.g., a variable named *DOC* would be clobbered). This is nothing new to most APL'ers, and at least we still did not have to bend over backwards within *WSDOC* to avoid shadowing.

```
      )LOAD MYAPP
MYAPP saved Mon Oct 24 11:59:14 1994
      )COPY TOOLS DOC
TOOLS saved Mon Oct 24 12:06:31 1994
      DOC.WSDOC
Workspace: MYAPP
Namespace: #
...function listings...
```

We invoke *WSDOC* directly from its namespace.[4] Once *WSDOC* starts executing (in the *DOC* namespace), it tells ⎕*NL* to look *out* to its *parent* namespace for a list of all defined functions and operators. (Actually, it executes ⎕*NL* *in* the parent namespace.) In this example, the parent of *DOC* is the root, but since both ⎕*NL* and ⎕*CR* refer to the parent (rather than the root) namespace, *WSDOC* will document the *current* namespace (the one that *DOC* was copied into). This is useful because you might want to run *WSDOC* in a workspace where)*CS* has been used to change the current namespace to something other than the root namespace.

4.4. Using the Session Namespace

APL/W has a special system namespace named ⎕*SE* that is used by the session manager and is therefore always[5] available. Because it survives)*LOAD* and)*CLEAR*, it is a very handy place to store development tools.

You may store objects into the session namespace in the usual way, but to save the namespace permanently (so that it will survive)*OFF*), you must choose Save from the Session menu.

We can copy *WSDOC* and its subroutine into the session namespace without re-writing the functions:

```
      )LOAD TOOLS
TOOLS saved Mon Oct 24 12:06:31 1994
      )CS DOC
#.DOC
      '⎕SE' ⎕NS 'WSDOC' 'WSDISP'
⎕SE
```

---

[4] Alternatively, you could first)*CD DOC* and then just type *WSDOC*.

[5] Actually, the ⎕*SE* namespace is *almost* always available. One instance when it is *not* available is when executing the run-time version of APL/W, so it is *not* a good place to keep utilities required by an application that may eventually be distributed as a stand-alone product.

Then we can invoke *WSDOC* at any time:

```
      )LOAD MYAPP
MYAPP saved Mon Oct 24 11:59:14 1994
      ⎕SE.WSDOC
Workspace: MYAPP
Namespace: #
...function listings...
```

Since we have exploited the always-available nature of the ⎕*SE* namespace, we can run *WSDOC* from *any* workspace without having to)*COPY* anything first. And because we invoke it directly from ⎕*SE*, not even a single name in the target workspace is added, displaced, or shadowed!

But here's the bad news: ⎕*SE.WSDOC* executes ⎕*NL* and ⎕*CR* in its parent namespace, and since the parent of the ⎕*SE* namespace is the root namespace, it will *always* examine the *root* namespace. So, as it stands, we cannot use *WSDOC* from ⎕*SE* to document a current *non-root* namespace.

In order to make our tool document *any* current namespace, we must exploit an APL/W GUI query:

```
      '⎕SE' ⎕WG 'CurSpace'
```

⎕*WG* gets Windows object properties. Here we ask for the CurSpace[6] property of the ⎕*SE*[7] object. When executed in the ⎕*SE* namespace, this returns the namespace that it was invoked *from* (as opposed to its parent), which is exactly what we need:

```
      )CS ⎕SE
⎕SE

    ∇ WSDOC2;M;NS;V
[1]   ⎕←'Workspace: ',⎕WSID
[2]   NS←'⎕SE' ⎕WG 'CurSpace'
[3]   ⎕←'Namespace: ',NS
[4]   M←⍕NS,'.⎕NL 3 4' ⍝ Names matrix
[5]   V←↓M             ⍝ Nested vector
[6]   WSDISP2¨V        ⍝ Display each
    ∇

    ∇ WSDISP2 N
[1]   ⎕←' '            ⍝ Blank line
[2]   ⎕←'∇ ',N         ⍝ Show name
[3]   ⎕←⍕NS,'.⎕CR N'   ⍝ Definition
    ∇
```

