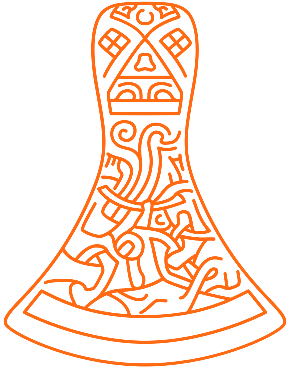
DYALOG

Glasgow 2024

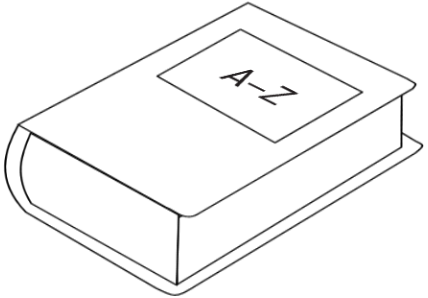
Namespaces in Dyalog



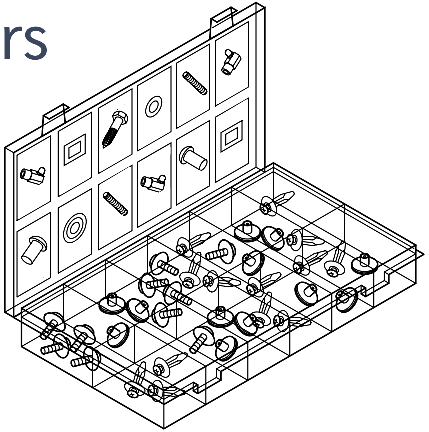

Adám Brudzewsky


John Daintree

Overview

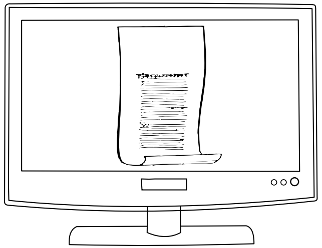


1. Self-contained dictionary of name-value pairs



2. Structured container to organise your code

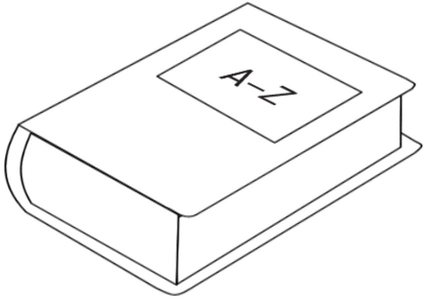
3. Display form and scripted namespaces



The namespace as a

self-contained dictionary

of name-value pairs



Dictionary

```
      ns←⎕NS⍬                       create a namespace
      ≡ns                           it's a kind of simple scalar
0
      ]display ns (2 2⍴ns)          used like any other scalar
┌→─────────────────────────────────────────────────────────────────────┐
│                   ┌→─────────────────────────────────────────────┐   │
│ #.[Namespace]  ↓ #.[Namespace]   #.[Namespace] │   │
│                   │ #.[Namespace]   #.[Namespace] │   │
│                   └#─────────────────────────────────────────────┘   │
└∊─────────────────────────────────────────────────────────────────────┘
```


DYALOG

Dictionary

Value accessed with dot:

```
        dictionary.name ← value     set value
value ← dictionary.name             get value

      ns←⎕NS⍬            create a namespace
      ns.foo←1 2 3       populate it
      10×ns.foo          use the value
10 20 30
```

Dictionary: Dotted expressions

Left side of the dot must be an array of namespace(s).

```
      (ns1 ns2)←(⎕NS⍬)(⎕NS⍬)
      ns1.num←77
      ns2.num←99
      ns1.num ns2.num
77 99
```


DYALOG

Dictionary: Dotted expressions

Left side of the dot must be an array of namespace(s):

```
      (ns1 ns2).num
77 99
      (2 3⍴ns1 ns2).num
77 99 77
99 77 99
```

Dictionary: Dotted expressions

Scalar extension:

```
      (a b)←55
      (a b)
55 55
      (ns1 ns2).num←55
      (ns1 ns2).num
55 55
```

Dictionary: Dotted expressions

Scalar extension:

```
      (a b)←55
      (a b)
55 55
      (ns1 ns2).num←55
      (ns1 ns2).num  ≡  (ns1.num ns2.num)
55 55
```

Dictionary: Dotted expressions

Right side of the dot can be any expression (that is, without ⋄) executed in the namespace(s) on the left side.

```
      (ns1 ns2)←(⎕NS⍬)(⎕NS⍬)
      ns1.(a b)←'Hello' 'World'
      ns1.(a,b)
HelloWorld
```

Dictionary: Dotted expressions

Right side of the dot can be any expression (that is, without ⋄) executed in the namespace(s) on the left side.

```
      ns2.a←{2/⍵}
      ns2.b←2 2⍴6 7 8 9
      ns2.(a,b)
6 6 7 7 8 8 9 9
```

Dictionary: Dotted expressions

Right side of the dot can be any expression (that is, without ⋄) executed in the namespace(s) on the left side.

```
      (2 3⍴ns1 ns2).(a,b)
```

