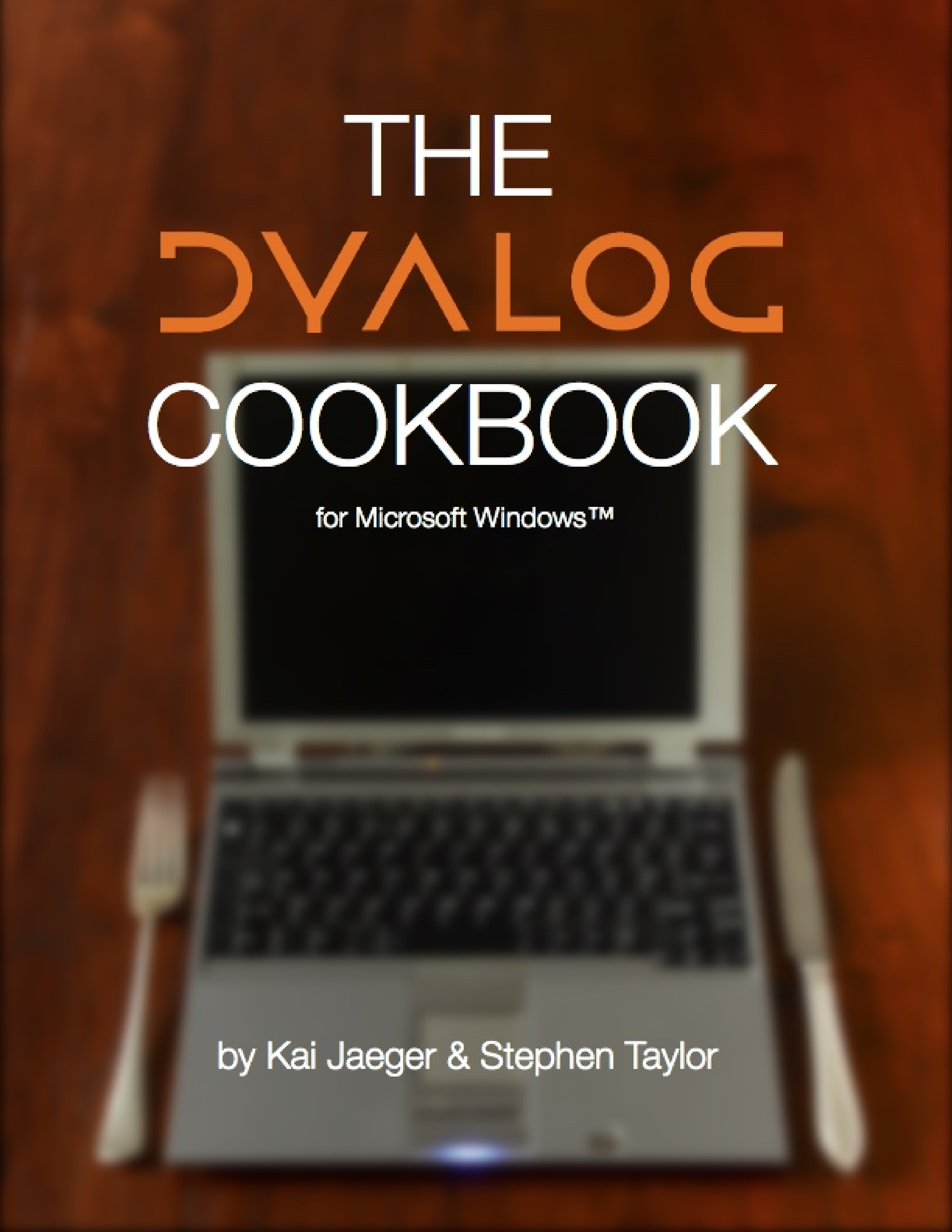


THE DYALOG COOKBOOK

for Microsoft Windows™

by Kai Jaeger & Stephen Taylor

The Dyalog Cookbook

Stephen Taylor and Kai Jaeger

With grateful thanks to Kenneth E. Iverson (1920-2004), for a language we can think in, and to his collaborators and followers who make it work.

Contents

Introduction

You want to write a Windows[1] application in Dyalog APL. You have already learned enough of the language to put some of your specialist knowledge into functions you can use in your work. The code works for you. Now you want to turn it into an application others can use. Perhaps even sell it.

This is where you need professional programming skills. How to install your code into an unknown computer. Have it catch, handle and log errors. Manage the different versions of your software as it evolves. Provide online help.

You are not necessarily a professional programmer. Perhaps you don't have those skills. Perhaps you need a professional programmer to turn your code into an application. But you've come a long way already. Perhaps you can get there by yourself - with *The Dyalog Cookbook*. Alternatively, you might be a professional programmer wondering how to solve these familiar problems in Dyalog APL.

The Dyalog Cookbook is about how to turn your Dyalog code into an application. We'll cover packaging your code into a robust environment. And we'll introduce some software development tools you've managed without so far, which will make your life easier.

You might continue as the sole developer of your application for a long time yet. But if it becomes a successful product you will eventually want other programmers collaborating on it. So we'll set up your code in a source-control system that will accommodate that. Even while you remain a sole developer, a source-control system will also allow you to roll back and recover from your own mistakes.

Not so long ago it was sufficient for an application to be packaged as an EXE that could be installed and run on other PCs. Nowadays many corporate clients run programs in terminal servers or in private clouds. So we'll look at how to organise your program to run as tasks that communicate with each other. Many applications written in Dyalog focus on some kind of numerical analysis, and can have CPU-intensive tasks. We'll look at how such tasks can be packaged to run either in background or on remote machines.

Method

It's conventional in this context for authors to assure readers that the techniques expounded here have been hammered out, proven and tested in many successful applications. That is true of individual components here, particularly of scripts from the APLTree library. But the development tools introduced by Dyalog in recent years are still finding their places with development teams.

[1]Perhaps one day you would like it to ship on multiple platforms. Perhaps one day we'll write that book too. Meanwhile, Microsoft Windows.

Some appear here in print for the first time. This book is the first sustained attempt to combine all the current Dyalog tools into an integrated approach.

Many of the issues addressed here are entangled with each other. We'll arrive at our best solutions by way of interim solutions. Proposing some wickedly intricate 'complete solution' framework does little to illuminate the problems it solves. So we'll add features – INI files, error handling, and so on – one at a time, and as we go we'll find ourselves revisiting the code that embeds the earlier features.

If you are an experienced Dyalog developer, you may be able to improve on what is described here. For this reason *The Dyalog Cookbook* remains for now an open-source project on GitHub[2].

Working through book, you get to understand how the implementation issues and the solutions to them work. In Part 1 you will find 'framework' code for your application, growing more complex as the book progresses. You can find scripts for these interim versions in the `code` folder on the book website. Watch out: they are interim solutions. You are of course welcome to simply copy and use the last version of the scripts. But there is much to be learned while stumbling.

Finally in Part 2 we'll introduce some professional writing techniques that might make maintaining your code easier – in what we hope will be a long useful future for it!

What you need to use the Dyalog Cookbook

- The Dyalog Version 15.0 Unicode interpreter or later.
- To know how to use namespaces, classes and instances. The utility code in the Cookbook is packaged as namespaces and classes. This is the form in which it is easiest for you to slide the code into your app without name conflicts. We recommend you use classes to organise your own code! But even if you don't, you need to know at least how to use classes. This is a deep subject, but all you need to know is the basics: creating an instance of a class and using its methods and properties, or just using the methods of a static class. See *Dyalog Programmer's Reference Guide* for an introduction.

We have not attempted to 'dumb down' our use of the language for readers with less experience. In a some cases we stop to discuss linguistic features; mostly not. If you see an expression you cannot read, a little experimentation and consultation of the reference material should show how it works. We encourage you to take the time to do this. Generally speaking – not invariably – short, crisp expressions are less work for you and the interpreter to read. Learn them and prefer them.

Acknowledgements

We are deeply grateful for contributions, ideas, comments and outright help from our colleagues, particularly from Gil Athoraya, Morten Kromberg, Nick Nickolov, and Paul Mansour.

[2]https://github.com/5jt/dyalog-cookbook

We jealously claim any errors as entirely our own work.

Stephen Taylor & Kai Jaeger

1. Packaging your application

Structure

In this chapter we consider your choices for making your program available to others, and for taking care of the source code, including tracking the changes through successive versions.

To follow this, we'll make a very simple program. It counts the frequency of letters used in one or multiple text files. (This is simple, but useful in cryptanalysis, at least at hobby level.) We'll put the source code under version control, and package the program for use.

Let's assume you've done the convenient thing. Your code is in a workspace. Everything it needs to run is defined in the workspace. Maybe you set a latent expression, so the program starts when you load the workspace.

In this chapter, we shall convert a DWS (saved workspace) to some DYALOG scripts and a DYAPP script to assemble an active workspace from them.

How can you distribute your program?

Send a workspace file (DWS)

Could not be simpler. If your user has a Dyalog interpreter, she can also save and send you the crash workspace if your program hits an execution error. But she will also be able to read your code – which might be more than you wish for.

If she doesn't have an interpreter, and you are not worried about her someday getting one and reading your code, and you have a Run-Time Agreement with Dyalog, you can send her the Dyalog Run-Time interpreter[ˆext] with the workspace. The Run-Time interpreter will not allow the program to suspend, so when the program breaks the task will vanish, and your user won't see your code. All right so far. But she will also not have a crash workspace to send you.

If your application uses multiple threads, the thread states can't be saved in a crash workspace anyway.

You need your program to catch and report any errors before it dies.

Send an executable file (EXE)

This is the simplest form of the program to install, because there is nothing else it needs to run: everything is embedded within the EXE. You export the workspace as an EXE, which can have the Dyalog Run-Time interpreter bound into it. The code cannot be read. As with the workspace-based runtime above, your program cannot suspend, so you need it to catch and report any errors before dying.

We'll do that!

Where should you keep the code?

Let's start by considering the workspace you will export as an EXE.

The first point is PCs have a lot of memory relative to your application code volume. So all your Dyalog code will be in the workspace. That's probably where you have it right now: all saved in a workspace.

Your workspace is like your desk top – a great place to get work done, but a poor place to store things. In particular it does nothing to help you track changes and revert to an earlier version.

Sometimes a code change turns out to be for the worse, and you need to undo it. Perhaps the change you need to undo is not the most recent change.

We'll keep the program in manageable pieces – 'modules' – and keep those pieces in text files under version control.

For this there are many *source-control management* (SCM) systems and repositories available. Subversion, GitHub and Mercurial are presently popular. These SCMs support multiple programmers working on the same program, and have sophisticated features to help resolve conflicts between them.

Whichever SCM you use (we used GitHub for writing this book and the code in it) your source code will comprise class and namespace scripts (DYALOGs) and a *build script* (DYAPP) to assemble them.

You'll keep your local working copy in whatever folder you please. We'll refer to this *working folder* as `Z:\` but it will of course be wherever suits you.

Versions

In real life you will produce successive *versions* of your program, each better than the last. In an ideal world, all your users will have and use the latest version. In that ideal world, you have only one version to maintain: the latest. In the real world, your users will have and use multiple versions. If you charge for upgrading to a newer version, this will surely happen. And even in your ideal world, you have to maintain at least two versions: the latest and the next.

What does it mean to maintain a version? At the very minimum, you keep the source code for it, so you could recreate its EXE from scratch, exactly as it was distributed. There will be things you want to improve, and perhaps bugs you must fix. Those will all go into the next version, of course. But some you may need to put into the released version and re-issue it to current users as a patch.

So in *The Dyalog Cookbook* we shall develop in successive versions. Our 'versions' are not ready to ship, so are probably better considered as milestones on the way to version 1.0. You could think of them as versions 0.1, 0.2 and so on. But we'll just refer to them as Versions 1, 2, and so on.

Our first version won't even be ready to export as an EXE. It will just recreate MyApp.DWS from scripts: a DYAPP and some DYALOGs. We'll call it Version 0.

The MyApp workspace

We suppose you already have a workspace in which your program runs. We mean to get from your workspace to class and namespace scripts and a build script.

We don't have your wonderful code to hand so we'll use ours. We'll use a very small and simple program, so we can focus on packaging the code as an application, not on writing the application itself.

Your application will of course be much more interesting!

So we'll begin with the MyApp workspace. It's trivially simple (we'll extend a bit what it does as we go) but for now it will stand in for your much more interesting code.

On encryption

Frequency counting relies on the distribution of letters being more or less constant for any given language. It is the first step in breaking a substitution cypher. Substitution cyphers have been superseded by public-private key encryption, and are mainly of historical interest, or for studying cryptanalysis. But they are also fun to play with.

We recommend *The Code Book: The secret history of codes & code-breaking* by Simon Singh and *In Code* by Sarah Flannery as introductions if you find this subject interesting.

From the `code\v01` folder on the book website load `LetterCount.dws`. Again, this is just the stand-in for your own code. Here's a quick tour.

Discussion

Function `TxtToCsv` takes the filepath of a TXT (text file) and writes a sibling CSV file containing the frequency count for the letters in the file. It uses function `CountLetters` to produce the table.

```
      ∆←'Now is the time for all good men'
      ∆,←' to come to the aid of the party.'
      MyApp.CountLetters ∆
N 2
O 8
W 1
I 3
S 1
T 7
H 3
E 6
```

```
M 3
F 2
R 2
A 3
L 2
G 1
D 2
C 1
P 1
Y 1
```

`CountLetters` returns a table of the letters in `⎕A` and the number of times each is found in the text. The count is insensitive to case and ignores accents, mapping accented to unaccented characters:

```
      ACCENTS
ÁÂÃÀÄÅÇĐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ
AAAAAACDEEEEIIIINOOOOOOUUUUY
```

That amounts to five functions. Two of them are specific to the application: `TxtToCsv` and `CountLetters`. The other three – `caseUp`, `join`, and `map` are utilities, of general use.

`toUppercase` uses the fast case-folding I-beam introduced in Dyalog 15.0.

`TxtToCsv` uses the file-system primitives `⎕NINFO`, `⎕NGET`, and `⎕NPUT` introduced in Dyalog 15.0.

`TxtToCsv` observes *Cannon's Canon*, which tells us to avoid representing options with numeric constants. Instead, it assigns descriptive names at the start and uses those. That's not too bad, but if every function did the same, we risk multiple definitions of the same constant, breaching the DRY principle – don't repeat yourself. (And open to errors: *A man with two watches never knows what time it is.*) We can do better by defining all the Dyalog constants we want to use in a single namespace in the root.

How to organise the code

To expand this program into distributable software we're going to add features, many of them drawn from the APLTree library[3]. To facilitate that we'll first organise the existing code into script files, and write a *build script* to assemble a workspace from them.

Start at the root. We're going to be conservative about defining names in the root of the workspace. Why? Right now the program stands by itself and can do what it likes in the workspace. But in the future your program might become part of a larger APL system. In that case it will share the workspace root with other objects you don't know anything about right now.

[3]http://archive.vector.org.uk/art10500730

So your program will be a self-contained object in the workspace root. Give it a distinctive name, not a generic name such as `Application` or `Root`. From here on we'll call it `MyApp`. (We know: almost as bad.)

But there *are* other objects you might define in the root. If you're using classes or namespaces that other systems might also use, define them in the root. For example, if `MyApp` should one day become a module of some larger system, it would make no sense for each module to have its own copy of, say, the APLTree class `Logger`.

With this in mind, let's distinguish some categories of code, and how the code in `MyApp` will refer to them.

General utilities and classes
: For example, the `APLTreeUtils` namespace and the `Logger` class. (Your program doesn't yet use these utilities.) In the future, other programs, possibly sharing the same workspace, might use them too.

Your program and its modules
: Your top-level code will be in `#.MyApp`. Other modules and MyApp-specific classes may be defined within it.

Tools and utility functions specific to MyApp
: These might include your own extensions to Dyalog namespaces or classes. Define them inside the app object, eg `#.MyApp.Utils`.

Your own language extensions and syntax sweeteners
: For example, you might like to use functions `means` and `else` as simple conditionals. These are effectively your local *extensions* to APL, the functions you expect always to be around. Define your collection of such functions into a namespace in the root, eg `#.Utilities`.

The object tree in the workspace might eventually look something like:

```
#
|-⊛Constants
|-⊛APLTreeUtils
|-⊛Utilities
|-○MyApp
| |-⊛Common
| |-⊛Engine
| |-○TaskQueue
| \-⊛UI
| \-⊛Utils
\-○Logger
```

⊛ denotes a namespace, ○ a class.

The objects in the root are ‘public’. They comprise `MyApp` and objects other applications might use. (You might add another application that uses `#.Utilities`. Everything else is encapsulated within `MyApp`. Here’s how to refer in the MyApp code to these different categories of objects.