This revised tool allows us to document a non-root current namespace:

```
      )LOAD TOOLS
TOOLS saved Mon Oct 24 12:06:31 1994
      )CS DOC
#.DOC
      ⎕SE.WSDOC2
Workspace: TOOLS
Namespace: #.DOC
...function listings...
```

Of course, you do need to beware of name conflicts *within* the ⎕*SE* namespace. (What if two developers give you a *WSDOC* tool and both suggest that they be placed in ⎕*SE*?) It would be safer to store such tools in separate namespaces inside the session namespace:

```
      )CS ⎕SE
⎕SE
      'REX' ⎕NS 'WSDOC2' 'WSDISP2'
⎕SE.REX
      )ERASE WSDOC2 WSDISP2
      )LOAD TOOLS
TOOLS saved Mon Oct 24 12:06:31 1994
      )CS DOC
#.DOC
      ⎕SE.REX.WSDOC2
Workspace: TOOLS
Namespace: #.DOC
...function listings...
```

4.5. Using Assigned Functions

If you are willing to displace a name in your current namespace, you may take advantage of Dyadic's assignable function feature and create an "alias" whose name may be more convenient to type.

Suppose you have a function named *XREF* stored in a namespace named *UTIL* within the ⎕*SE* namespace. In any workspace/namespace, you could then execute:

```
      ⎕NC¨'XREF' 'FOO'
0 3
      XREF←⎕SE.UTIL.XREF
      ⎕NC¨'XREF' 'FOO'
3 3
      XREF 'FOO'
...cross reference listing...
```

Unfortunately, this technique will not work for *WSDOC* because it is niladic — only monadic and dyadic functions may be assigned in this way.

The definition of an assigned function is not dynamic — it is fixed at the time of assignment (a copy of the target function is stored in the current namespace of your active

---

[6] The CurSpace property is not documented, but I noticed it being used in the ⎕*SE.WSDoc* namespace that is supplied with APL/W.

[7] ⎕*SE* is treated as a special system GUI object; it is the only object with type Session.

workspace). So if you change *XREF* in the □*SE.UTIL* namespace, the *assigned* version of *XREF* will *not* change. Depending on your perspective, this behavior might be considered a problem or an advantage.

5. IBM APL2 Namespaces

In APL2, you use an external function (supplied with the product) named *PACKAGE* to create a namespace from a saved workspace:

```
      )CLEAR
CLEAR WS
      SCALE←2
      □FX 'R←FOO A' 'R←A×SCALE'
FOO
      N←128
      (□NL 2 3),□NC □NL 2 3
FOO   3
N     2
SCALE 2
      )SAVE TEMP
1994-10-23 10.43.38 (GMT)
      )CLEAR
CLEAR WS
      3 11 □NA 'PACKAGE'
1
      )FNS
PACKAGE
      PACKAGE 'TEMP'
TEMP TEXT A
```

□*NA* is used to access (name associate) the supplied routine *PACKAGE* via processor 11[8], and returns a 1, which indicates that the association was successful. *PACKAGE* is then used to create a namespace from the workspace *TEMP*. Note that the namespace is created from the *saved* (not active) workspace, and that the *entire* saved workspace (not just a set of specified objects) is packaged into the namespace. The namespace now exists in a CMS[9] file whose name is returned by *PACKAGE* and is based on the saved workspace name. (If this file already exists, it will be overwritten without warning.)

But our new namespace is not yet available for use in any workspace — to access objects in a namespace, you must first make associations between a workspace and the namespace.

In APL2, the active workspace is like a namespace, but not quite — there is not necessarily a saved and/or packaged version of the active workspace. IBM makes a nice distinction here, introducing the term "namescope". Each namespace and the active workspace contains its own namescope. Each namescope contains a distinct set of objects that are known and usable in that namescope.