HelloWorld	6 6 7 7 8 8 9 9	HelloWorld
6 6 7 7 8 8 9 9	HelloWorld	6 6 7 7 8 8 9

Exercise 1a: Creation Easy

Create a namespace `req` where `status` is `200` and `Method` is `{4+2×⍵}`.

```
      your ⋄ expressions
      req.status
200
      req.Method 200
404
```

Exercise 1b: Updating Easy

Within `req`, apply `Method` to `status`, and update `status` with the result.

```
      req.status
200
      your_expression
      req.status
404
```

Exercise 1c: Importing a workspace Medium

Write a function `Into` that copies a workspace into a namespace (using `⎕CY`).

```
      dfns←⎕NS⍬
      'dfns.dws' Into dfns
      dfns.disp dfns.morse'SOS'
┌───┬───┬───┐
│...│---│...│
└───┴───┴───┘
```

"By-value" versus "by-reference"

Non-namespaces have value semantics:

```
      A←1 2 3
      B←A
      A B
 1 2 3  1 2 3
```

arrays have the same value

```
      A←4 5 6
      A B
 4 5 6  1 2 3
```

B not affected

DYALOG

“By-value” versus “by-reference”

Non-namespaces have value semantics:

```
      A←1 2 3
      B←A
      A B
 1 2 3  1 2 3
```

arrays have the same value

```
      A[3]←9
      A B
 1 2 9  1 2 3
```

B not affected

“By-value” versus “by-reference”

Non-namespaces have value semantics:

```
A←3
```

“By-value” versus “by-reference”

Non-namespaces have value semantics:

```
      A←3 ⋄ f←A∘+
      f 100
103
      A←4
      f 100
103
      f←A∘+
      f 100
104
```

A was passed by value – f hasn’t changed

f has changed – not A

“By-value” versus “by-reference”

Non-namespaces have value semantics:

```
∇ {A}←foo A        arguments are implicitly local
  A[2]←9
∇

A←1 2 3
foo A
A                  A hasn't changed
1 2 3
```

“By-value” versus “by-reference”

Non-namespaces have value semantics:

```
    ∇ {A}←foo A      arguments are implicitly local
      A[2]←9
    ∇

    A←1 2 3
    A←foo A          reassigns A
    A
1 9 3
```

“By-value” versus “by-reference”

Namespaces have reference semantics

- Arrays are pure values, irrespective of the actual instance
- Except any namespace, which is a container

In the context of Dyalog APL:
namespaces **are** called *references* or *refs*

Name class: 2 is array, 3 is function, 9 is scalar reference

“By-value” versus “by-reference”

Namespaces have reference semantics

```
      ns1←⎕NS⍬
      ns2←ns1                    both names designate the same ns
      ns2.vec                    name vec undefined in ns2
VALUE ERROR

      ns1.vec←7 8 9
      ns2.vec                    name vec defined in ns2
7 8 9
```


DYALOG

“By-value” versus “by-reference”

Namespaces have reference semantics

```
      ns1←⎕NS⍬
      ns2←ns1
      ns1.vec←7 8 9

      ns2.vec[2]←10
      ns1.vec
7 10 9
```

both names designate the same ns

value in `ns1` has changed

“By-value” versus “by-reference”

Namespaces have reference semantics

```
      ns1←⎕NS⍬
      ns1.vec←7 8 9
      ∇ ns SetVec2 value
        ns.vec[2]←value
      ∇

      ns1 SetVec2 10
      ns1.vec
7 10 9
```

ns1 has been modified

“By-value” versus “by-reference”

Namespaces have reference semantics

```
      ns1←⎕NS⍬
      ns2←ns1
      ns1.vec←7 8 9
      ns1 SetVec2 100
      ∇ ns SetVec2 value
          ns.vec[2]←value
      ∇
      ns2.vec
7 100 9

      ns3←⎕NS⍬
      ns1 SetVec2 10
      ns3 SetVec2 1000
      ns1.vec
VALUE ERROR: Undefined name: vec
7 10 9
```

ns2 has been modified

new, unrelated namespace

ns1 has been modified

“By-value” versus “by-reference”

Namespaces have reference semantics

```
      ns1←⎕NS⍬
      ns1.vec←7 8 9
      ∇ SetVec3 vec
        vec[3]←10
      ∇

      SetVec3 ns1.vec
      ns1.vec
7 8 9
```

takes array

an array: passed by value

"By-value" versus "by-reference"

How to achieve "by-value" semantics with namespaces:

```
      ns1←⎕NS⍬
      ns1.vec←7 8 9

      ns3←⎕NS ns1
      ns3.vec[2]←10

      ]disp (ns1 ns3).vec
┌→────┬→─────┐
│7 8 9│7 10 9│
└~───→┴~────→┘
```

clone: make "deep" copy

only new copy changed

“By-value” versus “by-reference”

Namespaces have reference semantics

Allows you to:

- Track individual entities independently of their content
- Pass modifiable arguments to functions (use with care!)