1. `log←⎕NEW #.Logger`
2. `queue←⎕NEW TaskQueue`
3. `tbl←Engine.CountLetters txt`
4. `foo←(bar>3) #.Utilities.means 'ok' #.Utilities.else 'error'`

That last is pretty horrible. It needs some explanation.

Many languages offer a short-form syntax for if/else, eg (JavaScript, PHP)

```
foo = bar>3 ? 'ok' : 'error' ;
```

Some equivalents in Dyalog:

```
:If bar>3 ⋄ foo←'ok' ⋄ :Else ⋄ foo←'error' ⋄ :EndIf
foo←(⎕IO+bar>3)⊃'error' 'ok'
foo←⊃(bar>3)⌽'error' 'ok'
```

`means` and `else` here provide a short-form syntax:

```
foo←(bar>3) means 'ok' else 'error'
```

The readability gain is largely lost if we have to qualify the functions with their full paths:

```
foo←(bar>3) #.Utilities.means 'ok' #.Utilities.else 'error'
```

We can improve it by defining aliases within `#.MyApp`:

```
(C U)←#.(Constants Utilities)
```

allowing (4) to be written as

```
foo←(bar>3) U.means 'ok' U.else 'error'
```

Why not use ⎕PATH?

`⎕PATH` tempts us. We could set `⎕PATH←'#.Utilities'`. The expression above could then take its most readable form:

```
foo←(bar>3) means 'ok' else 'error'
```

Trying to resolve the names `means` and `else`, the interpreter would consult `⎕PATH` and find them in `#.Utilities`. So far so good: this is what `⎕PATH` is designed for. It works fine in simple cases, but it will lead us into problems later:

- As long as each name leads unambiguously to an object, shift-clicking on it will display it in the editor, a valuable feature of APL in development and debugging. The editor allows us to change code during execution, and save those changes back to the scripts. But `⎕PATH` can interfere with this and break that valuable connection.
- Understanding the scope of the space in which a GUI callback executes can be challenging enough; introducing `⎕PATH` makes it harder still.

Project Gutenberg

We'll raid Project Gutenberg[4] for some texts to read.

We're tempted by the complete works of William Shakespeare but we don't know that letter distribution stayed constant over four centuries. Interesting to find out, though, so we'll save a copy as `Z:\texts\en\shakespeare.dat`. And we'll download some 20th-century books as TXTs into the same folder. Here are some texts we can use.

```
      ↑⊃(⎕NINFO⍠'Wildcard' 1) 'z:\texts\en\*.txt'
z:/texts/en/ageofinnocence.txt
z:/texts/en/dubliners.txt
z:/texts/en/heartofdarkness.txt
z:/texts/en/metamorphosis.txt
z:/texts/en/pygmalion.txt
z:/texts/en/timemachine.txt
z:/texts/en/ulysses.txt
z:/texts/en/withthesehands.txt
z:/texts/en/wizardoz.txt
```

[4]https://www.gutenberg.org/

MyApp reloaded

We'll first make MyApp a simple 'engine' that does not interact with the user. Many applications have functions like this at their core. Let's enable the user to call this engine from the command line with appropriate parameters. By the time we give it a user interface, it will already have important capabilities, such as logging errors and recovering from crashes.

Our engine will be based on the `TxtToCsv` function. It will take one parameter, a fully-qualified filepath for a folder or file. If a file, it will write a sibling CSV. If a folder, it will read any TXT files in the folder, count the letter frequencies and report them as a CSV file sibling to the folder. Simple enough. Here we go.

Building from a DYAPP

In your text editor open a new document. (Or you can take the DYAPP from the folder for this chapter.)

You need a text editor that handles Unicode. If you're not already using a Unicode text editor, Windows' own Notepad will do for occasional use. (Set the default font to APL385 Unicode.) For a full-strength multifile text editor Notepad++[5] works well.

Here's how the object tree will look:

```
#
|-⊛Constants
|-⊛Utilities
\-⊛MyApp
```

See the *SALT User Guide* for more about DYAPPs.

The file tree will look like this:

[5]https://notepad-plus-plus.org/

```
z:\code\v01\Constants.dyalog
z:\code\v01\MyApp.dyalog
z:\code\v01\Utilities.dyalog
z:\code\v01\MyApp.dyapp
```

So `z:\code\v01\MyApp.dyapp` looks like this:

```
Target #
Load Constants
Load Utilities
Load MyApp
```

and the DYALOGs look like this. `Constants.dyalog`:

You can download all the scripts in this chapter from the corresponding folder in the book website. Or create the namespaces in the session and use SALT to save them to files.

```
:Namespace Constants
    ⍝ Dyalog constants

    :Namespace NINFO

        ⍝ left arguments
        NAME←0
        TYPE←1
        SIZE←2
        MODIFIED←3
        OWNER_USER_ID←4
        OWNER_NAME←5
        HIDDEN←6
        TARGET←7
        :Namespace TYPES
            NOT_KNOWN←0
            DIRECTORY←1
            FILE←2
            CHARACTER_DEVICE←3
            SYMBOLIC_LINK←4
            BLOCK_DEVICE←5
            FIFO←6
            SOCKET←7
        :EndNamespace

    :EndNamespace

   :Namespace NPUT
      OVERWRITE←1
   :EndNamespace

:EndNamespace
```

Later on we'll introduce a more convenient way to represent and maintain the definitions of constants. This will do nicely for now.

And `Utilities.dyalog`:

```
:Namespace Utilities

    map←{
          (old new)←⍺
          nw←∪⍵
          (new,nw)[(old,nw)⍳⍵]
      }

    toLowercase←0∘(819⌶)
    toUppercase←1∘(819⌶)

:EndNamespace
```

And another:

```
:Namespace MyApp

⍝ Aliases
    (C U)←#.(Constants Utilities) ⍝ must be already defined

⍝ === VARIABLES ===

    ACCENTS←↑'ÁÂÃÀÄÅÇĐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ' 'AAAAAACDEEEEIIIINOOOOOOUUUUY'

⍝ === End of variables definition ===

    CountLetters←{
      {⍺(≢⍵)}⌸⎕A{⍵⌿⍨⍵∊⍺}(↓ACCENTS)U.map U.toUppercase ⍵
    } ⍝ Table of letter counts in string ⍵


    ∇ {ok}←TxtToCsv fullfilepath;xxx;csv;stem;path;files;txt;type;lines;nl
      ;enc;tgt;src;tbl
   ⍝ Write a sibling CSV of the TXT located at fullfilepath,
   ⍝ containing a frequency count of the letters in the file text
      csv←'.csv'
      :Select type←C.NINFO.TYPE ⎕NINFO fullfilepath
      :Case 1 ⍝ folder
          tgt←fullfilepath,csv
          files←⊃(⎕NINFO⍠'Wildcard' 1)fullfilepath,'\*.txt'
      :Case 2 ⍝ file
          (path stem xxx)←⎕NPARTS fullfilepath
          tgt←path,stem,csv
          files←,⊂fullfilepath
      :EndSelect
      tbl←0 2⍴'A' 0
      :For src :In files
          (txt enc nl)←⎕NGET src
          tbl⍪←CountLetters txt
      :EndFor
      lines←{⍺,',',⍕⍵}/⊃{⍺(+/⍵)}⌸/↓[1]tbl
      ok←×(lines enc nl)⎕NPUT tgt C.NPUT.OVERWRITE
    ∇

:EndNamespace
```

Launch the DYAPP by double-clicking on its icon in Windows Explorer. Examine the active session. We see

```
- Constants
  - NINFO
    - NAME
    - ...
    - TYPES
      - NOT_KNOWN
      - DIRECTORY
      - ...
  - NPUT
    - OVERWRITE
- MyApp
  - ACCENTS
  - CountLetters
  - TxtToCsv
- Utilities
  - map
  - toLowercase
  - toUppercase
```

If you also see namespaces `SALT_Data` ignore them. They are part of how the Dyalog editor updates script files.

We have converted the saved workspace to a DYAPP that assembles the workspace from DYALOGs. We can use `MyApp.dyapp` anytime to recreate the app as a workspace. But we have not saved a workspace. We will always assemble a workspace from scripts.

Package MyApp as an executable

For Version 1.0 we'll package `MyApp` as an EXE. Version 1.0 will run from the command line and it will run 'headless' – without a user interface (UI). It won't have a session either.

Output to the session log

What happens to values that would otherwise be written in the session log? They disappear. That's not actually a problem, but it is tidy to catch anything that would otherwise be written to the UI, including empty arrays.

`TxtToCsv` has a shy result, so it won't write its result to the session. That's fine.

We'll also fix three key environment variables for it in `MyApp`:

```
(⎕IO ⎕ML ⎕WX)←1 1 3 ⍝ environment variables
```

Reading arguments from the command line

`TxtToCsv` needs an argument. The EXE must take it from the command line. We'll give `MyApp` a niladic function `StartFromCmdLine`. The DYAPP will use it to start the program:

```
Target #
Load Constants
Load Utilities
Load MyApp
Run MyApp.SetLX
```

and in `MyApp.dyalog`:

```
 ∇ SetLX
⍝ Set Latent Expression in root ready to export workspace as EXE
 #.⎕LX←'MyApp.StartFromCmdLine'
 ∇

 ∇ StartFromCmdLine;args
⍝ Read command parameters, run the application
   {}TxtToCsv 2⊃2↑⌷2 ⎕NQ'.' 'GetCommandLineArgs'
 ∇
```

This is how MyApp will run when called from the Windows command line.

We're now nearly ready to export the first version of our EXE.

1. From the File menu pick *Export*.
2. Pick `Z:\` as the destination folder.
3. From the list *Save as type* pick *Standalone Executable*.
4. Set the *File name* as `MyApp`.
5. Check the *Runtime application* and *Console application* boxes.
6. Click *Save*.

You should see an alert message: *File Z:\MyApp.exe successfully created.* (This occasionally fails for no obvious reason. If it does, delete or rename any prior version and try again.)

Use the *Version* button to bind to the EXE information about the application, author, version, copyright and so on. Specify an icon file to replace the Dyalog icon with one of your own.

Let's run it. From a command line:

```
c:\Users\A.N. Other>CD Z:\
Z:\>MyApp.exe texts\en
```

Looking in Windows Explorer at `Z:\texts\en.csv`, we see its timestamp just changed. Our EXE works!

Logging what happens

MyApp 1.0 is now working, but handles errors poorly. See what happens when we try to work on a non-existent folder. In the command shell

```
Z:\code\v01>MyApp.exe Z:\texts\de
```

We see an alert message: *This Dyalog APL runtime application has attempted to use the APL session and will therefore be closed.*

MyApp failed, because there is no folder `Z:\texts\de`. That triggered an error in the APL code. The interpreter tried to signal the error to the session. But a runtime task has no session, so at that point the interpreter popped the alert message and MyApp died.

MyApp 2.0 could do better. In several ways.

- Have the program write a log file recording what happens.
- Set traps to catch and log foreseeable problems.
- Set a top-level trap to catch and report unforeseen errors and save a crash workspace for analysis.

Save a copy of `Z:\code\v01` as `Z:\code\v02`.

Start with the log file. We'll use the APLTree `Logger` class, which we'll now install in the workspace root. If you've not already done so, copy the APLTree library folder into `Z:\code`.[6] Now edit `Z:\code\v02\MyApp.dyapp` to include some library code:

```
Target #
Load ..\AplTree\APLTreeUtils
Load ..\AplTree\FilesAndDirs
Load ..\AplTree\Logger
Load Constants
Load Utilities
Load MyApp
Run MyApp.SetLX
```

and run the DYAPP to recreate the MyApp workspace.

The `Logger` class is now part of MyApp. Let's get the program to log what it's doing.

Within `MyApp`, some changes. Some aliases for the new code.

[6]You can download the complete APLTree library from the APL Wiki.

```
⍝ Aliases (referents must be defined previously)
    (A F L)←#.(APLTreeUtils FilesAndDirs Logger) ⍝ from APLTree
    (C U)←#.(Constants Utilities)
```

Where to keep the logfiles?

Where is MyApp to write the logfile? We need a filepath we know exists. That rules out `fullfilepath`. We need a logfile even if that isn't a valid path.

We'll write logfiles into a subfolder of the Current Directory. Where will that be?

When the EXE launches, the Current Directory is set by the command shell. Eg:

```
Z:\code\v02>MyApp.exe Z:\texts\en
```

Current Directory is `Z:\code\v02` and the logfiles will appear in `Z:\code\v02\Logs`.

```
Z:>code\v02\MyApp.exe texts\en
```

Current Directory is `Z:` and the logfiles will appear in `Z:\Logs`.

If this version of MyApp were for shipping that would be a problem. An application installed in `C:\Program Files` cannot rely on being able to write logfiles there. That is a problem to be solved by an installer. We'll come to that later. But for this version of MyApp the logfiles are for your eyes only. It's fine that the logfiles appear wherever you launch the EXE. You just have know where they are.

In developing and testing MyApp, we create the active workspace by running `MyApp.dyapp`. The interpreter sets the Current Directory of the active workspace as the DYAPP's parent folder. That too is sure to exist.

```
      #.FilesAndDirs.PWD
Z:\code\v02
```

We need `TxtToCsv` to ensure the Current Directory contains a `Logs` folder.

```
      'CREATE!' F.CheckPath 'Logs' ⍝ ensure subfolder of current dir
```

Now we set up the parameters for the Logger object. First we use the Logger class' sttaic `CreatePropertySpace` method to get a property space (object) with an initial a set of default parameters. We then modify those and use the object to create the Logger object. You can use the property space's `List` method to display its properties.)

If `TxtToCsv` can log what it's doing, it makes sense to check its argument. We wrap the earlier version of the function in an if/else:

```
     ∇ {ok}←TxtToCsv fullfilepath;∆;csv;stem;path;files;txt;type;lines;nl
         ;enc;tgt;src;tbl
    ⍝ Write a sibling CSV of the TXT located at fullfilepath,
    ⍝ containing a frequency count of the letters in the file text
       'CREATE!'F.CheckPath'Logs' ⍝ ensure subfolder of current dir
       ∆←L.CreatePropertySpace
       ∆.path←'Logs\' ⍝ subfolder of current directory
       ∆.encoding←'UTF8'
       ∆.filenamePrefix←'MyApp'
       ∆.refToUtils←#
       Log←⎕NEW L(,⊂∆)
       Log.Log'Started MyApp in ',F.PWD
       Log.Log'Source: ',fullfilepath

       :If ~⎕NEXISTS fullfilepath
           Log.Log'Invalid source'
           ok←0
       :Else

           ⍝ as before...

       :EndIf
     ∇
```

Notice how defining the aliases `A`, `L`, and `W` in the namespace script – the environment of `StartFromCmdLine` and `TxtToCsv` – makes the function code more legible.

The foreseeable error that aborted the runtime task – on an invalid filepath – has now been replaced by log messages to say no files were found.

As the logging starts and ends in `TxtToCsv`, we can run this in the workspace now to test it.

```
      #.MyApp.TxtToCsv 'Z:\texts\en'
      ⊃(⎕NINFO⍠1) 'Logs\*.LOG'
 MyApp_20160406.log
      ↑⎕NGET 'Logs\MyApp_20160406.log'
2016-04-06 13:42:43 *** Log File opened
2016-04-06 13:42:43 (0) Started MyApp in Z:\code\v02
2016-04-06 13:42:43 (0) Source: Z:\texts\en
2016-04-06 13:42:43 (0) Target: Z:\texts\en.csv
2016-04-06 13:42:43 (0) 244 bytes written to Z:\texts\en.csv
2016-04-06 13:42:43 (0) All done
```

Let's see if this works also for the exported EXE. Run the DYAPP to rebuild the workspace. Export as before, but to `Z:\code\v02`, and in the Windows command line run the new `MyApp.exe`.

```
    Z:\code\v02>MyApp.exe Z:\texts\en
```

Yes! The output TXT gets produced as before, and the work gets logged in `Z:\code\v02\Logs`.

Let's see what happens now when the filepath is invalid.

```
    Z:\code\v02>MyApp.exe Z:\texts\de
```

No warning message – the program made an orderly finish. And the log?

```
      ↑⎕NGET 'Logs\MyApp_20160406.log'
2016-04-06 13:42:43 *** Log File opened
2016-04-06 13:42:43 (0) Started MyApp in Z:\code\v02
2016-04-06 13:42:43 (0) Source: Z:\texts\en
2016-04-06 13:42:43 (0) Target: Z:\texts\en.csv
2016-04-06 13:42:43 (0) 244 bytes written to Z:\texts\en.csv
2016-04-06 13:42:43 (0) All done
2016-04-06 13:42:50 *** Log File opened
2016-04-06 13:42:50 (0) Started MyApp in Z:\code\v02
2016-04-06 13:42:50 (0) Source: Z:\texts\de
2016-04-06 13:42:50 (0) Invalid source
```

Yes!

We now have MyApp logging its work in a subfolder of the application folder and reporting problems which it has anticipated.