To use an object in your namespace you must first use □*NA* again to establish an association between a name in your active workspace (current namescope) and a name in the namespace. This time the left argument specifies the name of the namespace and processor 11, and the right argument specifies the name of the object. The namespace can be accessed from *any* workspace/namescope, *without* using system commands:

```
      )CLEAR
CLEAR WS
      'TEMP' 11 □NA 'FOO'
1
      )FNS
FOO
      N←32
      FOO N
64
```

Your active workspace now contains two namescopes: one based on the *TEMP* namespace, and the "root" (which is the current namescope). While *FOO* is executing, the *TEMP* namescope[10] is current and its caller is the root. The root namescope can always be made the current namescope by executing)*RESET*.

In IBM terminology, the "current" namescope is the namescope in which execution is currently taking place, and the "caller's" namescope is the namescope from which the current namescope was entered. APL2 namespaces are not arranged in a parental hierarchy; it is better to think of them as parallel.

5.1. Aliases

If you want to access an object from a namespace, but that object's name conflicts with an object name in the current namescope, you must declare an "alias". You do this by specifying *two* names in the right argument of □*NA*: first the alias that you want to use in the current namescope, and then the original name in the target namespace:

---

[8] Processor 11 provides access to objects outside the active workspace: either objects in APL2 namespaces or routines written in other languages. The 3 is a name class, telling processor 11 to look for a *function*.

[9] This all works similarly under MVS/TSO too, but I'm sticking with VM/CMS for these examples.

[10] Actually, in APL2, neither namespaces nor namescopes have names — a file containing a namespace definition has a name, but the namespace itself is unnamed. But personally, because the file has a name, I find it easier to think of the namespace as having the same name.

```
      )VARS
N
      N
32
      'TEMP' 11 ⎕NA 'PN N'
1
      )VARS
N       PN
      PN
128
      FOO N PN
64 256
```

You must also use an alias to invoke a system function or variable in the namespace:

```
      'TEMP' 11 ⎕NA⊃'NL ⎕NL' 'NC ⎕NC'
1 1
      (⎕NL 2 3),⎕NC ⎕NL 2 3
FOO 3
N   2
NC  3
NL  3
PN  2
      (NL 2 3),NC NL 2 3
FOO   3
N     2
SCALE 2
```

5.2. Save Changes In a Namespace

In APL2, namespaces do not exist within the workspace. Intermediate results, modified objects, and the name table consume workspace storage, but all other objects are totally outside the active workspace. Any changes that you make to objects in the namespace are "local" to the namespace, and will be saved along with the active workspace by a)*SAVE* command. However, the stored copy of the *external* namespace is *not* changed:

```
      'TEMP' 11 ⎕NA 'SCALE'
1
      SCALE←10 100
      )SAVE TEMPSAVE
1994-10-23 13.47.02 (GMT)
      )CLEAR
CLEAR WS
      'TEMP' 11 ⎕NA 'FOO'
1
      FOO 8
16
      )LOAD TEMPSAVE
Saved 1994-10-23 13.47.02 (GMT)
      FOO 8
80 800
```

If the external namespace *is* changed (that is, if its source workspace is re-)*SAVE*d and re-*PACKAGE*d), then the next time the application workspace is loaded, any objects associated with the namespace will be *re-initialized* from the external namespace as soon as they are used, and the changes that *were* in the active workspace's copy of the namespace will be lost:

```
      )LOAD TEMPSAVE
Saved 1994-10-23 13.47.02 (GMT)
      FOO 8
80 800
      )LOAD TEMP
Saved 1994-10-23 10.43.38 (GMT)
      SCALE←4
      )SAVE
1994-10-23 13.52.51 (GMT)
      3 11 ⎕NA 'PACKAGE'
1
      PACKAGE 'TEMP'
TEMP TEXT A
      )LOAD TEMPSAVE
Saved 1994-10-23 13.47.02 (GMT)
      FOO 8
32
```

This characteristic can be somewhat unsettling when first encountered, but it is very helpful in maintaining utility functions that are used in many applications. For instance, if you enhance your user-interface utilities and then re-package their namespace, *all* saved workspaces that use the utilities via their namespace would then automatically start accessing the new versions.

5.3. Create/Change Objects In a Namespace

You cannot directly manipulate objects in an APL2 namespace — you must reach into the namespace by associating with a system function (or previously-defined and packaged user-defined function) and then use that function to do the work.