"By-value" versus "by-reference"

Namespaces have reference semantics

Note the name class (`40⎕ATX`):

- depth-0 refs: 9
- other arrays: 2 (even if consisting entirely of refs)

“By-value” versus “by-reference”

Namespaces have reference semantics

Naming convention tip:

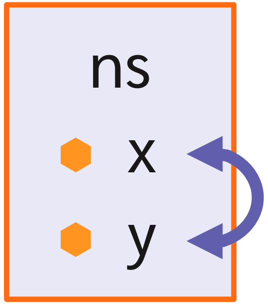
- Due to the different semantics, it may be worth naming “dottable” names in a recognisable way, for example:

`name_`

Exercise 2a: Swap names Easy

Write an *expression* that swaps the values of variables `x` and `y` in a namespace `ns` .

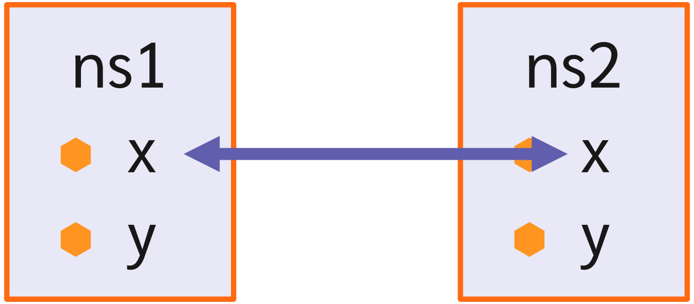
```
      ns←⎕NS⍬
      ns.x←10 ⋄ ns.y←20
      your_expression
      ns.x ns.y
20 10
```



Exercise 2b: Swap namespaces Easy

Write an *expression* that swaps the values of the variables named `x` in the namespaces `ns1` and `ns2`.

```
      ns1←⎕NS⍬ ⋄ ns2←⎕NS⍬
      ns1.x←10 ⋄ ns2.x←20
      ns1.y←30 ⋄ ns2.y←40
      your_expression
      ns1.x ns1.y ns2.x ns2.y
20 30 10 40
```



Exercise 2c: Testing if reference Medium

Write a function `ScalarRef` that returns a scalar Boolean value indicating whether its argument is a scalar namespace.

```
      ns←⎕NS⍬ ⋄ ns.a←10
      vec←ns.a 'abc' (ns ns) ns (⎕NS⍬) 42 ⍬ #
      ScalarRef¨ vec
0 0 0 1 1 0 0 1
```

Use one or more of these scalar namespace properties:

- `40⎕ATX` is 9
- `⎕DR` is 326
- Depth (≡) is 0 (simple scalar)
- Allows dot syntax (`ns.name`)

Exercise 2d: Indicate references Hard

Write a function `RefMask` that returns an array of the same structure as its argument, but with bits indicating any ns refs.

```
      ]disp RefMask (⊂⊂⊂⊂1 2 ns) 3 ns (2 2⍴'abc',⎕NS⍬)
┌→─────────────┬─┬─┬───┐
│┌────────────┐│0│1│0 0│
││┌──────────┐││ │ │0 1│
│││┌────────┐│││ │ │   │
││││┌─────┐ ││││ │ │   │
│││││0 0 1│ ││││ │ │   │
││││└~───→┘ ││││ │ │   │
```

Dictionary access under program control

```
value ← dictionary.name
```

get value

But how do we do this when the name can vary?

```
      ns←⎕NS⍬
      ns.foo←1 2 3

      name←'foo'
      10×ns⍎name
10 20 30
```

Dictionary access under program control

```
dictionary.name ← value
```

set value

But how do we do this when the name can vary?

```
      ns←⎕NS⍬
      name←'foo'

      ns⍎name,'←1 2 3'
      10×ns.foo
10 20 30
```

Exercise 3a: Get multiple values Medium

Write a function `Get` that takes a namespace as left argument and a vector of names as right argument and returns the corresponding values from the namespace.

```
      ns←⎕NS⍬
      ns.(foo bar baz)←(1 2 3) 'Hello' 42

      names←'foo' 'bar' 'foo' 'baz'
      ns Get names
 1 2 3  Hello  1 2 3  42
```

Exercise 3b: Set a value Medium

Write a function `Set` that takes a two-element `(name value)` vector as right argument, then does the corresponding assignment.

```
      Set 'my' (⍳5)
      10×my
10 20 30 40 50
```


DYALOG

Exercise 3c: Set in a namespace Medium

Improve `Set` so that it takes a namespace as left argument and does the corresponding assignment in that namespace.

```
      ns←⎕NS⍬
      ns Set 'my' (⍳5)
      10×ns.my
10 20 30 40 50
```

Exercise 3d: Set multiple values Hard

Improve `Set` so that it handles multiple two-element `(name value)` vectors as right argument.

```
      ns←⎕NS⍬
      ns Set ('yours' 'Hello')('mine' 'World')
      ns.(yours mine)
 Hello  World
```


DYALOG

Dictionary access under program control

Beware: ⍎ is potentially dangerous.

```
ns←⎕NS⍬
name←'⎕OFF⋄'     !!!
value←1 2 3

ns {⍺.{⍎⍺,'←⍵'}/⍵} name value
```

Solution: In production, validate your arguments (¯1≠4⊃⎕ATX).