Next we need to consider how to handle and report errors we have *not* anticipated. We should also return some kind of error code to Windows. If MyApp encounters an error, any process calling it needs to know.

Handling errors

MyApp now anticipates, tests for and reports certain foreseeable problems with the parameters. We'll now move on to handle errors more comprehensively.

Save a copy of `Z:\code\v02` as `Z:\code\v03`.

What are we missing?

1. Other problems are foreseeable. The file system is a rich source of ephemeral problems and displays. Many of these are caught and handled by the APLTree utilities. They might make several attempts to read or write a file before giving up and signalling an error. Hooray. We need to handle the events signalled when the utilities give up.
2. The MyApp EXE terminates with an all-OK zero exit code even when it has caught and handled an error. It would be a better Windows citizen if it returned custom exit codes, letting a calling program know how it terminated..
3. By definition, unforeseen problems haven't been foreseen. But we foresee there will be some! A mere typo in the code could break execution. We need a master trap to catch any events that would break execution, save them for analysis, and report them in an orderly way.

Inspecting Windows exit codes

How do you see the exit code returned to Windows? You can see it in the command shell like this:

```
Z:\code\v03>MyApp.exe Z:\texts\en

Z:\code\v03>echo Exit Code is %errorcode%
Exit Code is 0

Z:\code\v03>MyApp.exe Z:\texts\de

Z:\code\v03>echo Exit Code is %errorcode%
Exit Code is 101
```

Foreseen errors

We'll start with the second item from the list above. Now the result of `TxtToCsv` gets passed to `⎕OFF` to be returned to the operating system as an exit code.

```
∇ StartFromCmdLine;exit;args
 ⍝ Read command parameters, run the application
  ⎕WSID←'MyApp'
  args←⌷2 ⎕NQ'.' 'GetCommandLineArgs'
  Off TxtToCsv 2⊃2↑args
∇

∇ Off returncode
  :If #.A.IsDevelopment
    →
  :Else
    ⎕OFF returncode
  :Endif
∇
```

And we'll define in `#.MyApp` a child namespace of exit codes.

```
:Namespace EXIT
    OK←0
    INVALID_SOURCE←101
    UNABLE_TO_READ_SOURCE←102
    UNABLE_TO_WRITE_TARGET←103
:EndNamespace
```

We define an `OK` value of zero for completeness. (We really *are* trying to eliminate from our functions numerical constants that the reader has to interpret.) In Windows, an exit code of zero is a normal exit. All the exit codes are defined in this namespace. The function code can refer to them by name, so the meaning is clear. And this is the *only* definition of the exit-code values.

> We could have defined `EXIT` in `#.Constants`, but we reserve that script for Dyalog constants, keeping it as a component that could be used in other Dyalog applications. *These* exit codes are specific to MyApp, so are better defined in `#.MyApp`.

`TxtToCsv` still starts and stops the logging, but it now calls `CheckAgenda` to examine what's to be done, and `CountLettersIn` to do them. Both these functions use the function `Error`, local to `TxtToCsv`, to log errors.

```
∇ exit←TxtToCsv fullfilepath;∆;isDev;Log;LogError;files;tgt
 ⍝ Write a sibling CSV of the TXT located at fullfilepath,
 ⍝ containing a frequency count of the letters in the file text
  'CREATE!'F.CheckPath'Logs' ⍝ ensure subfolder of current dir
  ∆←L.CreatePropertySpace
  ∆.path←'Logs',F.CurrentSep ⍝ subfolder of current directory
  ∆.encoding←'UTF8'
  ∆.filenamePrefix←'MyApp'
  ∆.refToUtils←#
  Log←⎕NEW L(,⊂∆)
  Log.Log'Started MyApp in ',F.PWD
  Log.Log'Source: ',fullfilepath

  LogError←Log∘{code←EXIT⍎⍵ ⋄ code⊣⍺.LogError code ⍵}

  :If EXIT.OK=⊃(exit files)←CheckAgenda fullfilepath
      Log.Log'Target: ',tgt←(⊃,/2↑⎕NPARTS fullfilepath),'.CSV'
      exit←CountLettersIn files tgt
  :EndIf
  Log.Log'All done'
∇
```

Note the exit code is tested against `EXIT.OK`. Testing `0=exit` would work and read as well, but relies on `EXIT.OK` being 0. The point of defining the codes in `EXIT` is to make the functions relate to the exit codes only by their names.

See *Delta, the Heracitlean variable* in Part 2 for a discussion of how and why we use the `∆` variable.

`CheckAgenda` looks for foreseeable errors. In general, we like functions to *start at the top and exit at the bottom.* Returning from the middle of a function can lead to confusion. We'll come back to this topic in a moment. `CheckAgenda` follows a common pattern of validation logic: a cascade of tests with corresponding actions to handle the error, terminating in an 'all clear'.

```
∇ (exit files)←CheckAgenda fullfilepath;type
  :If ~(type←C.NINFO.TYPE ⎕NINFO fullfilepath)∊C.NINFO.TYPES.(DIRECTORY FILE)
      (files exit)←(LogError 'INVALID_SOURCE')('')
  :ElseIf ~⎕NEXISTS fullfilepath
      (files exit)←(LogError 'SOURCE_NOT_FOUND')('')
  :Else
      :Select type
      :Case C.NINFO.TYPES.DIRECTORY
          files←⊃(⎕NINFO⍠'Wildcard' 1)fullfilepath,'\*.txt'
      :Case C.NINFO.TYPES.FILE
          files←,⊂fullfilepath
      :EndSelect
      exit←EXIT.OK
  :EndIf
∇
```

CountLettersIn can get to work now knowing its arguments are valid. But it's working with the file system, and valid file operations can fail for all sorts of reasons, including unpredictable and ephemeral network conditions. So we set traps to catch and report failures.

```
∇ exit←CountLettersIn (files tgt);i;txt;tbl;enc;nl;lines;bytes
 ⍝ Exit code from writing a letter-frequency count for a list of files
  tbl←0 2⍴'A' 0
  exit←EXIT.OK ⋄ i←1
  :While exit=EXIT.OK
      :Trap 0
          (txt enc nl)←⎕NGET i⊃files
          tbl⍪←CountLetters txt
      :Else
          exit←LogError 'UNABLE_TO_READ_SOURCE'
      :EndTrap
  :Until (≢files)<i←i+1
  :If exit=EXIT.OK
      lines←{⍺,',',⍕⍵}/⊃{⍺(+/⍵)}⌸/↓[1]tbl
      :Trap 0
          bytes←(lines ec nl)⎕NPUT tgt C.NPUT.OVERWRITE
      :Else
          exit←LogError'UNABLE_TO_WRITE_TARGET'
          bytes←0
      :EndTrap
      Log.Log(⍕bytes),' bytes written to ',tgt
  :Endif
∇
```

In this context the `:Trap` structure has an advantage over `⎕TRAP`. When it fires, and control advances to its `:Else` fork, the trap is immediately cleared. So there is no need explicitly to reset the trap to avoid an open loop.

The handling of error codes and messages can easily obscure the rest of the logic. Clarity is not always easy to find, but is well worth working for. This is particularly true where there is no convenient test for an error, only a trap for when it is encountered.

In such cases, it is tempting to use a `:Return` statement to abort the function. But it can be confusing when a function 'aborts' in the middle, and we have learned a great respect for our capacity to get confused. Aborting from the middle of a function may also skip essential tidying up at the end.

We meet this issue reading files, where we trap errors and abort within a loop. Note how the use of while/until allows – unlike a for loop – to test at the ends of the loop both the counter and the exit code.

Rather than simply reporting an error in the file operation, you might prefer to delay a fraction of a second, then retry, perhaps two or three times, in case the problem is ephemeral and clears quickly.

This is in fact a deep topic. Many real-world problems can be treated by fix-and-resume when running under supervision, ie with a UI. Out of disk space? Offer the user a chance to delete some files and resume. But at this point we're working 'headless' – without a UI – and the simplest and lightest form of resilience will serve for now.

We'll provide this in the form of a `retry` operator. This will catch errors in its operand (monadic or dyadic) and retry up to twice at 500-msec intervals.

```
retry←{
    ⍺←⊣
    0::⍺ ⍺⍺ ⍵⊣⎕DL .5
    0::⍺ ⍺⍺ ⍵⊣⎕DL .5
    ⍺ ⍺⍺ ⍵
}
```

The `⍺⍺` in `retry` marks it as an operator, modifying how a function works. `⍺⍺` refers to the function. Assigning `⊣` as the default value of `⍺` makes the operator *ambivalent*: it can modify dyadic functions as well as monadic functions. The error guard `0::` means *in the event of any error*. We use `retry` to modify the file reads and writes in `CountLettersIn`:

```
(txt enc nl)←⎕NGET retry i⊃files
...
bytes←(lines enc nl)⎕NPUT retry tgt C.NPUT.OVERWRITE
```

The `retry` operator goes into `#.MyApp`, not `#.Utilities`, because its strategy of two-more-tries is specific to this application.

Unforeseen errors

Our code so far covers the errors we foresee: errors in the parameters, and errors encountered in the file system. There remain the unforeseen errors, chief among them errors in our own code. If the code we have so far breaks, the EXE will try to report the problem to the session, find no session, and abort with an exit code of 4 to tell Windows "Sorry, it didn't work out."

If the error is easily replicable, we can easily track it down using the development interpreter. But the error might not be easily replicable. It could, for instance, have been produced by ephemeral congestion on a network interfering with file operations. Or the parameters for your app might be so complicated that it is hard to replicate the environment and data with confidence. What you really want for analysing the crash is a crash workspace, a snapshot of the ship before it went down.

For this we need a high-level trap to catch any event not trapped by `CountLettersIn`. We want it to save the workspace for analysis. We might also want it to report the incident to the developer – users don't always do this! For this we'll use the `HandleError` class from the APLTree.

Edit `Z:\code\v03\MyApp.dyapp`:

```
Target #
Load ..\AplTree\APLTreeUtils
Load ..\AplTree\FilesAndDirs
Load ..\AplTree\HandleError
Load ..\AplTree\Logger
Load Constants
Load Utilities
Load MyApp
Run MyApp.SetLX
```

And set an alias `H` for it in the preamble of the `MyApp` namespace:

```
    (A F H L)←#.(APLTreeUtils FilesAndDirs HandleError Logger) ⍝ from APLTree
```

Define a new exit code constant:

```
    OK←0
    APPLICATION_CRASHED←100
    INVALID_SOURCE←101
```

100? Why not 4, the standard Windows code for a crashed application? The distinction is useful. An exit code of 100 will tell us MyApp's trap caught and reported the crash. An exit code of 4 tells you even the trap failed!

We want the high-level trap only when we're running headless, so we'll start as soon as `StartFromCmdLine` begins, setting `HandleError` to do whatever it can before we can give it more specific information derived from the calling environment.

```
∇ StartFromCmdLine;exit;args
 ⍝ Read command parameters, run the application
  ⍝ trap unforeseen problems
  ⎕TRAP←0 'E' '#.HandleError.Process '''''
  args←⌷2 ⎕NQ'.' 'GetCommandLineArgs'
  Off TxtToCsv 2⊃2↑args
∇
```

This trap will do to get things started and catch anything that falls over immediately. We need to get more specific now in `TxtToCsv`. Before getting to work with `CheckAgenda`, refine the global trap definition:

```
    ...
  LogError←Log∘{code←EXIT⍎⍵ ⋄ code⊣⍺.LogError code ⍵}

  isDev←#.A.IsDevelopment
  ⍝ refine trap definition
  #.ErrorParms←H.CreateParms
  #.ErrorParms.errorFolder←F.PWD
  #.ErrorParms.returnCode←EXIT.APPLICATION_CRASHED
  #.ErrorParms.(logFunctionParent logFunction)←Log'Log'
  #.ErrorParms.trapInternalErrors←~isDev
  :If isDev
      ⎕TRAP←0⍴⎕TRAP
  :Else
      ⎕TRAP←0 'E' '#.HandleError.Process ''#.ErrorParms'''
  :EndIf

  :If EXIT.OK=⊃(exit files)←CheckAgenda fullfilepath
    ...
```

`CheckAgenda` and `CountLettersIn` are as before.

The test for `isDev` determines whether (true) the application is allowed to suspend, or (false) errors are to be caught and handled. Later we will set this in an INI file. For now we set it by testing whether the interpreter is development or runtime.

`#.ErrorParms.errorFolder` – write crash files as siblings of the EXE.

`#.ErrorParms.(logFunctionParent logFunction)` – we set a ref to the `Logger` instance, so `HandleError` can write on the log.

Test the global trap

We can test this! Put a full stop[7] in the first few lines of `CountLettersIn`.

```
      tbl←0 2⍴'A' 0
      exit←EXIT.OK ⋄ i←1
      . ⍝ DEBUG
```

This will definitely break. It is not caught by any of the other traps.

Save the change. In the session, run

```
`#.MyApp.TxtToCsv 'Z:\texts\en'`
```

See that `CountLettersIn` suspends on the new line. This is just as you want it. The global trap does not interfere with your development work.

Now export the workspace to `Z:\code\v03\myapp.exe` and run it from the DOS command shell in `Z:\code\v03`:

```
Z:\code\v03>myapp.exe Z:\texts\en

Z:\code\v03>echo Exit Code was %errorcode%
Exit Code was 100
```

and what do we get?

Predictably we get no new result CSV. In `Z:\code\v03\Logs`, we find in the LOG a record of what happened.

First, the log entry records the crash then breaks off:

[7]The English poets among us love that the tersest way to bring a function to a full stop is to type one. (American poets will of course have typed a period and will think of it as calling time out.)

```
2016-05-13 10:56:28 *** Log File opened
2016-05-13 10:56:28 (0) Started MyApp in Z:\code\v03
2016-05-13 10:56:28 (0) Source: Z:\texts\en
2016-05-13 10:56:28 (0) Target: Z:\texts\en.CSV
2016-05-13 10:56:28 (0) *** Error
2016-05-13 10:56:28 (0) Error number=2
2016-05-13 10:56:28 (0) SYNTAX ERROR
2016-05-13 10:56:28 (0) CountLettersIn[4] . ⍝ DEBUG
2016-05-13 10:56:28 (0)                  ^
```

We also have an HTM with a crash report, an eponymous DWS containing the workspace saved at the time it broke, and an eponymous DCF whose single component is a namespace of all the variables defined in the workspace. Some of this has got to help.

However, the crash files names are simply the timestamp prefixed by an underscore:

```
Z:/code/v03/_20160513105628.dcf
Z:/code/v03/_20160513105628.dws
Z:/code/v03/_20160513105628.html
```

We can improve this by setting the Workspace Identification at launch time.

```
    ∇ StartFromCmdLine;exit;args
     ⍝ Read command parameters, run the application
      ⍝ trap unforeseen problems
      ⎕TRAP←0 'E' '#.HandleError.Process '''''
      ⎕WSID←'MyApp'
      args←⌷2 ⎕NQ'.' 'GetCommandLineArgs'
      exit←TxtToCsv 2⊃2↑args
      ⎕OFF exit
    ∇
```

Export the EXE again and run

```
Z:\code\v03>MyApp.exe Z:\texts\en
```

Now your crash files will have better names:

```
Z:/code/v03/MyApp_20160513112024.dcf
Z:/code/v03/MyApp_20160513112024.dws
Z:/code/v03/MyApp_20160513112024.html
```

Remove the deliberate error from `#.MyApp`, save your work and re-export the EXE.

Crash files

What's *in* those crash files?