Defining a new function and erasing an existing object in a namespace are straightforward: use an alias for ⎕*FX* and ⎕*EX* in the namespace:

```
      'TEMP' 11 ⎕NA⊃'FX ⎕FX' 'EX ⎕EX'
1 1
      FX 'R←AVG A' 'R←(+/A)÷⍴A'
AVG
      EX 'SCALE'
1
```

And as we have already seen, you may change the value of an existing variable in a namespace by associating with it and then simply re-assigning it:

```
      'TEMP' 11 ⎕NA 'N'
1
      N←0
```

Creating a *new* variable in a namespace is trickier because there is no name there to associate with. APL2 does not allow you to associate an alias with the assignment arrow or any *primitive* function (such as ⍎)[11]; you can only associate with *named* objects. Again, a system function is called for; ⎕EC[12] is typically used for ad hoc work like this:

```
      'TEMP' 11 ⎕NA 'EC ⎕EC'
1
      SINK←EC 'B←52'
```

5.4. Query Current Namespace

The external function *QNS* (supplied by IBM) may be used to query the current namespace.[13] It takes a mandatory right argument of zero.[14] The name of the root namespace is returned as an empty vector:

```
      3 11 ⎕NA 'QNS'
1
      Q←QNS 0
      2 ⎕TF[15] 'Q'
Q←'' 11
```

What does *QNS* return when executed from a namespace? This would have been easier to demonstrate if we had defined it *before* packaging the workspace *TEMP*. Instead we must now reach into the namespace with two system functions aliases, first defining *QNS* (with ⎕*NA*) and then executing it (with ⎕*EC*), so that both the association and execution take place in the namespace:

```
      'TEMP' 11 ⎕NA 'NA ⎕NA'
1
      3 11 NA 'QNS'
1
      Q←3⊃EC 'QNS 0'
      2 ⎕TF 'Q'
Q←'TEMP' 11
```

Note that *QNS* returns precisely the left argument that we need to ⎕*NA* our way into the namespace.

5.5. WSDOC Using APL2

Putting this all together, here is how you might build a *WSDOC* tool with APL2:

```
      )CLEAR
CLEAR WS

     ∇ WSDOC;A;CR;M;NL;SINK;V
[1]    A←'NL ⎕NL' 'CR ⎕CR'
[2]    SINK[16]←'' 11 ⎕NA⊃A
[3]    M←NL 3 4          ⍝ Names matrix
[4]    V←⊂[1+⎕IO]M       ⍝ Nested vector
[5]    WSDISP¨V          ⍝ Display each
     ∇

     ∇ WSDISP N
[1]    ⎕←' '             ⍝ Blank line
[2]    ⎕←'∇ ',N          ⍝ Show name
[3]    ⎕←CR N            ⍝ Definition
     ∇

      )SAVE TOOLS
1994-10-30  9:55:09
      3 11 ⎕NA 'PACKAGE'
1
      PACKAGE 'TOOLS'
TOOLS TEXT A
      )LOAD MYAPP
1994-10-24 11:59:14 (GMT)
      'TOOLS' 11 ⎕NA 'WSDOC'
1
      WSDOC
...function listings...
```

APL2 does not have a ⎕*WSID*, so it is not a simple matter to report the current workspace name.[17]

The process of defining the tool in the active workspace potentially displaces one name (*WSDOC*). Of course, if you *anticipate* a name conflict, you could "manually" associate an alias with *WSDOC*:

```
      )LOAD MYAPP
1994-10-24 11:59:14 (GMT)
      'TOOLS' 11 ⎕NA 'MYALIAS WSDOC'
1
      MYALIAS
...function listings...
```

---

[11] Similarly, APL/W doesn't permit anything like *NS*.⍎*X*.

[12] *R*←⎕*EC* *X* performs "execute controlled" of expression *X*; the third item of *R* is the result of ⍎*X* (or an error message).

[13] More precisely, *QNS* queries the left argument of ⎕*NA* of the function that was used to enter the current namescope.