— Look forward to ⎕VGET and ⎕VSET in version 20.0...

Dictionary access in version 20.0

Exercise 3a: Get multiple values

```
Get←⎕VGET
```

Exercise 3b: Set a value

```
Set←{⎕VSET⊂⍵} or Get←⎕VSET⊂
```

Exercise 3c: Set in a namespace

```
Set←{⍺⎕VSET⊂⍵} or Set←⎕VSET∘⊂
```

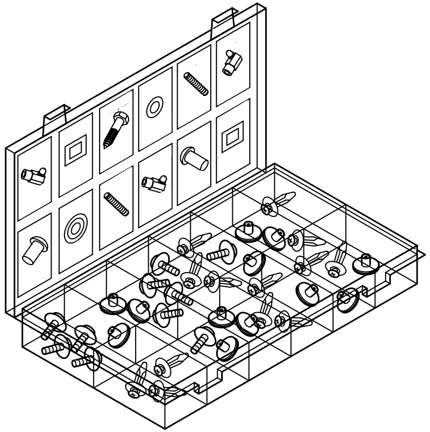
Exercise 3d: Set multiple values

```
Set←⎕VSET
```

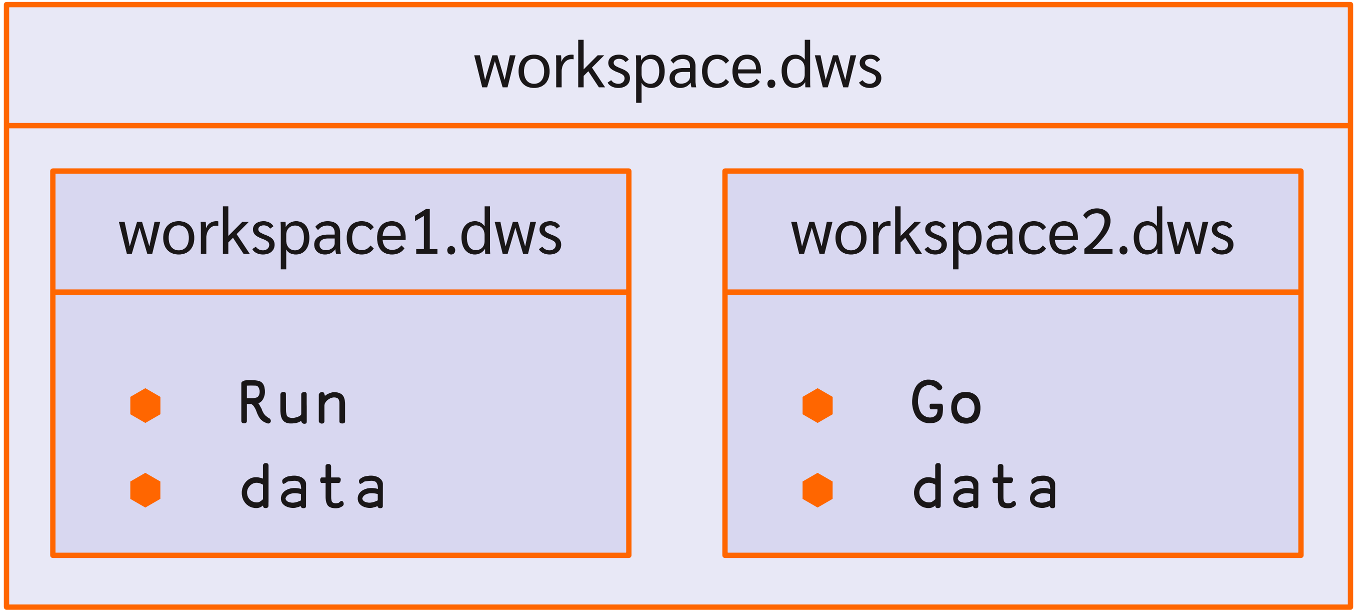
The namespace as a

structured container

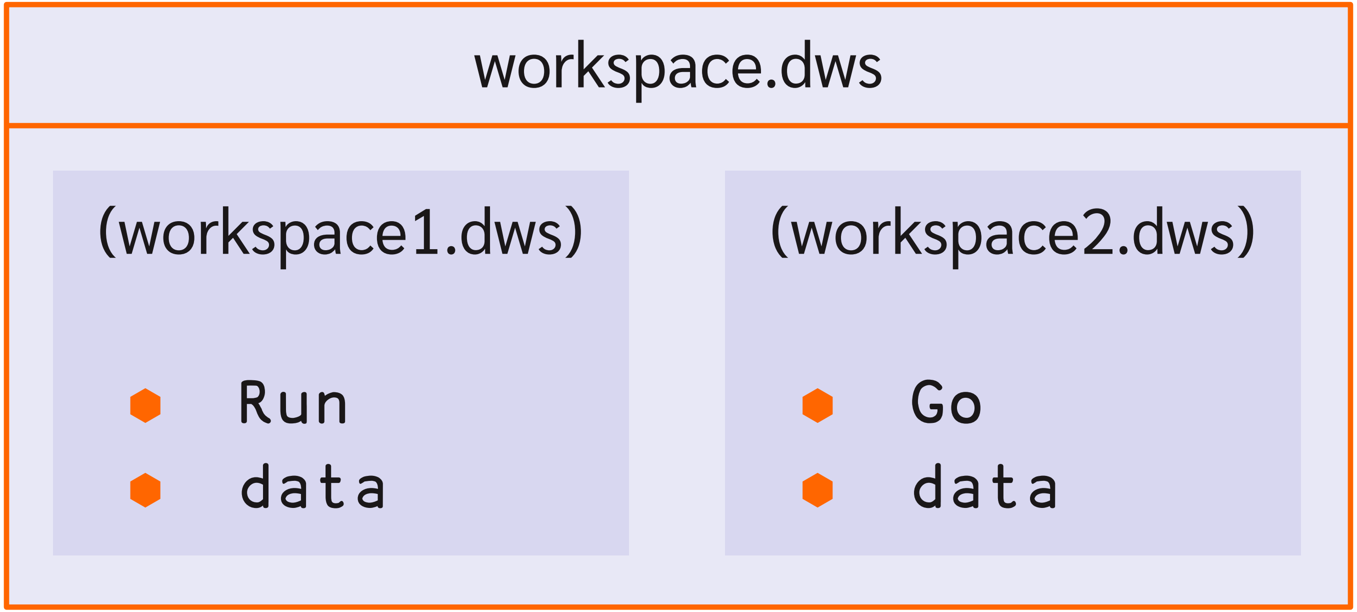
to organise your code



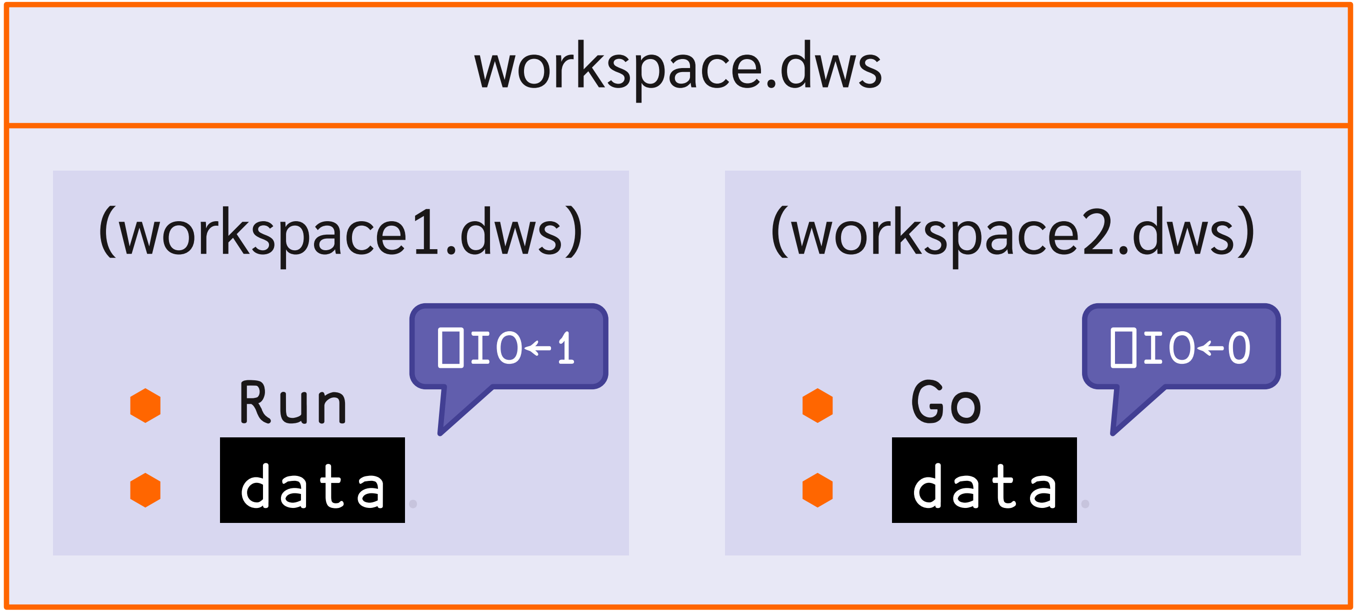
Structured container