The HTML contains a report of the crash and some key system variables:

```
MyApp_20160513112024

Version:            Windows 15.0.27377.0 W Runtime
⎕WSID:
⎕IO:                1
⎕ML:                1
⎕WA:                16408308
⎕TNUMS:             0
Category:
EM:                 SYNTAX ERROR
HelpURL:
EN:                 2
ENX:                0
InternalLocation:   parse.c 1722
Message:
OSError:            0 0
Current Dir:        Z:\code\v03
Stack:

#.HandleError.Process[17]
#.MyApp.CountLettersIn[4]
#.MyApp.TxtToCsv[30]
#.MyApp.StartFromCmdLine[5]
Error Message:

SYNTAX ERROR
CountLettersIn[4] . ⍝ DEBUG
                 ^
```

More information is saved in a single component – a namespace – on the DCF.

```
      'Z:/code/v03/MyApp_20160513112024.dcf' ⎕FTIE 1
      ⎕FSIZE 1
1 2 7300 1.844674407E19
      q←⎕FREAD 1 1
      q.⎕NL ⍳10
AN
Category
CurrentDir
DM
EM
EN
ENX
HelpURL
InternalLocation
LC
Message
OSError
TID
TNUMS
Trap
Vars
WA
WSID
XSI


      q.Vars.⎕NL 2
ACCENTS
args
exit
files
fullfilepath
i
isDev
tbl
tgt
```

The DWS is the crash workspace. Load it. The Latent Expression has been disabled, to ensure MyApp does not attempt to start up again.

```
      ⎕LX
⎕TRAP←0 'S' ⍝#.MyApp.StartFromCmdLine
```

The State Indicator shows the workspace captured at the moment the HandleError object saved the workspace. Your problem – the full stop in `MyApp.CountLettersIn` – is two levels down in the stack.

```
      )SI
#.HandleError.SaveErrorWorkspace[6]*
#.HandleError.Process[23]
#.MyApp.CountLettersIn[4]*
#.MyApp.TxtToCsv[30]
#.MyApp.StartFromCmdLine[5]
```

You can clear `HandleError` off the stack with a naked branch arrow. When you do so, you'll find the original global trap restored. Disable it. Otherwise any error you produce while investigating will trigger `HandleError` again!

```
      →
      )SI
#.MyApp.CountLettersIn[4]*
#.MyApp.TxtToCsv[30]
#.MyApp.StartFromCmdLine[5]
      ⎕TRAP
  0 E #.HandleError.Process '#.ErrorParms'
      ⎕←⎕TRAP←0/⎕TRAP
```

In development you'll discover and fix most errors while working from the APL session. Unforeseen errors encountered by the EXE will be much rarer. Now you're all set to investigate them!

Configuration settings

Right now our MyApp program counts only Latin letters, ignores case and maps accented characters to their unaccented correspondents. That makes it of less use for French and German texts and of no use at all for Greek or Japanese texts. Other alphabets are possible. They could be supplied with the EXE as alphabet DATs and selected with command-line options. The user could supplement them with her own alphabet DATs. The default alphabet could be any one of the alphabet DATs. Or we might store all the alphabets as a single XML document.

Thinking more widely, an application's configuration includes all kinds of state: e.g., window positions, recent filepaths, and GUI themes.

In the chapter on Logging, we considered the question of where to keep application logs. The answer depends in part on what kind of application you are writing. Will there be single or multiple instances? For example, while a web browser might have several windows open simultaneously, it is nonetheless a single instance of the application. Its user wants to run just one version of it, and for it to remember her latest preferences and browsing history. But a machine may have many users, and each user needs her own preferences and history remembered.

Our MyApp program might well form part of other software processes, perhaps running as a service. There might be multiple instances of MyApp running at any time, quite independently of each other, each with quite different configuration settings.

Where does that leave us? We want configuration settings:

As defaults for the application in the absence of any other configuration settings
: These must be coded into the application, so it will run in the absence of any configuration files. But an administrator should be able to revise these settings for a site. So they should be saved somewhere for all users. This filepath is represented in Windows by the `ALLUSERSPROFILE` environment variable. So we might look there for a `MyApp\MyApp.ini` file.

As part of the user's profile
: The Windows environment variable `USERPROFILE` points to the individual user's profile, so we might look there for a `MyApp\MyApp.ini` file.

For invocation when the application is launched.
: We can look in the command line arguments for an INI.

From the above we get a general pattern for configuration settings:

1. Set from program code
2. Overwrite from ALLUSERSPROFILE if any
3. If INI in command line, overwrite from it; else overwrite from USERPROFILE

Using the Windows Registry

The Windows Registry is held in memory, so it is fast to read. It has been widely used to store configuration settings. Many would say *abused*. We follow a consensus opinion that it is well to minimise use of the Registry.

Settings needed by Windows itself *have* to be stored in the Registry. For example, associating a file extension with your application, so that double clicking on its icon launches your application.

The APLTree class WinReg[8] provides methods for handling the Windows Registry.

MyApp doesn't need the Windows Registry at this point. We'll store its configurations in configuration files.

INI, JSON, or XML configuration files?

Three formats are popular for configuration files: INI, JSON and XML. INI is the oldest, simplest, and most crude. The other formats offer advantages: XML can represent nested data structures, and JSON can do so with less verbosity. Both XML and JSON depend upon unforgiving syntax: a single typo in an XML document can render it impossible to parse.

We want configuration files to be suitable for humans to read and write, so you might consider the robustness of the INI format an advantage. Or a disadvantage: a badly-formed XML document is easy to detect, and a clear indication of an error.

Generally, we prefer simplicity and recommend the INI format where it will serve.

Parameters for MyApp

We'll introduce some choices for MyApp that can then be set by configuration parameters.

`ACCENTED`
: A flag to control whether accented characters are distinguished, or mapped to their unaccented forms. By default, this will be off.

`ALPHABETS`
: We'll furnish MyApp with a repertoire of named alphabets, and allow more alphabets to be defined in configuration files.

`ALPHABET`
: By default, MyApp will use the English alphabet. But we'll allow another default language to be configured.

[8] http://aplwiki.com/WinReg

`OUTPUT`
: Output has so far been to a CSV eponymous with and sibling to the source. Now that can be specified.

`SOURCE`
: So far, the source files have been specified either in the APL session, or in the Windows command line. But it might be convenient for a program calling `MyApp.exe` to specify everything in an INI file, so we'll make the source a configurable parameter.

Configuration *à la mode*

Distinguish two modes in which we run the MyApp code:

Application mode
: The exported EXE is run from the Windows command line.

Session mode
: The application has been assembled by the DYAPP and is being run in the development interpreter.

The mode will determine where we look for configuration parameters.

Sources of configuration parameters

Program code

We want MyApp to run in the absence of any external configuration parameters. So the program code must provide default values for all parameters.

User profiles

Windows provides filepaths for profiles for both individual users and all users.

In Application mode, the All Users profile is consulted for defaults.

In Session mode, the User profile is consulted for defaults.

Command line

In Application mode, any parameter specified in the command line supersedes other values for that parameter.

```
Z:\code\v04\>MyApp.exe ALPHABET=French ACCENTED=No Z:\texts\fr
```

In Session mode, we ignore parameters set on the command line, assuming that they relate to the Dyalog development environment.

We might also call `MyApp.exe` specifying only an INI of configuration parameters:

```
Z:\code\v04>type M:\jobqueue\job008.ini
ALPHABET=French
ACCENTED=No
SOURCE=M:\texts\Rimbaud.txt
Z:\code\v04>MyApp.exe J:\queue\job008.ini OUTPUT=J:\results\out008.csv
```

Or we might use INIs as job 'profiles', for example:

```
Z:\code\v04>type M:\profiles\p06.ini
ALPHABET=French
ACCENTED=No
Z:\code\v04>MyApp.exe J:\profiles\p06.ini SOURCE=M:\texts\Rimbaud.txt OUTPUT=J:\\
results\out008.csv
```

Manifest

Here's our manifest for Version 4: `MyApp.dyapp`:

```
Target #
Load ..\AplTree\APLTreeUtils
Load ..\AplTree\FilesAndDirs
Load ..\AplTree\HandleError
Load ..\AplTree\IniFiles
Load ..\AplTree\Logger
Load Constants
Load Utilities
Load MyApp
Run MyApp.Start 'Session'
```

We've included IniFiles to the DYAPP's build list. We'll use IniFiles to handle INI files.

ADOC is useful for reading a class's documentation. The APLTree library scripts all have ADOC documentation. You can read it in a browser:

```
]adoc_browse #.IniFiles
```

The DYAPP's `Run` command now calls `MyApp.Start`, specifying the mode.

Configuration parameters in program code

We'll start by implementing these parameters in the program code, independently of any external configuration files. Then continue to look for other sources of configuration parameters. A new function `GetParameters` will do that when the application starts, and put the results – all the configuration parameters – in `#.MyApp.Params`.

The INIs can define new alphabets, so we'll put a namsepace `ALPHABETS` inside `#.MyApp.Params`.

```
    ∇ p←GetParameters mode;args;fromexe;fromallusers;fromcmdline;fromuser;env;pa\
ths;ini;alp
      ⍝ Derive parameters from defaults and command-line args (if any)

      ⍝ Application defaults: in the absence of any other values
       (p←⎕NS'').(accented alphabet source)←0 'English' '' ⍝ defaults
       p.ALPHABETS←⎕NS'' ⍝ container for new alphabet definitions
       p.ALPHABETS.English←⎕A
       p.ALPHABETS.French←'AÁÂÀBCÇDEÈÊÉFGHIÌÍÎJKLMNOÒÓÔPQRSTUÙÚÛVWXYZ'
       p.ALPHABETS.German←'AÄBCDEFGHIJKLMNOÖPQRSßTUÜVWXYZ'
       p.ALPHABETS.Greek←'ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ'
```

We've arbitrarily supposed here that diacritical marks in classical Greek are guides to pronunciation and are not to be counted as accented characters.

And we extend our map of accented characters, remembering different characters can look the same:

```
    ∆←'ÁÂÃÀÄÅÇĐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝάΆέΈήΉίϊΐΊόΌύϋΎώΏ'
    ACCENTS←↑∆ 'AAAAAACDEEEEIIIINOOOOOOUUUUYAAEEHHIIIIOOYYYΩΩ'
```

```
      =/⊃¨p.ALPHABETS.(English French) ⍝ A is A
1
      =/⊃¨p.ALPHABETS.(English Greek)  ⍝ A is not Alpha
0
```

Setting values in the program code ensures `MyApp.exe` will be able to run in the absence of any external INIs.

Now we see what is on the command line that called it, and look around for other INIs.

```
        args←⌷2 ⎕NQ'.' 'GetCommandLineArgs'   ⍝ Command Line
        env←U.GetEnv                          ⍝ Windows Environment


       ⍝ An INI for this app as a sibling of the EXE
        fromexe←(⊃⎕NPARTS⊃args),⎕WSID,'.INI' ⍝ first arg is source of EXE
       ⍝ First INI on the command line, if any
        fromcmdline←{×≢⍵:⊃⍵ ⋄ ''}{⍵/⍨'.INI'∘≡¨¯4↑¨⍵}(1↓args)
       ⍝ An INI for this app in the ALLUSERS profile
        fromallusers←env.ALLUSERSPROFILE,'\',⎕WSID,'.INI'
       ⍝ An INI for this app in the USER profile
        fromuser←env.USERPROFILE,'\',⎕WSID,'.INI'
```

That identifies four possible INIs we might read. Which ones we consult depends on the mode:

```
        :Select mode
        :Case 'Application'
            paths←fromexe fromallusers fromcmdline
        :Case 'Session'
            paths←fromexe fromallusers fromuser
        :EndSelect
```

Case and configuration parameters

It looks like we're almost there with configuration parameters. We can create an instance of IniFiles. Its `Convert` method will take a list of INIs, and return a namespace of parameters, with conflicts between the INIs resolved.

But case gets in the way.

Windows is case-insensitive, so `ALPHABET=FRENCH`, `alphabet=french`, and `Alphabet=French` should be equivalent. Any of them has to be interpreted to set the APL variable `alphabet`: in APL, case matters. So we'll loop through the paths.

We have already defined some alphabets `English`, `French` etc, so we'll set a convention that alphabet names are in title case.

```
:For path :In {⍵/⍨⎕NEXISTS¨⍵}{⍵/⍨×≢¨U.trim¨⍵}paths
   ⍝ Allow INI entries to be case-insensitive
    ini←⎕NEW #.IniFiles(,⊂path)
    vars←U.m2n ini.⎕NL 2
    :For parm :In {⍵/⍨ini.Exist¨'Config:'∘,¨⍵}PARAMS
       ⍝ Alphabet names are title case, eg Greek
        Δ←⊃ini.Get'Config:',parm
        parm p.{⍎⍺,'←⍵'}U.toTitlecase⍣(parm≡'alphabet') Δ
    :EndFor

    :If ini.Exist'Alphabets:'
        Δ←(ini.Convert ⎕NS'') ⍝ breaks if keys are not valid APL names
        a←Δ.⍎'ALPHABETS'U.ciFindin U.m2n Δ.⎕NL 9
       ⍝ Alphabet names are title case, eg Russian
        Δ←,' ',a.⎕NL 2 ⍝ alphabet names
        (U.toTitlecase Δ)p.ALPHABETS.{⍎⍺,'←⍵'}a⍎Δ
    :EndIf
:EndFor
```

For this we've put new functions into `#.Utilities`: `m2n` *matrix to nest*, `ciFind` *case-independent find*, and `toTitlecase`. You can also observe the use of the *power* operator `⍣` instead of an if/else control structure.

Finally, in Application mode, we check the command line for any parameters set directly:

```
    :If mode≡'Application' ⍝ set params from the command line
    :AndIf ×≢a←{⍵/⍨'='∊¨⍵}args
        Δ←a⍳¨'=' ⋄ (k v)←((Δ-1)↑¨a)((Δ+1)↓¨a)
        Δ←(≢PARAMS)≥i←⊃⍳/U.toUppercase¨¨PARAMS k
        (⍕PARAMS[Δ/i]) p.{⍎⍺,'←⍵'} Δ/v
    :EndIf
```

Checking the agenda

Now that configuration parameters can be specified in INIs, there is more to check. So we pass the entire namespace of parameters to `CheckAgenda` as a left argument. `CheckAgenda` deploys a new error code – for an invalid alphabet name - and extends its result also to return the collation alphabet, with or without accented characters.

```
    ∇ (exit files alphabet)←params CheckAgenda ffp;fullfilepath;type
      (files alphabet)←'' '' ⍝ error defaults
      fullfilepath←F.NormalizePath ffp
      type←C.NINFO.TYPE ⎕NINFO fullfilepath
      :If 0=≢fullfilepath~' '
      :OrIf ~⎕NEXISTS fullfilepath
          exit←LogError'SOURCE_NOT_FOUND'
      :ElseIf ~type∊C.NINFO.TYPES.(DIRECTORY FILE)
          exit←LogError'INVALID_SOURCE'
      :ElseIf 2≠params.(ALPHABETS.⎕NC alphabet)
          exit←LogError'INVALID_ALPHABET_NAME'
      :Else
          exit←EXIT.OK
          :Select type
          :Case C.NINFO.TYPES.DIRECTORY
              files←⊃(⎕NINFO⍠'Wildcard' 1)fullfilepath,'\*.txt'
          :Case C.NINFO.TYPES.FILE
              files←,⊂fullfilepath
          :EndSelect
          alphabet←params.{(ALPHABETS⍎alphabet)~(~accented)/⍵}ACCENTS[1;]
      :EndIf
    ∇
```

Initialising the workspace

Checking the agenda requires more or less the same work whether we are starting in Session or in Application mode. That’s why the DYAPP (see above) now finishes with:

```
Run MyApp.Start 'Session'
```

The validated configuration parameters will be set in a namespace `MyApp.Params`:

```
    ∇ Start mode;exit
    ⍝ Initialise workspace for session or application
    ⍝ mode: ['Application' | 'Session']
      :If mode≡'Application'
          ⍝ trap problems in startup
          ⎕TRAP←0 'E' '#.HandleError.Process '''''
      :EndIf
```

```
  ⎕WSID←'MyApp'
  Params←GetParameters mode
  :Select mode
  :Case 'Session'
      ⎕←'Alphabet is ',Params.alphabet
      ⎕←'Defined alphabets: ',⍕U.m2n Params.ALPHABETS.⎕NL 2
      #.⎕LX←'#.MyApp.Start ''Application''' ⍝ ready to export
  :Case 'Application'
      exit←TxtToCsv Params.source
      Off exit
  :EndSelect
∇
```

What we think about when we think about encapsulating state

The configuration parameters are set in `Start` but the `Params` namespace is not passed explicitly to `TxtToCsv`. Perhaps `TxtToCsv` just refers to `Params`?

```
∇ exit←TxtToCsv fullfilepath;∆;Log;LogError;files;tgt;alpha

...

  :If EXIT.OK=⊃(exit files alpha)←Params CheckAgenda fullfilepath
      Log.Log'Target: ',tgt←(⊃,/2↑⎕NPARTS fullfilepath),'.CSV'
      exit←alpha CountLettersIn files tgt
  :EndIf
  Log.Log'All done'
∇
```

Yes, that's it. Bit of a compromise here. Let's pause to look at some other ways to write this:

- Let `TxtToCsv` ignore `Params`. It doesn't read or set the contents. `CheckAgenda` can read `Params` instead. But, where we can, we avoid passing information through globals and 'semiglobals'. The exact opposite practice is to pass everything through function arguments. There is no appreciable performance penalty for this. The interpreter doesn't make 'deep copies' of the arguments unless and until they are modified in the called function (which we hardly ever do) – instead the interpreter just passes around references to the original variables.
- So we could pass `Params` as a left argument of `TxtToCsv`, which then simply gets passed to `CheckAgenda`. No performance penalty for this, as just explained, but now we've loaded the

syntax of `TxtToCsv` with a namespace it makes no direct use of, an unnecessary complication of the writing. And we've set a left argument we (mostly) won't want to specify when working in Session mode. We could make this left argument optional, taking `#.MyApps.Params` as its default. The cost of that is an if/else control statement, setting a value that, er, `TxtToCsv` still isn't reading or setting for itself. (And if we did want to set a left argument for `TxtToCsv` to use in Session mode, it would probably be the name of an alphabet…

The matter of *encapsulating state* – which functions have access to state information, and how it is shared between them – is very important. Poor choices can lead to tangled and obscure code.