[14] Presumably, the purpose of this unused argument is simply to avoid having *QNS* be niladic. Future extensions to a monadic *QNS* are possible without changing its syntax.

[15] *R*←2 ⎕*TF* '*X*' returns the "transfer form" of object *X*; ⍎*R* may be used to recreate the object.

[16] Brief tirade: *Why* must I code *SINK*← (or ⎕*WA*←) when I just want to throw away a result? It seems obvious that *monadic* assignment should be allowed for this purpose!

[17] IBM does supply a defined function named *WSNAME* in the *TRANSFER* workspace in public library 2, which does the same thing. But after taking a look at it, you may still wish they would implement ⎕*WSID*.

5.6. Examining the Root Namespace

An annoying problem with using `'' 11 ⎕NA` is that it fails unless it is executed from a namespace. This creates an inconvenient development cycle — as it stands, *WSDOC* must be packaged before it can be run, and re-packaged between each change and test. It would be more convenient if we could test without packaging.

You can work around this problem by fixing a cover function instead of associating an alias when you detect that you are running in the root namespace. One way to do this is to use *QNS* before attempting an association:

```
[…]   SINK←3 11 ⎕NA 'QNS'
[…]   →('' ≡1⊃QNS 0)⍴ROOT
[…]   SINK←'' 11 ⎕NA 'NL ⎕NL'
[…]   →OK
[…]  ROOT:
[…]   SINK←⎕FX 'R←NL A' 'R←⎕NL A'
[…]  OK:
[…]   M←NL 3 4
```

Or, you can attempt an association and examine ⎕*NA*'s result — if it fails, you're not running in a namespace:

```
[…]   →('' 11 ⎕NA 'NL ⎕NL')⍴OK
[…]   SINK←⎕FX 'R←NL A' 'R←⎕NL A'
[…]  OK:
[…]   M←NL 3 4
```

With either of these techniques you can proceed to use the same "alias" for the system function whether testing (unpackaged) or in production (packaged). (But it sure would be a lot simpler if you could *always* use `'' 11 ⎕NA`...)

Of course, if you execute such code in unpackaged form, local variables may shadow objects in the target namespace, but during development this is typically an acceptable tradeoff for the faster testing cycle.

5.7. Current Non-Root Namespace

Note that because we use `'' 11 ⎕NA` to associate with the system functions, *WSDOC* examines the *root* namespace in the active workspace (which isn't necessarily the caller's namespace).

In APL2, the only way to get into immediate execution mode with a non-root namespace as the current namespace is for a function executing in the namespace to suspend.

Since we professionals never make programming errors that cause unintentional suspensions, it is a good debugging practice to include a hook that will allow a deliberate suspension when desired. This can be as simple as reaching into the namespace and setting a stop control:

```
      'TEMP' 11 ⎕NA⊃'EC ⎕EC' 'FOO'
1 1
      SINK←EC 'S∆FOO←1'
      FOO 5
FOO[1]
```

But what if we want to be able to document a non-root current namespace? Unfortunately, *QNS* cannot be used to query its *caller's* namespace, so to make *WSDOC* work in such a context, we must use *QNS* to query the *current* namespace *before* entering the tool namespace:

```
      )CLEAR
CLEAR WS

    ∇ WSDOC2;NS;QNS;SINK;WSSUB2
[1]   SINK←3 11 ⎕NA 'QNS'
[2]   NS←QNS 0          ⍝ Where are we?
[3]   SINK←'TOOLS2' 11 ⎕NA 'WSSUB2'
[4]   WSSUB2 NS
    ∇

    ∇ WSSUB2 NS;A;CR;M;NL;SINK;V
[1]   ⎕←'Namespace: ''',(⍕NS),''''
[2]   A←'NL ⎕NL' 'CR ⎕CR'
[3]   SINK←NS ⎕NA⊃A
[4]   M←NL 3 4          ⍝ Names matrix
[5]   V←⊂[1+⎕IO]M       ⍝ Nested vector
[6]   WSDISP2¨V         ⍝ Display each
    ∇

    ∇ WSDISP2 N
[1]   ⎕←' '             ⍝ Blank line
[2]   ⎕←'∇ ',N          ⍝ Show name
[3]   ⎕←CR N            ⍝ Definition
    ∇

      )OUT TOOLS2 WSDOC2
      )ERASE WSDOC2
      )SAVE TOOLS2
1994-10-30  9:55:09
      3 11 ⎕NA 'PACKAGE'
1
      PACKAGE 'TOOLS2'
TOOLS2 TEXT A
      )LOAD MYAPP
1994-10-24 11:59:14 (GMT)
      'TEMP' 11 ⎕NA⊃'EC ⎕EC' 'FOO'
1 1
      SINK←EC 'S∆FOO←1'
      FOO 5
FOO[1]
      )IN TOOLS2 WSDOC2
      WSDOC2
Namespace: 'TEMP'
...function listings...
```