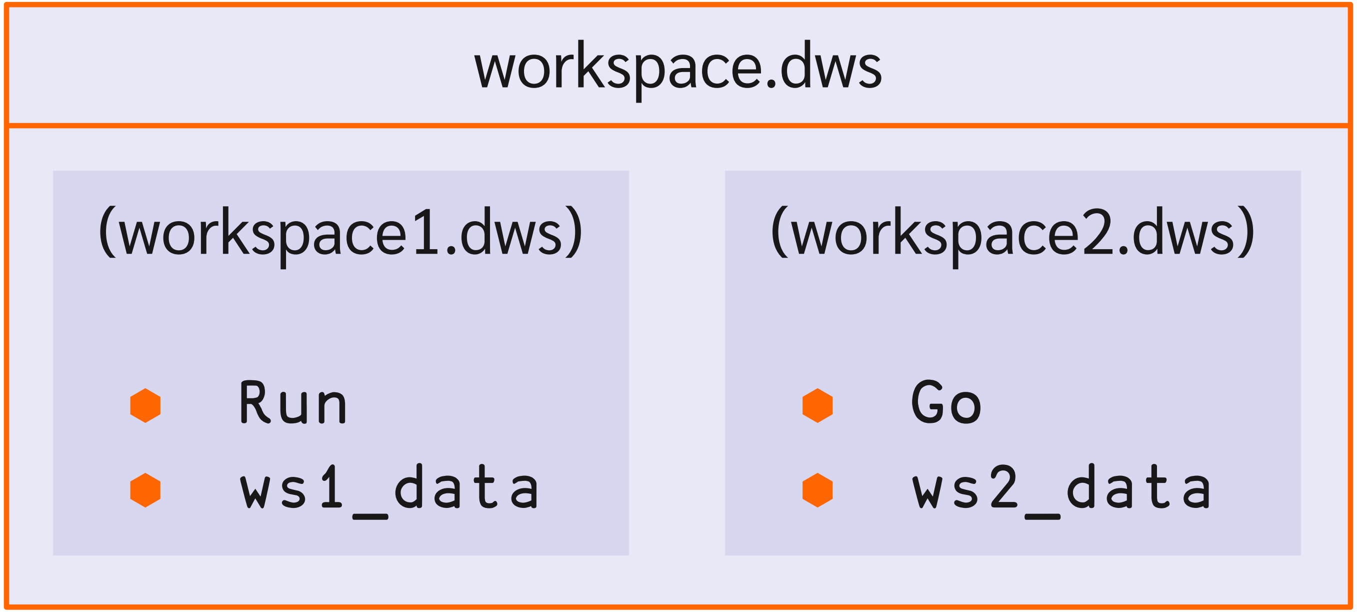
Structured container



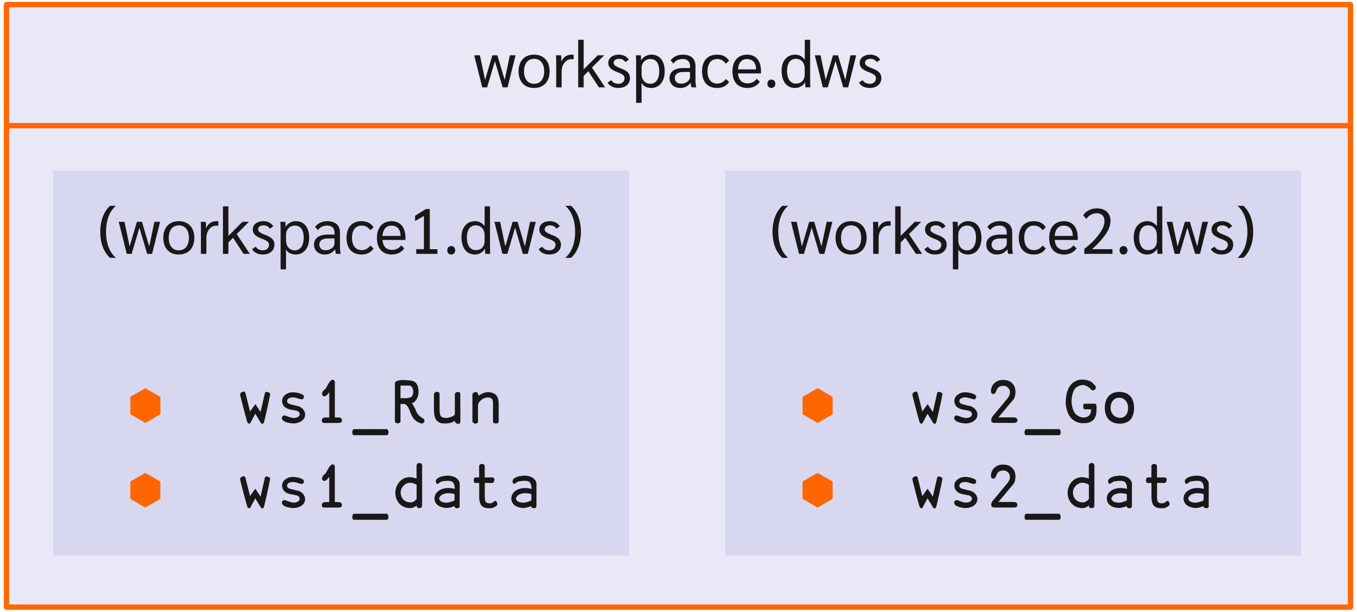
Structured container



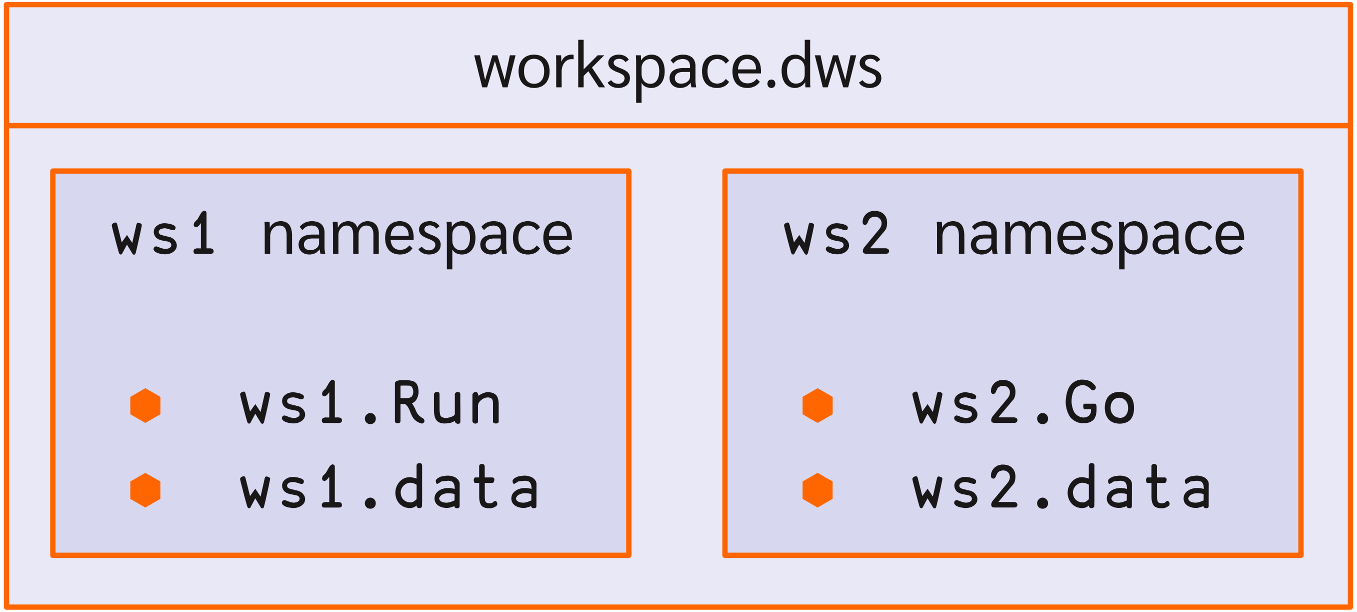
Structured container



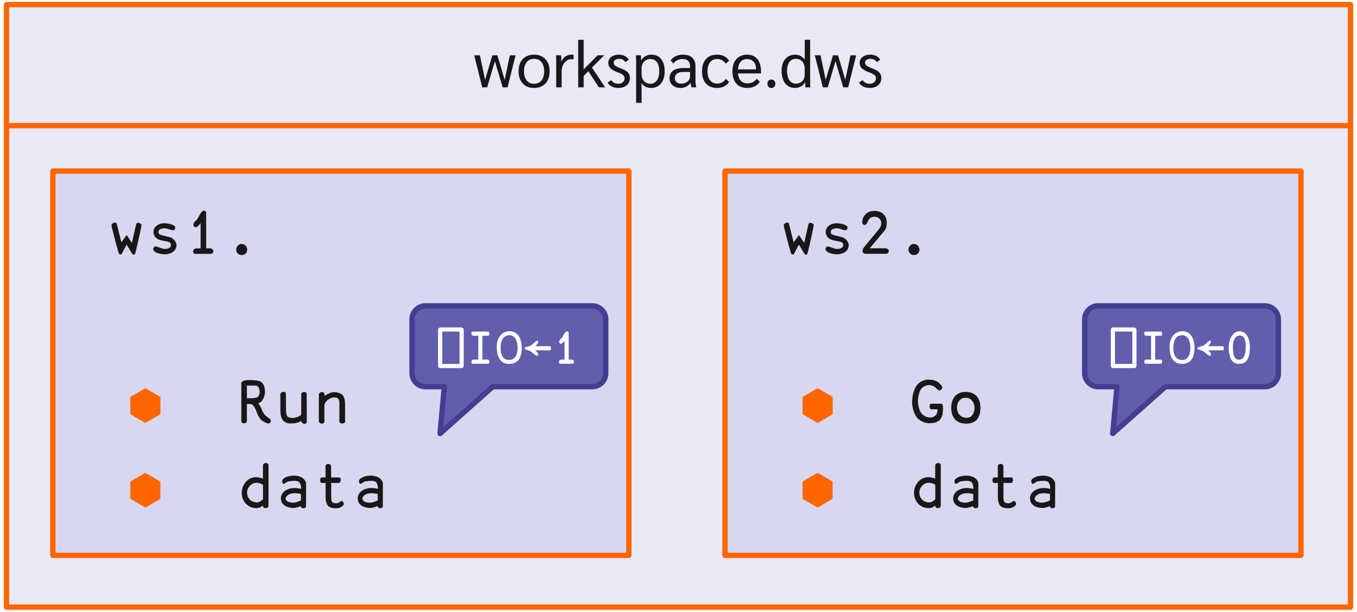
Structured container



Structured container



Structured container



Structured container

Tree	**Workspace**	**File system**	
Node	Namespace	Directory	
Leaf	Name	File	
		UNIX	WINDOWS
Roots	`# ⎕SE`	`/`	`C:\ D:\`
Separator	`.`	`/`	`\`
Parent node	`##`	`..`	`..`
Access current node	`⎕THIS`	`.`	`.`
Report current node	`⎕THIS`	`pwd`	`cd`