From time to time you will be offered (not by us) rules that attempt to make the choices simple. For example: *never communicate through globals*. (Or semi-globals.[9]) There is some wisdom in these rules, but they masquerade as satisfactory substitutes for thought, which they are not. Just as in a natural language, any rule about writing style meets occasions when it can and should be broken. Following style 'rules' without considering the alternatives will from time to time have horrible results, such as functions that accept complex arguments only to pass them on unexamined to other functions.

Think about the value of style 'rules' and learn when to apply them.

Sometimes it's only after writing many lines of code that it becomes apparent that a different choice would have been better. And sometimes it becomes apparent that the other choice would be so much better that it's worth unwinding and rewriting a good deal of what you've done. (Then be glad you're writing in a terse language.)

We share these musings here so you can see what we think about when we think about encapsulating state; and also that there is often no clear right answer. Think hard, make your best choices, and be ready to unwind and remake them later if necessary.

Accents

We'll retain our map of accented to unaccented characters and convert any accented character *not in the alphabet* to its unaccented equivalent.

However, we'll suppose that the diacritical marks in classical Greek are not to distinguish different characters, so we'll extend the accents map to Greek:

```
∆←'ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝάΆέΈήΉίϊΐΊόΌύϋΎώΏ'
ACCENTS←↑∆ 'AAAAAACDEEEEIIIINOOOOOOUUUUYAAEEHHIIIIOOYYYΩΩ'
```

Note that for Greek this will map Á to the Greek Alpha, not the visually indistinguishable Roman A.

[9]So-called *semi-globals* are variables to be read or set by functions to which they are not localised. They are *semi-globals* rather than globals because they are local to either a function or a namespace. From the point of view of the functions that do read or set them, they are indistinguishable from globals – they are just mysteriously 'around'.

```
      '.*'[1+#.MyApp.(ACCENTS∊Params.ALPHABETS.Greek)]
............................................
...........................*****************
```

Where and how shall we set the alphabet to be used? We start by revising `CountLetters` to take an alphabet as left argument. And now that the alphabet is explicit, it makes sense to sort the result into alphabetical order.

```
      CountLetters←{
          accents←↓ACCENTS/⍨~ACCENTS[2;]∊⍺ ⍝ ignore accented chars in alphabet ⍺
          ⍺{⍵[⍺⍋⍵[;1];]}{⍺(≢⍵)}⌸⍺{⍵/⍨~⍵∊⍺}accents U.map U.toUppercase ⍵
      }
```

Reality check. From Project Gutenberg[10] save the UTF-8 text files for Homer's *Iliad* and *Odyssey* in Greek. Top and tail them in a text editor to remove the English-language header and licence. That will still leave a few stray Roman characters, and lots of line numbers:

```
      )CS #.MyApp
#.MyApp
      ≢il←⊃⎕NGET 'Z:\texts\gr\iliad.txt'
679693
      ⎕A ⎕D CountLetters¨⊂il
 B 1  0 1459
 C 1  1  766
 D 1  2  791
 E 4  3  775
 H 1  4  727
 I 1  5 1749
 K 1  6  455
 L 5  7  444
 O 6  8  359
 P 1  9  316
 R 2
 S 2
 T 2
 V 1
```

But most of it is in the Greek alphabet:

[10] http://www.projectgutenberg.org/

```
      Params.ALPHABETS.Greek CountLetters il
Α 50459
Β  4533
Γ  8955
Δ  7535
Ε 29370
Ζ  2132
Η 10347
Θ  5346
Ι 34425
Κ 20629
Λ 13520
Μ 16659
Ν 28182
Ξ  2655
Ο 41993
Π 17518
Ρ 21353
Σ 38822
Τ 41108
Υ 14682
Φ  6165
Χ  7085
Ψ   794
Ω  5877
```

Putting it together

For our first ‘sea trial’ we’ll use an INI file to define the Russian alphabet, and specify it to count the letter frequency in a poem by Pushkin[11].

One INI should define everything we need:

```
[CONFIG]
ALPHABET=RUSSIAN
SOURCE=Z:\TEXTS\PUSHKIN.TXT
[ALPHABETS]
RUSSIAN=АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
```

Happily, Wikipedia assures us we have no accented characters to handle.

We’ll save the INI in the All-users Profile:

[11] http://www.gutenberg.org/ebooks/5316

```
      ⎕CMD 'echo %allusersprofile%'
C:\ProgramData
```

i.e. in `C:\ProgramData\MyApp.ini` and test in Session mode.

```
clear ws
Booting Z:\code\v04\MyApp.dyapp
Loaded: #.APLTreeUtils
Loaded: #.FilesAndDirs
Loaded: #.HandleError
Loaded: #.IniFiles
Loaded: #.Logger
Loaded: #.Constants
Loaded: #.Utilities
Loaded: #.MyApp
Alphabet is Russian
Defined alphabets:  English  French  German  Greek  Russian
      MyApp.TxtToCsv 'Z:\texts\ru'
0
```

And the result? In `Z:\texts\ru.csv`:

```
А,109
Б,22
В,57
Г,19
Д,32
Е,101
Ж,13
З,16
И,61
Й,29
К,48
Л,45
М,30
Н,75
О,99
П,34
Р,58
С,76
Т,87
У,42
```

```
Ф,2
Х,14
Ц,4
Ч,10
Ш,12
Щ,6
Ы,31
Ь,18
Э,2
Ю,7
Я,26
```

Looks like a win.

Testing: the sound of breaking glass

Our application here is simple – just count letter frequency in text files.

All the other code has been written to package the application for shipment, and to control how it behaves when it encounters problems.

Developing code refines and extends it. We have more developing to do. Some of that developing might break what we already have working. Too bad. No one's perfect. But we would at least like to know when we've broken something – to hear the sound of breaking glass behind us. Then we can fix the error before going any further.

In our ideal world we would have a team of testers continually testing and retesting our latest build to see if it still does what it's supposed to do. The testers would tell us if we broke anything. In the real world we have programs – tests – to do that.

What should we write tests for? "Anything you think might break," says Kent Beck[12], author of *Extreme Programming Explained*. We've already written code to allow for ways in which the file system might misbehave. We should write tests to discover if that code works. We'll eventually discover conditions we haven't foreseen and write fixes for them. Then those conditions too join the things we think might break, and get added to the test suite.

Why you want to write tests

Notice when you break things

Some functions are more vulnerable than others to being broken under maintenance. Many functions are written to encapsulate complexity, bringing a common order to a range of different arguments. For example, you might write a function that takes as argument any of a string[13], a vector of strings, a character matrix or a matrix of strings. If you later come to define another case, say, a string with embedded line breaks, it's easy enough inadvertently to change the function's behaviour with the original cases.

If you have tests that check the function's results with the original cases, it's easy to ensure your changes don't change the results unintentionally.

[12]Kent Beck, in conversation with one of the authors.

[13]APL has no *string* datatype. We use the word as a casual synonym for *character vector*.

More reliable than documentation

No, tests don't replace documentation. They don't convey your intent in writing a class or function. They don't record your ideas for how it should and should not be used, references you consulted before writing it, or thoughts about how it might later be improved.

But they do document with crystal clarity what it is *known* to do. In a naughty world in which documentation is rarely complete and even less often revised when the code is altered, it has been said the *only* thing we know with certainty about any given piece of software is what tests it passes.

Understand more

Test-Driven Design (TDD) is a high-discipline practice associated with Extreme Programming. TDD tells you to write the tests *before* you write the code. Like all such rules, we recommend following TDD thoughtfully. The reward from writing an automated test is not *always* worth the effort. But it is a very good practice and we strongly recommend it.

If you are writing the first version of a function, writing the tests first will clarify your understanding of what the code should be doing. It will also encourage you to consider boundary cases or edge conditions: for example, how should the function above handle an empty string? A character scalar? TDD first tests your understanding of your task. If you can't define tests for your new function, perhaps you're not ready to write the function either.

If you are modifying an existing function, write new tests for the new things it is to do. Run the revised tests and see that the code fails the new tests. If the unchanged code *passes* any of the new tests… review your understanding of what you're trying to do!

Write better

Writing functions with a view to passing formal tests will encourage you to write in *functional style*. In pure functional style, a function reads only the information in its arguments and writes only its result. No side effects or references.

```
  ∇ Z←mean R;r
[1] Z←((+/r)÷≢r←,R)
  ∇
```

In contrast, this line from `TxtToCsv` reads a value from a namespace external to the function (`EXIT.APPLICATION_CRASHED`) and sets another: `#.ErrorParms.returnCode`.

```
  #.ErrorParms.returnCode←EXIT.APPLICATION_CRASHED
```

In principle, `TxtToCsv` *could* be written in purely functional style. References to classes and namespaces `#.HandleError`, `#.APLTreeUtils`, `#.FilesAndDirs`, `EXIT`, and `#.ErrorParms` could all be passed to it as arguments. If those references ever varied – for example, if there were an alternative namespace `ReturnCodes` sometimes used instead of `EXIT` – that might be a useful way to write `TxtToCsv`. But as things are, cluttering up the function's *signature* – its name and arguments – with these references harms rather than helps readability. It is an example of the cure being worse than disease.

You can't write *everything* in pure functional style but the closer you stick to it, the better your code will be, and the easier to test. Functional style goes hand in hand with good abstractions, and ease of testing.

Why you don't want to write tests

There is nothing magical about tests. Tests are just more code. The test code needs maintaining like everything else. If you refactor a portion of your application code, the associated tests need reviewing – and possibly revising – as well. In programming, the number of bugs is generally a linear function of code volume. Test code is no exception to this rule. Your tests are both an aid to development and a burden on it.

You want tests for everything you think might break, but no more tests than you need.

Beck's dictum – test anything you think might break – provides useful insight. Some expressions are simple enough not to need testing. If you need the indexes of a vector of flags, you can *see* that `{⍵/⍳≢⍵}` will find them. It's as plain as `2+2` making four. You don't need to test that. APL's scalar extension and operators such as *outer product* allow you to replace nested loops (a common source of error) with expressions which don't need tests. The higher level of abstraction enabled by working with collections allows not only fewer code lines but also fewer tests.

Time for a new version of MyApp. Make a copy of `Z:\code\v04` as `Z:\code\v05`.

Setting up the test environment

We'll need the Tester class from the APLTree library. And a namespace of tests, which we'll dub `#.Tests`.

Write `Z:\code\v05\Tests.dyalog`:

```
:Namespace Tests
⍝ Dyalog Cookbook, Version 05
⍝ Tests
⍝ Vern: sjt24jun16

:EndNamespace
```

and include both scripts in the DYAPP:

```
Target #
Load ..\AplTree\APLTreeUtils
Load ..\AplTree\FilesAndDir
Load ..\AplTree\HandleError
Load ..\AplTree\IniFiles
Load ..\AplTree\Logger
Load ..\AplTree\Tester
Load Constants
Load Utilities
Load Tests
Load MyApp
Run MyApp.Start 'Session'
```

Run the DYAPP to build the workspace. In the session execute `]adoc_browse #.Tester` to see the documentation for the Tester class, and browse also to aplwiki.com/Tester[14] to see the discussion there.

Unit and functional tests

> Unit tests tell a developer that the code is *doing things right*; functional tests tell a developer that the code is *doing the right things*.

It's a question of perspective. Unit tests are written from the programmer's point of view. Does the function or method return the correct result for given arguments? Functional tests, on the other hand, are written from the user's point of view. Does the software do what its *user* needs it to do?

Both kinds of tests are important. If you are a professional programmer you need a user representative to write functional tests. If you are a domain-expert programmer[15] you can write both.

In this chapter we'll tackle unit tests. Later in the book we'll consider functional tests.

[14]http://aplwiki.com/Tester

[15]an expert in the domain of the application rather than an expert programmer, but who has learned enough programming to write the code.

Writing unit tests

Unit tests without state

Utilities are a good place to start writing tests. Many utility functions are simply names assigned to common expressions. Others encapsulate complexity, making similar transformations of different arguments.

We'll start in `#.Utilities` with the simple case-transforming functions: `toLowercase`, `toUppercase`, and `toTitlecase`. The first two of these merely compose left arguments of `0` and `1` to the experimental I-beam `819⌶`.

Why test that? When could that ever break? Well, an experimental I-beam is just that: experimental. It might be withdrawn. If that ever happened, we'd write new versions of `toLowercase` and `toUppercase`, and we would definitely want tests. But the time to write the tests is now, when we are most clear what we need the functions to do.

One thing we need these functions to do is handle case in bicameral scripts other than Latin, eg Greek and Cyrillic. That is neglected by many case-switching utilities. So we'll test a few Greek characters as well. In `Z:\code\v05\Tests.dyalog`:

```
:Namespace Tests
⍝ Dyalog Cookbook, Version 05
⍝ Tests
⍝ Vern: sjt26jun16

    EN_lower←'abcdefghijklmnopqrstuvwxyz'
    ⍝ accented Latin and Greek characters
    AccentedUpper←'ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝΆΈΉΊÓΌÚΎΏ'
    AccentedLower←'áâãàäåçðèêëéìíîïñòóôõöøùúûüýάέήíóόúύώ'
```

Now some boundary cases

```
∇ Z←Test_toLowercase_001(debugFlag batchFlag)
 ⍝ boundary case
  Z←''≢#.Utilities.toLowercase''
∇

∇ Z←Test_toLowercase_002(debugFlag batchFlag);∆
 ⍝ no case
  Z←∆≢#.Utilities.toLowercase ∆←' .,/'
∇
```

The two-flag right argument is part of the signature of a test function. We'll come back to what the flags do. The result `Z` indicates `¯1`: the test broke ; `0`: no problem found; `1`: problem found.

Now the basic English alphabet, and the Latin and Greek accented characters

```
∇ Z←Test_toLowercase_003(debugFlag batchFlag)
 ⍝ base case
  Z←EN_lower≢#.Utilities.toLowercase ⎕A
∇

∇ Z←Test_toLowercase_004(debugFlag batchFlag)
 ⍝ accented Latin and Greek characters
  Z←AccentedLower≢#.Utilities.toLowercase AccentedUpper
∇
```

This is tedious already, but writing tests is all about being thorough, so we'll replicate these four tests for `toUppercase` (reversing the arguments) and throw in a couple for `toTitlecase` as well.

```
⍝ #.Utilities.toTitlecase

∇ Z←Test_toTitlecase_001(debugFlag batchFlag)
 ⍝ base case
  Z←'The Quick Brown Fox'≢ #.Utilities.toTitlecase'the QUICK brown FOX'
∇

∇ Z←Test_toTitlecase_002(debugFlag batchFlag)
 ⍝ Greek script
  Z←'Όι Πολλοί'≢ #.Utilities.toTitlecase'όι ΠΟΛΛΟΊ'
∇
```

That will do as a start. Notice each test is defined as a function that returns a scalar flag indicating whether it has found an error. (Not whether it has passed.) No test has referred to any argument: we'll come back to that shortly.