Note that *WSDOC2* is *not* executed from the namespace — it is used in its native form in the target workspace. The sole reason for this is so that the current namespace name (*NS*) can be queried *before* being passed into the tool namespace as the argument to the packaged subroutine *WSSUB2*, which uses it to do its associations. Note also that since *WSDOC2* is defined in the target namespace, *WSSUB2* will "see" it and include it in the function listings.

5.8. Caller's Namespace

Another way to examine a non-root current namespace is to make use of a supplied routine named *EXP*[18] which evaluates expressions in its *caller's* namespace and works even when it is *not* packaged:

```
      )CLEAR
CLEAR WS

    ∇ WSDOC3;EXP;M;SINK;V
[1]   SINK←3 11 ⎕NA 'EXP'
[2]   M←EXP '⎕NL'(3 4) ⍝ Names matrix
[3]   V←⊂[1+⎕IO]M      ⍝ Nested vector
[4]   WSDISP3¨V        ⍝ Display each
    ∇

    ∇ WSDISP3 N
[1]   ⎕←' '            ⍝ Blank line
[2]   ⎕←'∇ ',N         ⍝ Show name
[3]   ⎕←EXP '⎕CR' N    ⍝ Definition
    ∇

      )SAVE TOOLS3
1994-10-30  9:55:09
      3 11 ⎕NA 'PACKAGE'
1
      'WSDOC3' PACKAGE 'TOOLS3'
TOOLS3 TEXT A
      )CLEAR
CLEAR WS
      )LOAD MYAPP
1994-10-24 11:59:14 (GMT)
      'TOOLS3' 11 ⎕NA 'WSDOC3'
1
      WSDOC3
...function listings...
```

Note above that *PACKAGE* accepts an optional left argument, which is a list of names in the namespace that you want to be accessible from outside the namespace. (Without a left argument, access to *any* name in the namespace is permitted.) In this case, the *only* name[19] that can be ⎕*NA*'d in the namespace is *WSDOC3*; the subroutine *WSDISP3*, while still available to *WSDOC3*, is "private" and will never be visible from outside the *TOOLS3* namespace.

5.9. WSDOC Without)LOAD

One rather amazing thing you can do with APL2 is to document a saved workspace without even)*LOAD*ing it:

```
    ∇ WSDOC4 WS;CR;NL;PACKAGE;SINK
[1]   SINK←3 11 ⎕NA 'PACKAGE'
[2]   SINK←PACKAGE WS
[3]   SINK←WS 11 ⎕NA 'NL ⎕NL'
[4]   SINK←WS 11 ⎕NA 'CR ⎕CR'
[5]   M←NL 3 4         ⍝ Names matrix
[6]   V←⊂[1+⎕IO]M      ⍝ Nested vector
[7]   WSDISP4¨V        ⍝ Display each
    ∇

    ∇ WSDISP4 N
[1]   ⎕←' '            ⍝ Blank line
[2]   ⎕←'∇ ',N         ⍝ Show name
[3]   ⎕←CR N           ⍝ Definition
    ∇

      WSDOC4 'MYAPP'
...function listings...
```

There is no need to package *WSDOC4*. And not a single name is displaced in the target workspace!