Structured container: actions

Tree	Workspace	File system	
Node	Namespace	Directory	
Leaf	Name	File	
		UNIX	WINDOWS
Create node	⎕NS	mkdir	md
Copy node	⎕NS	cp	copy
Change current node	⎕CS	cd	cd
List node contents	⎕NL	ls	dir
⋮	⋮	⋮	⋮

Structured container

```
      data←7 8 9
      ns←⎕NS⍬
      ns.data←'hello'
      (data)(ns.data)
┌─────┬─────┐
│7 8 9│hello│
└─────┴─────┘
```

```
      ⎕CS ns
      ⎕THIS
#.[Namespace]
      data
hello
      ⎕CS ##
      ⎕THIS
#
      data
7 8 9
```

Change Space (`⎕CS ns`)

Return to parent (`⎕CS ##`)

Structured container

```
      data←7 8 9
      ns←⎕NS⍬
      ns.data←'hello'
      (data)(ns.data)
┌─────┬─────┐
│7 8 9│hello│
└─────┴─────┘
      ns.(sub←⎕NS⍬)
      ns.sub.data←'x'
```

```
      ⎕CS ns      Change Space
      data
hello
      ##.data
7 8 9
      ⎕CS sub     Change Space
      data
x
      ##.data
hello
```

Structured container

```
      data←7 8 9
      ns←⎕NS⍬
      ns.data←'hello'
      (data)(ns.data)
┌─────┬─────┐
│7 8 9│hello│
└─────┴─────┘
      ns.(sub←⎕NS⍬)
      ns.sub.data←'x'
```

```
      ⎕CS ns
      sub.(data #.data)
┌─┬─────┐
│x│7 8 9│
└─┴─────┘
```

Change Space

Exercise 4a: Is argument a root? Easy

Write a function `IsRoot` that takes a namespace as argument that tells you whether that namespace is a root namespace.

```
      IsRoot ⎕SE.Dyalog.Utils
0
      IsRoot #
1
      IsRoot ⎕SE
1
```

Exercise 4b: What is my root? Medium

Write a function `FindRoot` that takes a namespace as argument and returns its root.

```
      FindRoot ⎕SE.Dyalog.Utils
⎕SE
      FindRoot #
#
      FindRoot ⎕NS⍬
#
```

Exercise 4c: Our roots? Easy — based on `FindRoot`

Write a function `FindRoots` that takes an arbitrary array of namespaces and finds the root for each namespace.

```
      FindRoots ⍪⎕SE.Dyalog.Utils(#,⎕NS⍬)⎕SE
┌───────┐
│⎕SE    │
├───────┤
│ #   # │
├───────┤
│⎕SE    │
└───────┘
```

Exercise 4d: Namespace lineage Hard

Write a function `Line` that takes a single namespace and returns its lineage (as a vector of refs) from root to leaf.

```
      Line ⎕SE.Dyalog.Utils
 ⎕SE   ⎕SE.Dyalog   ⎕SE.Dyalog.Utils

      Line ⎕SE.cbbot.bandsb2.sb.io
 ⎕SE   ⎕SE.cbbot   ⎕SE.cbbot.bandsb2
⎕SE.cbbot.bandsb2.sb   ⎕SE.cbbot.bandsb2.sb.io
```

Parent hierarchy

Namespaces have a single immutable parent (fixed at creation time) which is the namespace where ⎕NS was called.

```
      ns←⎕NS⍬
      #=ns.##                    created in #
1
      ns.sub1←ns.⎕NS⍬
      ns=ns.sub1.##              created in #.ns
1
```

Parent hierarchy

Namespaces have a single immutable parent (fixed at creation time) which is the namespace where ⎕NS was called.

```
      ns.sub1.(sub2←⎕NS⍬)
      ns.sub1=ns.sub1.sub2.##     created in #.ns.sub1
1
      ns.sub1.sub2.##.##.##
#
```


DYALOG

Parent hierarchy

```
ns0←⎕NS⍬
ns1←⎕NS⍬
ns1.ns2←ns0
```

Names in a namespace are **not** (necessarily) its children:

```
      ns1=ns1.ns2.##
0
```

Note: `ns1.ns2 ≡ ns0` which was created in #

DYALOG

Parent hierarchy

```
      ns0←⎕NS⍬
      ns1←⎕NS⍬
      ns1.ns2←ns0
```

Children are **not** (necessarily) named from their parent:

```
      ns3←ns1.ns2.⎕NS⍬
```

Note: there exists no name such that `ns3 ≡ ns3.##⍎name`:

```
      ns1.ns2.ns3
VALUE ERROR: Undefined name: ns3
```

Two approaches to namespaces:

Purpose:	**Code structure**	**Data dictionary**
Usage:	rooted	self-contained
Creation:	named	unnamed
Display:	full path	description
Hierarchy:	tree	flat

Each approach is easy, mixing them is tough — don’t try!