Let's give these tests a run.

```
      #.Tester.Run #.Tests
--- Tests started at 2016-06-26 14:08:04  on #.Tests -----------------
  Test_toLowercase_001 (1 of 10) : boundary case
  Test_toLowercase_002 (2 of 10) : no case
  Test_toLowercase_003 (3 of 10) : base case
  Test_toLowercase_004 (4 of 10) : accented Latin and Greek characters
  Test_toTitlecase_001 (5 of 10) : base case
  Test_toTitlecase_002 (6 of 10) : Greek script
  Test_toUppercase_001 (7 of 10) : boundary case
  Test_toUppercase_002 (8 of 10) : no case
  Test_toUppercase_003 (9 of 10) : base case
* Test_toUppercase_004 (10 of 10) : accented Latin and Greek characters
 ----------------------------------------------------------------------
   10 test cases executed
   1 test case failed
   0 test cases broken
```

Ah. Now there's a surprise. Despite their simplicity, we already have a test that failed. Let's investigate.

```
      )CS #.Tests
#.Tests
      AccentedLower ≡ #.Utilities.{toLowercase toUppercase ⍵} AccentedLower
1
      AccentedUpper ≡ #.Utilities.{toUppercase toLowercase ⍵} AccentedUpper
0
```

So the problem is with `AccentedUpper`.

```
      where←{⍵/⍳≢⍵}
      where AccentedUpper≠#.Utilities.{toUppercase toLowercase ⍵} AccentedUpper
33 35
      AccentedUpper[33 35]
όύ
```

And there we have it. `AccentedUpper` includes two lowercase characters. Easy enough to miss if you're not familiar with Greek script. Easy enough to find and fix! Testing rocks. Redefine the accented character lists.

```
    ⍝ accented Latin and Greek characters
    AccentedUpper←'ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝΆΈΉΊΌΎΩ'
    AccentedLower←'áâãàäåçðèêëéìíîïñòóôõöøùúûüýάέήίόύώ'
```

Now our tests all pass. We'll proceed to something more substantial: `CountLetters`. A base case would run it against the English alphabet:

```
      Params.ALPHABETS.English CountLetters 'The Quick Brown Fox'
B 1
C 1
E 1
F 1
H 1
I 1
K 1
N 1
O 2
Q 1
R 1
T 1
U 1
W 1
X 1
```

That result could be specified more conveniently as two columns.

```
      ↓[1]Params.ALPHABETS.English CountLetters 'The Quick Brown Fox'
┌───────────────┬─────────────────────────────┐
│BCEFHIKNOQRTUWX│1 1 1 1 1 1 1 1 2 1 1 1 1 1 1│
└───────────────┴─────────────────────────────┘
```

So our new section in `#.Tests` becomes

```
    ⍝ #.MyApp.CountLetters

∇ Z←Test_CountLetters_001(debugFlag batchFlag);a;r
 ⍝ base case
  a←#.MyApp.Params.ALPHABETS.English
  r←('BCEFHIKNOQRTUWX')(1 1 1 1 1 1 1 1 2 1 1 1 1 1 1)
  Z←r≢↓[1]a #.MyApp.CountLetters'The Quick Brown Fox'
∇
```

Unit tests with state

You might have spotted some 'state' hidden in the last test. `CountLetters` refers to the matrix `ACCENTS` of accented characters. That table is a constant, so perhaps we can be allowed to say the last test is a stateless unit test.

The same cannot be true of the `CountLettersIn` function, which reads and writes files. The result depends on the state in the files.

At this point we need to incorporate files and folders into our test apparatus. We have suitable source folders and output files in `Z:\texts`, but we have a problem with using them as they are: we have no independent way of verifying the results for the contents.

Better to use shorter files for which the correct results can be determined independently. Best of all, let the tests generate the files.

The Tester class looks in the tests for a function called `Initial` and runs that first. We'll use this convention to create test files with known character frequencies.

```
C←#.Constants
TEST_FLDR←'./tests/'
∇ Initial;C;file
  :If ⎕NEXISTS TEST_FLDR
      :For file :In ⊃C.NINFO.NAME(⎕NINFO⍠'Wildcard' 1)TEST_FLDR,'*.*'
          ⎕NDELETE file
      :EndFor
  :Else
      ⎕MKDIR TEST_FLDR
  :EndIf
∇
...
```

Why loop on `file` rather than `⎕NDELETE¨` the list of files? Because `⎕NDELETE¨` will break on an empty list.

```
∇ Z←Test_CountLettersIn_001(debugFlag batchFlag);a;files;cf;cc2n;sas;res
 ⍝ across multiple files
  a←#.MyApp.Params.ALPHABETS.English
  cc2n←{2 1∘⊃¨⎕VFI¨2↓¨⍵}                       ⍝ CSV col 2 as numbers
  res←(¯1↓TEST_FLDR),'.csv'                     ⍝ results file
  sas←{⍵[?⍨≢⍵]}                                 ⍝ scramble a string
  cf←?1000⍴⍨5,≢a                                ⍝ random freqs for 5 files
  files←{TEST_FLDR,'test',⍵,'.txt'}∘⍕¨⍳≢cf
  (sas¨(↓cf)/¨⊂a)⎕NPUT¨files
  :If Z←#.MyApp.EXIT.OK≢a #.MyApp.CountLettersIn files res
  :OrIf Z←(+⌿cf)≢cc2n⊃⎕NGET res 1
  :EndIf
∇
```

Admire how elegantly the notation lets us generate randomised character frequencies from alphabet `a` for 5 files (`?1000⍴⍨5,≢a`) and scramble them (`{⍵[?⍨≢⍵]}`).

But our elegant test breaks!

```
      #.Tester.EstablishHelpersIn #.Tests
      #.Tests.Run
--- Tests started at 2016-07-24 11:18:08  on #.Tests ----------------------
# Test_CountLettersIn_001 (1 of 12) : across multiple files
  Test_CountLetters_001 (2 of 12) : base case
  Test_toLowercase_001 (3 of 12) : boundary case
  Test_toLowercase_002 (4 of 12) : no case
  Test_toLowercase_003 (5 of 12) : base case
  Test_toLowercase_004 (6 of 12) : accented Latin and Greek characters
  Test_toTitlecase_001 (7 of 12) : base case
  Test_toTitlecase_002 (8 of 12) : Greek script
  Test_toUppercase_001 (9 of 12) : boundary case
  Test_toUppercase_002 (10 of 12) : no case
  Test_toUppercase_003 (11 of 12) : base case
  Test_toUppercase_004 (12 of 12) : accented Latin and Greek characters
 -------------------------------------------------------------------------
   12 test cases executed
   0 test cases failed
   1 test case broken
```

Investigation reveals the problem:

```
      )CS #.Tests
#.Tests
      Initial
      Test_CountLettersIn_001 0 0
VALUE ERROR
CountLettersIn[21] Log.Log(⍕bytes),' bytes written to ',tgt
                  ^
```

`Log` is undefined. In the envisaged use in production, it is defined by and local to `TxtToCsv`, the function that calls `CountLettersIn`. We have a similar issue with function `LogError`. That design followed Occam's Razor[16]: (entities are not to be needlessly multiplied) in keeping the log object in existence only while needed. But it now prevents us from testing `CountLettersIn` independently. So we'll refactor `Log` to be a child of `#.MyApp`, created by `Start`:

```
  ...
  ⎕WSID←'MyApp'

  'CREATE!'F.CheckPath'Logs' ⍝ ensure subfolder of current dir
  ∆←L.CreatePropertySpace
  ∆.path←'Logs',F.CurrentSep ⍝ subfolder of current directory
  ∆.encoding←'UTF8'
  ∆.filenamePrefix←'MyApp'
  ∆.refToUtils←#
  Log←⎕NEW L(,⊂∆)

  Log.Log'Started MyApp in ',F.PWD

  LogError←Log∘{code←EXIT⍎⍵ ⋄ code⊣⍺.LogError code ⍵}

  Params←GetParameters mode
  ...
```

And `Initial` will call `#.MyApp.Start 'Session'`.

Now we have both stateless and state-full tests passing. This completes Version 5.

[16]*Non sunt multiplicanda entia sine necessitate.*

Testing in different versions of Windows

When you wrote for yourself, your code needed to run only on the version of Windows you use yourself. To ship it as a product you will have to support it on the versions your customers use.

You need to pick the versions of Windows you will support, and run your tests on all those versions. If you are not already a fan of automated tests, you are about to become one.

For this you will need either

- a test machine for each OS (version of Windows) you support; or
- a test machine and VM (virtual-machine) software

What VM software should you use? One of us has had good results with *Workstation Player* from VMware[17].

If you use VM software you will save a *machine image* for each OS. Include in each machine image your preferred development tools, such as text editor and Dyalog APL. You will need to keep each machine image up to date with fixes and patches to its OS and your tools.

The machine images are large, about 10Gb each. So you want several hundred gigabytes of fast SSD (solid-state drive) on your test machine. With this you should be able to get a machine image loaded in about 20 seconds.

[17]http://www.vmware.com

Make me

It's time to take a closer look at the process of building the application workspace and exporting the EXE. In this chapter we'll

- split the DYAPP into separate versions – one to create the development and testing environment, the other to assemble and export the application;
- automate the export process;
- write tests for the EXE.

We resume, as usual, by saving a copy of `Z:\code\v05` as `Z:\code\v06`.

Make me whole, make me lean

Our makefile, `MyApp.dyapp`, includes scripts we have no reason to include in the exported EXE:

```
Target #
Load ..\AplTree\APLTreeUtils
Load ..\AplTree\FilesAndDirs
Load ..\AplTree\HandleError
Load ..\AplTree\IniFiles
Load ..\AplTree\Logger
Load ..\AplTree\Tester
Load Constants
Load Utilities
Load Tests
Load MyApp
Run MyApp.Start 'Session'
```

`Tester` and `Tests` have no place in the finished application.

We could expunge them before exporting the active workspace as an EXE. Or – have *two* makefiles, one for the development environment, one for export.

Duplicate `MyApp.dyapp` and name the files `Develop.dyapp` and `Export.dyapp`. Edit `Export.dyapp` as follows:

```
Target #
Load ..\AplTree\APLTreeUtils
Load ..\AplTree\FilesAndDirs
Load ..\AplTree\HandleError
Load ..\AplTree\IniFiles
Load ..\AplTree\Logger
Load Constants
Load Utilities
Load MyApp
Run MyApp.Start 'Export'
```

Now, `#.MyApp.Start` doesn't yet have an Export mode, so we'd better give it one:

```
  ...
  :Select mode
  :Case 'Export'
      #.⎕LX←'#.MyApp.Start ''Application''' ⍝ ready to export
  :Case 'Session'
      ⎕←'Alphabet is ',Params.alphabet
      ⎕←'Defined alphabets: ',⍕U.m2n Params.ALPHABETS.⎕NL 2
  :Case 'Application'
      exit←TxtToCsv Params.source
      Off exit
  :EndSelect
```

Notice that Session mode no longer needs to set the Latent Expression.

Refactoring

We also notice now that the contents of `#.MyApp` divide into two groups. One is concerned with setting up the environment and communicating with the operating system. The other does the work of MyApp. We'll now refactor them into separate namespaces: `MyApp` and `Environment`.

`MyApp` will continue to contain the code for counting frequency. But the code for interrogating the environment, setting error traps, and starting logging – we'll move that into `Environment`. And we'll minimise the cross references between the two.

First, all the default parameter values for `MyApp` – the values it has to have in case they are set nowhere else – can go into a simple, static namespace:

```
:Namespace PARAMETERS
    :Namespace ALPHABETS
        English←⎕A
        French←'AÁÂÀBCÇDEÈÊÉFGHIÌÍÎJKLMNOÒÓÔPQRSTUÙÚÛVWXYZ'
        German←'AÄBCDEFGHIJKLMNOÖPQRSßTUÜVWXYZ'
        Greek←'ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ'
    :EndNamespace
    accented←0
    alphabet←'English'
    source←''
    output←''
:EndNamespace
```

We'll leave to `Environment` the definition of `MyApp.Params` and `MyApp.Log` and just make a note of that in `MyApp`:

```
⍝ Objects Log and Params are defined by #.Environment.Start
```

That lets us cut `MyApp` down to size:

```
      )CS MyApp
#.MyApp
      )FNS
CheckAgenda     CountLetters    CountLettersIn  TxtToCsv
      )VARS
ACCENTS ∆
```

`Environment` will get the `Start` function. For extra clarity we'll rename the start modes to `Develop`, `Export` and `Run`.

```
∇ Start mode;∆
⍝ Initialise workspace for development, export or use
⍝ mode: ['Develop' | 'Export' | 'Run']
  :If mode≡'Run`'
      ⍝ trap problems in startup
      #.⎕TRAP←0 'E' '#.HandleError.Process '''''
  :EndIf
  ⎕WSID←'MyApp'
```

It must now create the `Log` object 'over there' – within `#.MyApp`.

```
'CREATE!'#.FilesAndDirs.CheckPath'Logs' ⍝ ensure subfolder of current dir
∆←{
    ⍵.path←'Logs',F.CurrentSep ⍝ subfolder of current directory
    ⍵.encoding←'UTF8'
    ⍵.filenamePrefix←'MyApp'
    ⍵.refToUtils←#
    ⍵
}#.Logger.CreatePropertySpace
#.MyApp.Log←⎕NEW #.Logger(,⊂∆)
```

Read the arguments from the command line:

```
args←⌷2 ⎕NQ'.' 'GetCommandLineArgs'   ⍝ command line
#.MyApp.Log.Log¨('Command line arg ['∘,¨(⍕¨⍳≢args),¨⊂']: '),¨args
```

Set the `PARAMETERS` namespace in `#.MyApp`:

```
#.MyApp.PARAMETERS GetParameters mode args env
```

While we're at this, we've switched the arguments in `GetParameters` to follow an ancient APL convention for functions, that the right argument represents data and any left argument, some modifier for the function.[18]

What happens next depends upon the mode. If we're getting ready to develop, we want tools and tests. We'll ensure global error trapping is off, so we can investigate any errors. And we might as well start by running the tests:

```
:Select mode

:Case 'Develop'
    #.⎕TRAP←0⍴#.⎕TRAP
    ⎕←'Alphabet is ',#.MyApp.PARAMETERS.alphabet
    ⎕←'Defined alphabets: ',⍕U.m2n #.MyApp.PARAMETERS.ALPHABETS.⎕NL 2
    #.Tester.EstablishHelpersIn #.Tests
    #.Tests.Run
```

If we're starting in Export mode, everything is ready to export as an EXE. We'll just display the expression that does that.

[18]the best example of this are the circle functions represented by `○`.

```
  :Case 'Export'
      ⎕←U.ScriptFollowing
      ⍝ Exporting to an EXE can fail unpredictably.
      ⍝ Retry the following expression if it fails,
      ⍝ or use the File>Export dialogue from the menus.
      ⍝      #.Environment.Export '.\MyApp.exe'
```

We'll define that `Export` function in a moment. Right now, we'll set out what the EXE is to do when it runs:

```
  :Case 'Run'
      #.ErrorParms←{
          ⍵.errorFolder←#.FilesAndDirs.PWD
          ⍵.returnCode←#.MyApp.EXIT.APPLICATION_CRASHED
          ⍵.(logFunctionParent logFunction)←(#.MyApp.Log)('Log')
          ⍵.trapInternalErrors←~#.APLTreeUtils.IsDevelopment
      }#.HandleError.CreateParms
      #.⎕TRAP←0 'E' '#.HandleError.Process ''#.ErrorParms'''
      Off #.MyApp.TxtToCsv #.MyApp.Params.source
```

Now that `Export` function. You'll have noticed exporting an EXE can fail from time to time, and you've performed this from the *File* menu enough times to have tired of it. So we'll automate it. Automating it doesn't make it any more reliable, but it certainly makes retries easier.

```
∇ msg←Export filename;type;flags;resource;icon;cmdline;nl;success;try
  #.⎕LX←'#.Environment.Start ''Run'''

  type←'StandaloneNativeExe'
  flags←2 ⍝ BOUND_CONSOLE
  resource←''
  icon←F.NormalizePath '.\images\gear.ico'
  cmdline←''

  success←try←0
  :Repeat
      :Trap 11
          2 ⎕NQ'.' 'Bind',filename type flags resource icon cmdline
          success←1
      :Else
          ⎕DL 0.2
      :EndTrap
  :Until success∨50<try+←1
  msg←⊃success⌽('**ERROR: Failed to export EXE')('Exported ',filename)
  msg,←(try>1)/' after ',(⍕try),' tries'
  #.MyApp.Log.Log msg
∇
```

Basically, the choices you have been making from the *File > Export* dialogue, now wrapped as a function.

One last tweak to the setup for development. Let's have some handy tools defined in `#.MyApp` just for when we're working in there.

```
:Namespace DevTools
⍝ Developer tools
⍝ Vern: sjt25jul16

    fc←{⍺(≢⍵)}⌸ ⍝ frequency count
    same←{⍵≡¨⊂⊃⍵}∘,
    type←{type←{'CN'[⎕IO+0=⊃0⍴⊃⍣≡⍵]}}
    wi←{(≡⍵)(type ⍵)(⍴⍵)} ⍝ what is this array?

:EndNamespace
```

A few tweaks now to the makefiles. `Develop.dyapp`:

```
Target #
Load ..\AplTree\APLTreeUtils
Load ..\AplTree\FilesAndDirs
Load ..\AplTree\HandleError
Load ..\AplTree\IniFiles
Load ..\AplTree\Logger
Load ..\AplTree\Tester
Load Constants
Load Tests
Load Utilities
Load MyApp
Load Environment
Target #.MyApp
Load DevTools -disperse
Run #.Environment.Start 'Develop'
```

Note the `Target #.MyApp` followed by `Load DevTools -disperse`. That disperses the contents of the `DevTools` namespace into `#.MyApp`. (Watch out for name conflicts if you do this.) Now when we're working in `MyApp` we'll have some tools to hand.