5.10. Link-Editing

For performance reasons, before making extensive use of a namespace, it should be link-edited into a load library.[20] Several namespaces can be combined into one LOADLIB, each as a MEMBER. Lines 32-40 in Appendix A show how this is done.[21]

---

[18] *EXP*'s argument is a list of items. The possibilities are:

```
R←EXP⊂'NAME'     ⍝ Var or niladic fn
R←EXP 'NAME' RARG      ⍝ Monadic fn
R←EXP LARG 'NAME' RARG ⍝ Dyadic fn
R←EXP 'NAME' '←' VALUE ⍝ Assign var
```

[19] This *includes* system functions, so during testing you may want to add ⎕*EC* to the list to allow for unanticipated poking around in the namespace.

[20] Also, a namespace *must* be link-edited before it can be placed in a shared segment (CMS) or the Link Pack Area (TSO) so that the storage containing the namespace can be shared by multiple users.

[21] IBM also supplies two REXX execs that do link-editing; see AP2MP11L (for a LOADLIB) or AP2MP11M (for a MODULE) in the APL2 reference documentation.

6. Conclusion

The overall capabilities provided by the IBM and Dyadic implementations of namespaces are similar, and both are extremely powerful and very welcome enhancements to their respective systems.

The most significant differences between the two implementations arise from the very different ways that they store namespaces. Each design has its merits.

Because APL2 stores the initial contents of a namespace in a file that is external to any workspace, it is better suited for storing, maintaining, and distributing code that will be used by a number of applications. A namespace is available from any workspace, storage is not wasted by saving the contents of a namespace in every workspace, and changes to the code in a namespace can be distributed with ease. Since the stored namespace is read-only, and access may be restricted to certain objects, APL2 namespaces can be very useful in the construction of run-only applications. On the other hand, the process of creating the namespace is inconvenient (introducing something like the dread code/compile/run cycle into APL development), and requires that the programmer use possibly unfamiliar operating system facilities such as the linkage editor.

APL/W namespaces are significantly easier to create and use. And you do not have to know anything about the surrounding operating system environment to use them. (For those who *are* familiar with the well-known DOS directory scheme, learning is simplified because they follow this paradigm so closely.) They can be created on-the-fly, and it is not necessary to "declare" (associate) anything prior to using them. The maintenance and testing of code in a namespace is straightforward; the `)CS` command is a great convenience. Because it is available across workspaces, the `⎕SE` system namespace is very useful, particularly for developers. In general, they are more accessible to the average APL programmer. But because they are stored *in* a workspace, they are less well-suited for the maintenance and distribution of common code.

One somewhat troubling aspect of Dyadic's implementation is that it introduces new syntax into the APL language. The "dot" naming scheme will be familiar to users of many other systems and languages (UNIX, SQL, etc.). But when you see an expression like `A PLUS.TIMES B`, is there an inner product going on, or a reference to an object in a namespace? APL has gotten into trouble like this before: is / a function or an operator...?

IBM's approach (via `⎕NA`) is a much more generalized design, allowing access to all sorts of objects outside the workspace (including code written in other languages), and it does not introduce any new syntax to the language itself. But it extracts a toll in terms of ease of use.

Dyadic's approach is more APL-centric, and therefore may be more appealing to and more easily learned by the APL purist.

7. Acknowledgments

Writing a paper like this is like teaching almost any topic: you wind up learning a surprising amount more about a topic that you thought you knew pretty well to begin with! I am especially grateful to David Liebtag (IBM) and John Scholes (Dyadic) for their illuminating suggestions and clarifications. Thanks also to Bob Smith (Qualitas) and Bob Hendricks (Sail Systems) for their thoughtful readings of an early draft.

8. Bibliography

8.1. Reference Manuals

Dyalog APL for Windows [version 7.0], *Language Reference*, Dyadic Systems Limited, 1994.

APL2 Programming: System Services Reference, Version 2 Release 2, IBM publication number SH21-1054-01, 1994.

8.2. Conference Proceedings

John Scholes [Dyadic Systems], *Namespaces in Dyalog APL Version 7*, New York SIGAPL *APL as a Tool of Thought IX* conference proceedings, October 1994.

8.3. Other Publications

See also two old publications by members of the IBM APL2 Development group:

- Brent Hawks, *Alice in Packageland.* [1987.]
- Michael T. Wheatley, *Packaged Workspaces.* [1988.]

These are available by contacting the IBM APL2 Hotline: Internet apl2@vnet.ibm.com, or telephone 408-463-2752. Both are slightly out of date, but are nevertheless very interesting supplements to the current documentation.

9. Appendix A:

I find it very helpful to maintain APL2 tools in a "source code" workspace that is then "compiled" into a namespace. In this source workspace I keep the functions I want to package, other development and testing tools, and an *INSTALL* function which creates not only the namespace but a workspace and transfer file containing the specified "visible" tools. The constants on lines 7-10 should be changed to specify the name of the desired namespace and what you want packaged into it. After running *INSTALL*, you may fetch a tool with any of three methods: *)IN*, *)COPY*, or *⎕NA*; for example: *)IN FNTOOLS XREF*, *)COPY FNTOOLS XREF*, or *'TOOLS.FNTOOLS' 11 ⎕NA 'XREF'*.

```
    ∇ INSTALL;A;K;LL;Q;V;WS
[1]  ⍝ Create APL2 namespace from source (active) workspace
[2]  ⍝ Creates production loadlib, workspace, and transfer file
[3]  ⍝ Active workspace is lost in this process, so )SAVE first!
[4]  ⍝ Stack entire process so no shadowing of our locals
[5]  ⍝ 10/26/94  Rex Swain, Independent Consultant, 203-868-0131
[6]
[7]   LL←'TOOLS'                        ⍝ Target LOADLIB
[8]   WS←'FNTOOLS'                      ⍝ Target WS, APLTF, and member
[9]   V←'FNREPL RELABEL XREF'           ⍝ Visible objects
[10]  A←V,' HELP CR2VR XREF'            ⍝ All objects
[11]
[12]  K←'(FIFO)'                        ⍝ Prepare for FIFO stacking
[13]  Q←101 ⎕SVO 'K'                    ⍝ Share K with stack processor
[14]  K←')CHECK SYSTEM DEBUG(2)'        ⍝ Display stacked input
[15]
[16]  K←')OUT ',WS,' ',A                ⍝ Create temp TF (all objects)
[17]
[18]  ⍝ Create temporary workspace from temporary transfer file
[19]
[20]  K←')CLEAR'
[21]  K←')IN ',WS
[22]  K←')WSID ',WS
[23]  K←')SAVE'                         ⍝ Create temp WS from temp TF
[24]
[25]  ⍝ Create package (file FOO TEXT A) from temporary workspace
[26]
[27]  K←')CLEAR'
[28]  K←'3 11 ⎕NA ''PACKAGE'''
[29]  Q←1+⍷(⊂'''''),¨((V≠' ')⊂V),¨''''
[30]  K←Q,' PACKAGE ''',WS,''''
[31]
[32]  ⍝ Linkedit package into LOADLIB
[33]
[34]  K←')HOST FILEDEF SYSLMOD DISK ',LL,' LOADLIB A (RECFM U'
[35]  K←')HOST LKED ',WS,' (NAME ',WS,' NOTERM'
[36]  K←')HOST FILEDEF SYSLMOD CLEAR'
[37]  K←')HOST ERASE ',WS,' LKEDIT A'
[38]  K←')HOST ERASE ',WS,' TEXT A'
[39]  K←')HOST LOADLIB COMPRESS ',LL,' LOADLIB A (DISK'
[40]  K←')HOST ERASE LOADLIB LISTING A'
[41]
[42]  ⍝ Create production TF and WS with ⎕NA'd visible objects
[43]
[44]  K←')CLEAR'
[45]  K←'''',LL,'.',WS,''' 11 ⎕NA⊃',Q
[46]  K←')WSID ',WS
[47]  K←')SAVE'
[48]  K←')OUT ',WS,' ',V
[49]
[50]  K←')CHECK SYSTEM DEBUG(-2)'       ⍝ Shut off stack display
    ∇
```