Running the DYAPP creates our working environment and runs the tests:

```
clear ws
Booting Z:\code\v06\Develop.dyapp
Loaded: #.APLTreeUtils
Loaded: #.FilesAndDirs
Loaded: #.HandleError
Loaded: #.IniFiles
Loaded: #.Logger
Loaded: #.Tester
Loaded: #.Constants
Loaded: #.Tests
Loaded: #.Utilities
Loaded: #.MyApp
Loaded: #.Environment
Loaded: * 4 objects dispersed in #.MyApp
Alphabet is Russian
Defined alphabets:  English  French  German  Greek  Russian
--- Tests started at 2016-07-25 15:18:17  on #.Tests ---------------------
  Test_CountLettersIn_001 (1 of 12) : across multiple files
  Test_CountLetters_001 (2 of 12) : base case
  Test_toLowercase_001 (3 of 12) : boundary case
  Test_toLowercase_002 (4 of 12) : no case
  Test_toLowercase_003 (5 of 12) : base case
  Test_toLowercase_004 (6 of 12) : accented Latin and Greek characters
  Test_toTitlecase_001 (7 of 12) : base case
  Test_toTitlecase_002 (8 of 12) : Greek script
  Test_toUppercase_001 (9 of 12) : boundary case
  Test_toUppercase_002 (10 of 12) : no case
  Test_toUppercase_003 (11 of 12) : base case
  Test_toUppercase_004 (12 of 12) : accented Latin and Greek characters
 -------------------------------------------------------------------------
   12 test cases executed
   0 test cases failed
   0 test cases broken
```

The foregoing is a good example of how having automated tests allows us to refactor code with confidence that we'll notice and fix anything we break.

Slight tweaks to the export makefile and we can export an EXE. `Export.dyapp`:

```
Target #
Load ..\AplTree\APLTreeUtils
Load ..\AplTree\FilesAndDirs
Load ..\AplTree\HandleError
Load ..\AplTree\IniFiles
Load ..\AplTree\Logger
Load Constants
Load Utilities
Load MyApp
Load Environment
Run #.Environment.Start 'Export'
```

Run this new export makefile and we get a new session:

```
clear ws
Booting Z:\code\v06\Export.dyapp
Loaded: #.APLTreeUtils
Loaded: #.FilesAndDirs
Loaded: #.HandleError
Loaded: #.IniFiles
Loaded: #.Logger
Loaded: #.Constants
Loaded: #.Utilities
Loaded: #.MyApp
Loaded: #.Environment
 Exporting to an EXE can fail unpredictably.
 Retry the following expression if it fails,
 or use the File>Export dialogue from the menus.
      #.Environment.Export '.\MyApp.exe'
```

And if we execute the suggested expression…[19]

```
      #.Environment.Export '.\MyApp.exe'
Exported .\MyApp.exe
```

Testing the EXE

We've written tests for running MyApp from the session. We also need to test whether the EXE works too. We'll write `Test_Exe_001`.

[19]It would be great if `#.Environment.Export` could be run straight from the DYAPP. But the DYAPP has to finish before an EXE can be exported.

It’s not enough to examine the exit code that MyApp.exe returns to Windows. We also need to examine the results. So our test will follow the same strategy used in `Test_CountLettersIn_001`: it will generate a set of test files and a corresponding result table. This time it will run the EXE, then see if the result file exists and holds the correct results.

We start by refactoring out of `Test_Count_LettersIn` the code to be shared with `Test_Exe_001`:

```
∇ failed←Test_CountLettersIn_001(debugFlag batchFlag)
 ⍝ across multiple files
  failed←testOnFiles'APL' 5 'English'
∇

∇ failed←testOnFiles(testmode nfiles alph) ;cd;files;res;cc2n;cf;xxx;alphabet;cmd
  EraseFilesIn TEST_FLDR
  alphabet←#.MyApp.PARAMETERS.ALPHABETS⍎alph
  cf←?1000⍴⍨nfiles,≢alphabet                           ⍝ random freqs for nfile\
s files
  files←{TEST_FLDR,'test',⍵,'.txt'}∘⍕¨⍳nfiles          ⍝ full filenames
  ({⍵[?⍨≢⍵]}¨(↓cf)/¨⊂alphabet)⎕NPUT¨files
  res←(¯1↓TEST_FLDR),'.csv'                            ⍝ result file

  :If ~failed←{~⎕NEXISTS ⍵:0 ⋄ ~⎕NDELETE ⍵}res
      :Select testmode
      :Case 'APL'
          failed←#.MyApp.EXIT.OK≢alphabet #.MyApp.CountLettersIn files res
      :Case 'EXE'
          cmd←'.\myapp.exe source="',TEST_FLDR,'" alphabet=',alph
          xxx←⎕CMD cmd
          failed←~⎕NEXISTS res
      :EndSelect
      :If ~failed
          cc2n←{2 1∘⊃¨⎕VFI¨2↓¨⍵}                        ⍝ CSV col 2 as numbers
          failed←(+⌿cf)≢cc2n⊃⎕NGET res C.NGET.LINES
      :EndIf
  :EndIf
∇

∇ EraseFilesIn folder
  :For file :In ⊃C.NINFO.NAME(⎕NINFO⍠'Wildcard' 1)folder,'*.*'
      ⎕NDELETE file
  :EndFor
∇
```

As we can see, `testOnFiles` sets up the test files and results, then according to its argument, either runs `CountLettersIn` or launches the EXE.

Obvious to say but easy to overlook: after making any changes to the DYALOGs, export a new EXE to ensure that the EXE tests are testing the changes you just made. Then launch Develop.dyapp to run all the tests.

Workflow

With the two DYAPPs, your development cycle now looks like this:

1. Launch Develop.dyapp and review test results.
2. Fix any errors and rerun `#.Tests.Run`. (If you edit the test themselves, either rerun `#,Tester.EstablishHelpersIn #.Tests` or simply close the session and relaunch Develop.dyapp.)
3. Launch Export.dyapp, which will export a new EXE. Close the session.

Documentation – the Doc is in

Documentation is the bad mother of software. Programmers learn early that we depend on it but must not trust it. On the one hand we need it for the software we use. On the other we learn a great wariness of it for the software we develop. Understanding why this is so will help us see what to do about documenting MyApp.

It helps to distinguish three quite different things people refer to as *documentation.*

- instructions on how to use the application
- a description of what the application does
- a description of how the application works

Instructions on how to use the application

Unless you are writing a tool or components for other developers to use, all software is operated through a graphical user interface. Users know the common conventions of UIs in various contexts. The standard for UIs is relatively demanding. If you know what the application is for, it should be obvious how to use its basic features. The application might help you with wizards (dialogue sequences) to accomplish complex tasks. A user might supplement this by consulting what the Help menu offers. She might search the Web for advice. The last thing she is likely to do is go looking for a printed manual.

We'll come in a later chapter to how to offer online help from a Help menu. For now, we mention Help to exclude it from what we mean by *documentation.*

A description of what the application does

This is a useful thing to have, perhaps as a sales document. One or two pages suffices. Including limitations is important: files in certain formats, up to certain sizes. Perhaps a list of Frequently Asked Questions and their answers.

Beyond that, you have the formal tests. This is what you *know* the system does. It passes its tests. Especially if you're supporting your application on multiple versions of Windows, you'll want those tests to be extensive.

A description of how the application works

This is what you want when you revisit part of the code after six months – or six days in some cases. How does this section work? What's going on here?

In the best case the code explains everything. Software is a story told in two worlds. One world is the domain of the user, for example, a world of customer records. The other world is the arrays and namespaces used to represent them.

Good writing achieves a double vision. The transformations described by the code make sense in both worlds. Ken Iverson once coined the term *expository programming* for this writing. Expository programs reveal their workings to the reader. They also discover errors more easily, making it possible to "stare the bugs out". (David Armstrong liked to say the best writing style for a philosopher lets him see his errors before his colleagues do.)

APL requires little 'ceremonial code' – e.g. declarations of data type – and so makes high levels of semantic density achievable. It is perhaps easier to write expository code than in more commonly-used languages. But we have learned great respect for how quickly we can forget what a piece of code does. Then we need documentation in its third sense.

It's in this third sense that we'll discuss *documentation.*

The poor relation

We write software for people and people press us for results, which rarely include documentation. No one is pressing us for documentation.

Documentation is for those who come after us, quite probably our future selves. Since 80% of the lifetime costs of software are spent on maintenance, documentation is a good investment. If the software is ours, we're more likely to make that investment. But there will be constant pressure to defer writing it.

The common result of this pressure is that application code has either no documentation, or its documentation is not up to date. Out-of-date documentation is worse than having none. If you have no documentation you have no help with the code. You have to read it and run it to understand what it does. But however difficult that is, it is utterly reliable. Out-of-date documentation is worse: it will mislead you and waste your time before sending you back to the code. Even if the relevant part of it is accurate, once you learn to distrust it, its value is mostly gone.

The only place worth writing documentation is in the code itself. Maintaining documentation separately adds the uncertainty of matching versions. Writing the documentation as comments in the code encourages you to keep it in step with changes to the code. We write comments in three ways, serving slightly different purposes.

Header comments
A block of comments at the top of a function serves as an abstract, describing argument/s and

result and the relationship between them. If the function reads external variables, list them. If the function has side effects (ie writes files or sets external variables) list them.

Heading comments

Heading comment lines serve exactly as headings in a book or document, helping the reader to navigate its structure.

Trailing comments

:Comments at the ends of lines act as margin notes. Do not use them as a running translation of the code. Instead aim to for expository style and code that needs no translation. on lines where you're not satisfied you've achieved expository style, then do write an explanatory comment. Better to reserve trailing comments for other notes, such as `⍝ FIXME slow for >1E7 elements`. (Using a tag such as `FIXME` makes it easy to bookmark lines for review.) Aligning trailing comments to begin at the same column makes them easier to scan, and is considered *OCD compliant*[20].

The above conventions are simple enough and have long been in wide use.

If you are exporting scripts for others to use – for example, contributing to a library – then it's worth going a step further. You and other *authors* of a script need to read comments in the context of the code, But potential *users* of a script will want to know only how to call its methods.

Automatic documentation generation will extract documentation from your scripts for other users. Just as above, the documentation is maintained as comments in the code. But now header comments are presented without the code lines.

ADOC

ADOC is an acronym for *automatic documentation* generation. It works on classes, and on namespaces where certain conventions are observed.

In its most basic function, it lists methods, properties and fields and requires no comments in the code. In its more powerful function, it composes from header comments an HTML page. Honouring Markdown[21] conventions, it provides all the typographical conventions you need for documentation.

Previously only found as a class in the APLTree library, it is now shipped in Dyalog Version 15.0 as three user commands.

List

Lists the methods and fields of a class. (Requires no comments.)

[20] ^ocd

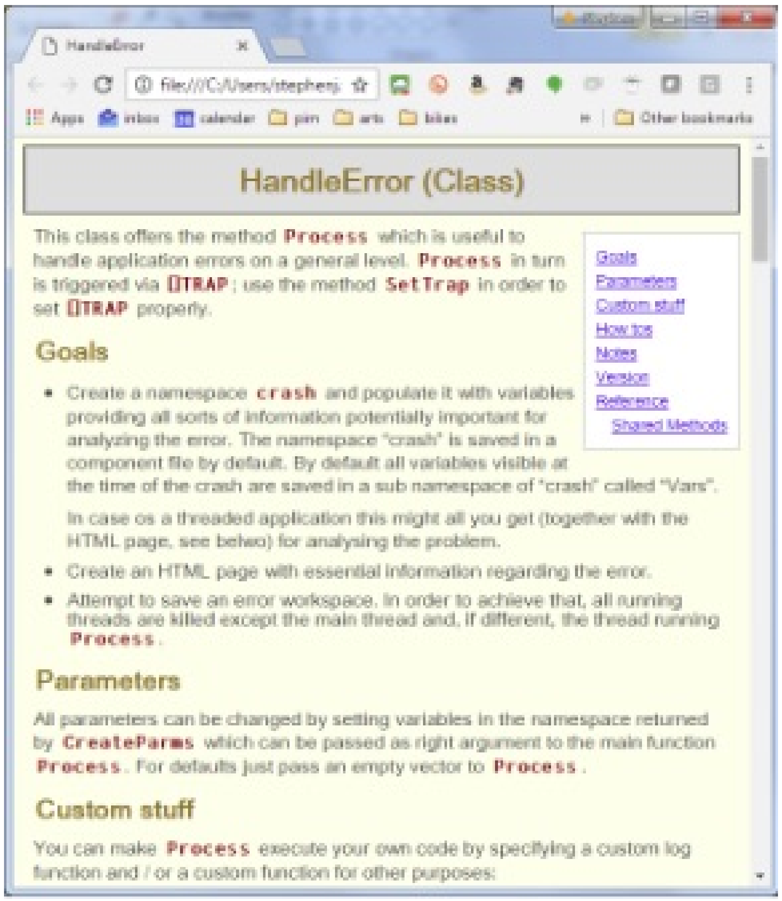
[21] https://en.wikipedia.org/wiki/Markdown

```
      ]adoc_list #.HandleError
*** HandleError (Class) ***

Shared Methods:
  CreateParms
  Process
  ReportErrorToWindowsLog
  SetTrap
  Version
```

Browse

```
]adoc_browse #.HandleError
```



HandleError (Class)

This class offers the method **Process** which is useful to handle application errors on a general level. **Process** in turn is triggered via **⎕TRAP**; use the method **SetTrap** in order to set **⎕TRAP** properly.

Goals

- Create a namespace **crash** and populate it with variables providing all sorts of information potentially important for analyzing the error. The namespace "crash" is saved in a component file by default. By default all variables visible at the time of the crash are saved in a sub namespace of "crash" called "Vars".

  In case os a threaded application this might all you get (together with the HTML page, see belwo) for analysing the problem.
- Create an HTML page with essential information regarding the error.
- Attempt to save an error workspace. In order to achieve that, all running threads are killed except the main thread and, if different, the thread running **Process**.

Parameters

All parameters can be changed by setting variables in the namespace returned by **CreateParms** which can be passed as right argument to the main function **Process**. For defaults just pass an empty vector to **Process**.

Custom stuff

You can make **Process** execute your own code by specifying a custom log function and / or a custom function for other purposes:

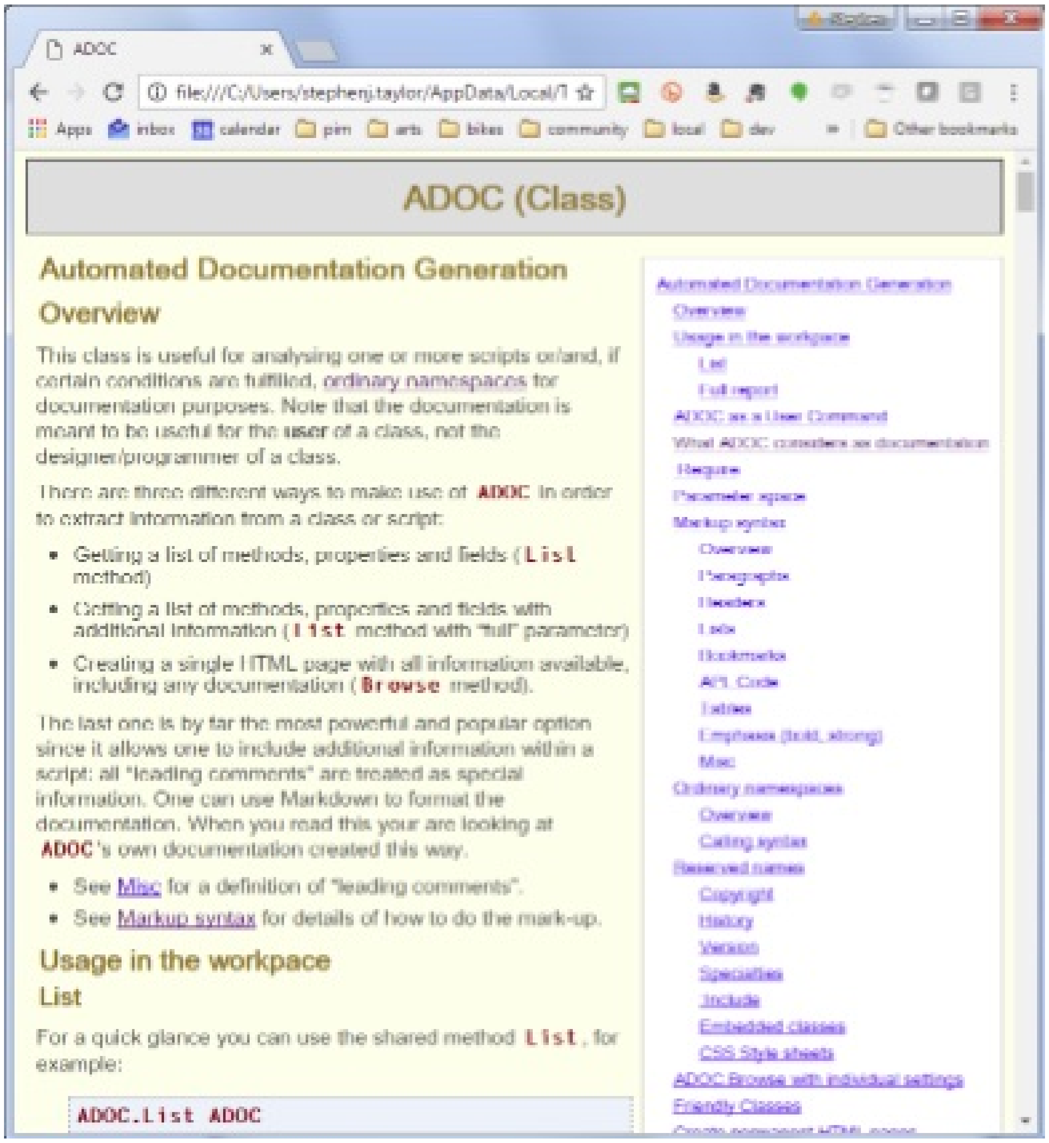
Using ADOC to browse the HandleError class

Composes in HTML a documentation page and displays it in your default browser.

Help

```
]adoc_help
```

Browses the ADOC class itself, displaying all the instructions you need to use it.



ADOC (Class)

Automated Documentation Generation

Overview

This class is useful for analysing one or more scripts or/and, if certain conditions are fulfilled, ordinary namespaces for documentation purposes. Note that the documentation is meant to be useful for the **user** of a class, not the designer/programmer of a class.

There are three different ways to make use of `ADOC` in order to extract information from a class or script:

- Getting a list of methods, properties and fields (`List` method)
- Getting a list of methods, properties and fields with additional information (`List` method with "full" parameter)
- Creating a single HTML page with all information available, including any documentation (`Browse` method).

The last one is by far the most powerful and popular option since it allows one to include additional information within a script: all "leading comments" are treated as special information. One can use Markdown to format the documentation. When you read this your are looking at `ADOC`'s own documentation created this way.

- See Misc for a definition of "leading comments".
- See Markup syntax for details of how to do the mark-up.

Usage in the workpace

List

For a quick glance you can use the shared method `List`, for example:

```
ADOC.List ADOC
```

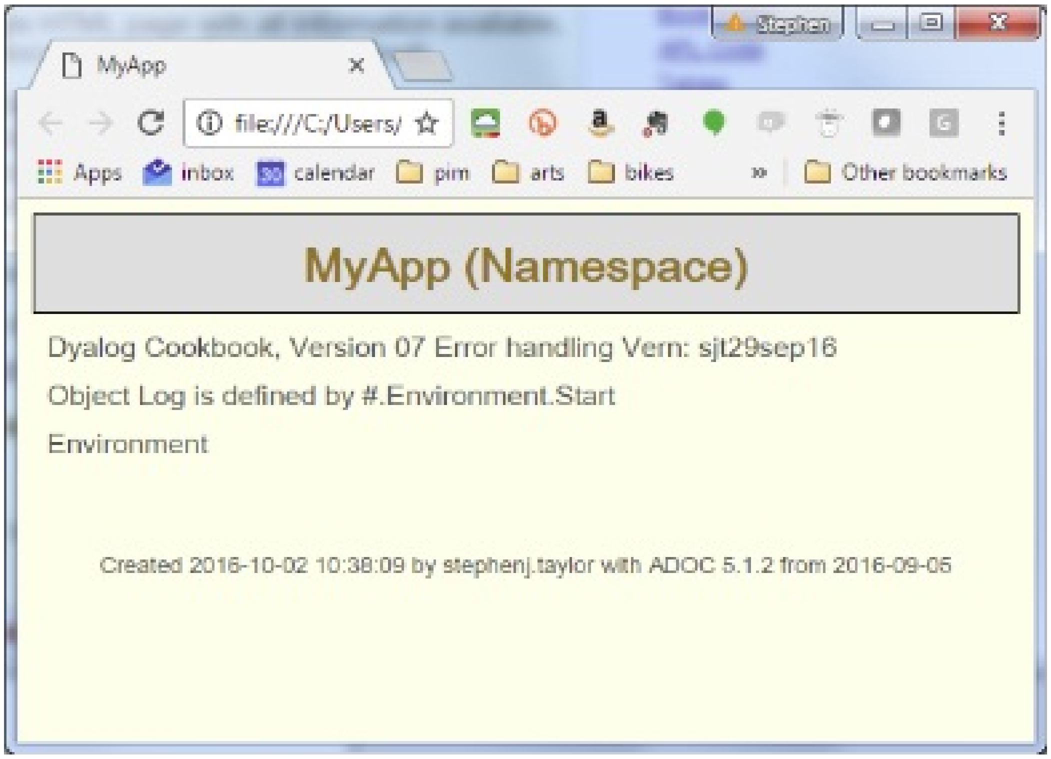
Automated Documentation Generation
  Overview
  Usage in the workpace
    List
    Full report
  ADOC as a User Command
  What ADOC considers as documentation
  Require
  Parameter space
  Markup syntax
    Overview
    Paragraphs
    Headers
    Lists
    Bookmarks
    APL Code
    Tables
    Emphasis (bold, strong)
    Misc
  Ordinary namespaces
    Overview
    Calling syntax
  Reserved names
    Copyright
    History
    Version
    Specialties
    Include
    Embedded classes
    CSS Style sheets
  ADOC Browse with individual settings
  Friendly Classes

ADOC's own documentation

ADOC for MyApp

How might ADOC help us? Start by seeing what ADOC has to say about MyApp as it is now:

```
]adoc_browse #.MyApp
```



MyApp (Namespace)

Dyalog Cookbook, Version 07 Error handling Vern: sjt29sep16

Object Log is defined by #.Environment.Start

Environment

Created 2016-10-02 10:38:09 by stephenj.taylor with ADOC 5.1.2 from 2016-09-05

Using ADOC to browse the MyApp namespace

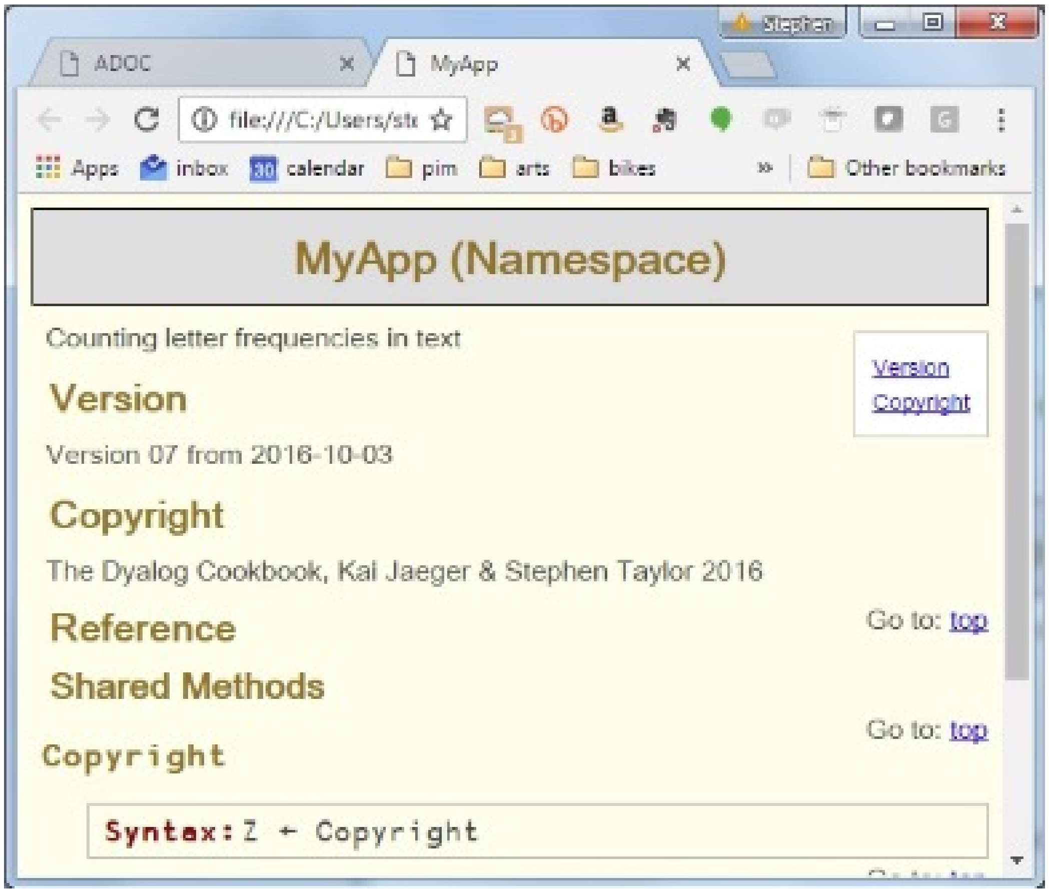
We see that ADOC has found and displayed the script's header comments. We can improve this a little by editing the top of the script to follow ADOC's conventions.

```
:Namespace MyApp
⍝ Counting letter frequencies in text

    ∇ Z←Copyright
      :Access Public Shared
      Z←'The Dyalog Cookbook, Kai Jaeger & Stephen Taylor 2016'
    ∇

    ∇ Z←Version
      :Access Public Shared
      Z←(⍕⎕THIS)'07' '2016-10-03'
    ∇
```

This gives us more prominent copyright and version notices.



MyApp (Namespace)

Counting letter frequencies in text

Version

Version 07 from 2016-10-03

Copyright

The Dyalog Cookbook, Kai Jaeger & Stephen Taylor 2016

Reference

Shared Methods

Copyright

```
Syntax: Z ← Copyright
```

Browsing the revised MyApp namespace

It's not much but then we're not exporting MyApp as a class for others to use. We'll revisit ADOC when we do that.

User interface

Modern graphical user interfaces (GUIs, or more simply, UIs) are a wonder. UI conventions are so widely known it is now unremarkable for people to start using applications without prior training, expecting the software to make clear what they need to do.

This is a high standard to meet, and writing UIs is a deep art. The primary platforms for professional writers of UIs are currently Windows Presentation Foundation (WPF) and a combination of HTML 5 and JavaScript (HTML/JS). These are rich platforms, which enable effective and attractive UIs to be written.

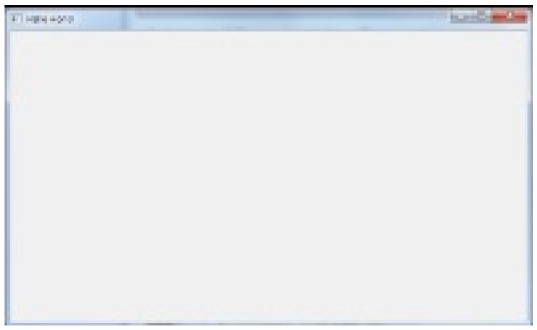
The high quality of these UIs is particularly important for mass-market software, where users are unskilled and unsupported.

WPF and HTML/JS have a high learning threshold. There is much to be mastered before you can write good UIs on these platforms.

You have an alternative. The GUI tools native to Dyalog support perfectly workmanlike GUIs. They exploit and extend your existing knowledge of Dyalog. If you are producing high-value software for a few users, rather than software for casual use by millions, a native Dyalog GUI might be your best platform.

Creating a GUI form in Dyalog could hardly be simpler:

```
UI←⎕NEW⊂'Form'
UI.Caption←'Hello world'
```



Hello world form

To the form we add controls, set callback functions to run when certain events occur, and invoke the form's `Wait` method. See the *Dyalog for Microsoft Windows Interface Guide* for details and tutorials.

Navigating the UI

Embedding a form into an application raises more difficult questions. Chief among them is where to put the form.

It is common for a callback to read or set other controls in the UI. The question is: how to find them? Callback functions always receive in their arguments a reference to the UI control that triggered the callback. How to navigate the UI's hierarchy (tree) of controls?

Keep in mind the following common practices we'll want to accommodate.

- A single callback function is often used to handle an event or events for several controls.
- It is common during development or maintenance to redesign parts of the UI. If you think of the UI as a tree rooted in its form, redesign can move entire branches of the tree.
- We might want multiple instances of the same form. For example, if the form allows us to browse a customer record, we might want two records open at the same time.

Here are some strategies for embedding and navigating the UI tree.

Use absolute names

This is the method used above: `UI←⎕NEW⊂'Form'`. The object `UI` is a child of the workspace root. It's the strategy implied by the interface tutorials. It's easy to read and understand:

```
UI.(MB←⎕NEW⊂'Menubar')
UI.MB.(MenuFile←⎕NEW⊂'Menu'('Caption' '&File'))
```

Notice that the `⎕NEW` that creates each control is executed within its parent. This constructs the UI tree. Notice too that the control is given a name within its parent. So, for example, we can refer to the File menu as `UI.MB.MenuFile`. This is clear enough, but it embeds the structure of the UI into the name of each control. So if we want to move a branch of the UI tree we have to find and edit every reference to controls in that branch.

If we want multiple instances of the form, we will need to pass our code *references* to forms, not *names* of forms.

Navigate relative paths

A callback can navigate the UI tree starting at the control that called it. `obj.##` gives it a reference to the `obj`s parent. Much as you construct relative filepaths, you can construct relative paths to other controls.

This strategy sacrifices a little clarity (you need the UI tree clearly in mind in order to read the path) but by avoiding absolute names it supports multiple instances.

It also reduces the editing required when you move a branch of the UI tree. Relative paths entirely within the moved branch continue to work as before. Only paths that cross into or out of the branch require editing.

Navigate by searching the UI tree

Relative paths support multiple instances and are more robust than absolute paths under changes to the UI, but they still bind the callbacks quite tightly to a particular structure of the tree.

This binding can be loosened by searching the UI tree. For example, a callback from a button could (use a utility function to) find the button's closest ancestor SubForm and thence a Grid object that is a child of the Subform.

A search could start from anywhere in the UI tree.

If you have worked on Web interfaces, you might wish to write utility functions that would implement the CSS-style selector syntax used by JQuery.

Assign unique names to controls

Continuing the line of thought above, writers of JQuery interfaces know the simplest (and fastest) way to identify a control is to use its unique ID. We recommend a similar method for native Dyalog UIs.[22]

In this strategy:

- The function that launches a form (of which the UI might contain many) creates, as a local variable, an empty namespace. Call this the *UI namespace*.
- The form – and all its child controls – is created in this namespace.
- As each control is created, it has a name assigned to it in the namespace.
- As each control is created, it has embedded in it a reference to the UI namespace itself.

The resulting UI namespace contains

- The UI object tree
- A uniquely-named reference to each control

[22]Thanks to Paul Mansour, the first person we know to describe this strategy.

Suppose the reference to the UI namespace is embedded in each control as its `ui` attribute. A callback on control `obj` can thus refer to another control `Foo` simply as `obj.ui.Foo` regardless of the structure of the UI tree.

This depends on every control being created with a `ui` property referring back to the UI namespace. We can protect against failure to define this property. A utility function `GetRef2ui` can search up the UI tree until it finds an ancestor object with this `ui` property defined.

We'll use this approach to build a simple user interface for MyApp. How simple? We'll have a menu bar, from which the language can be selected, and an Edit field onto which files and/or folder can be dropped. Analysis results or error messages get displayed in the edit. That simple.

A simple UI with native Dyalog forms

A new namespace script, UI in which a niladic function `Run` runs the user interface:

⍝ aliases (A E F)⟵#.(APLTreeUtils Environment FilesAndDirs) (M R U)⟵#.(MyApp RefNamespace Utilities)

```
∇ Run;ui
  ui←R.Create'User Interface'
  ui←CreateGui ui
  ui←Init ui
  ui.∆Path←F.PWD
  DQ ui.∆form
  Shutdown
 ⍝ done
∇
```

Here we see the outline clearly. An instance of the RefNamespace class is assigned to `ui`. It is a namespace, empty apart from some standard methods – try `]adoc_browse #.RefNamespace` to see details.

Functions `CreateGui` and `Init` build and initialise the user interface encapsulated in `ui`. Neither function needs to return a result, but doing so means the functions could be chained, for example:

```
  ui←Init CreateGui R.Create'User Interface'
```

Forms

Again, the functional style of `CreateGui` produces expository code.

```
∇ ui←CreateGui ui
  ui.∆LanguageCommands←''
  ui.∆MenuCommands←''

  ui←CreateForm ui
  ui←CreateMenubar ui
  ui←CreateEdit ui
  ui←CreateStatusbar ui
∇
```

The UI namespace gets a couple of empty lists as properties: `∆LanguageCommands` and `∆MenuCommands`. We'll come to those in the menu bar.

Creating the form is also straightforward:

```
∇ ui←CreateForm ui;∆
  ui.Font←⎕NEW'Font'(('Pname' 'APL385 Unicode')('Size' 16))
  ui.Icon←⎕NEW'Icon'(E.IconComponents{↓⍉↑⍵(⍺⍎¨⍵)}'Bits' 'CMap' 'Mask')

  ∆←''
  ∆,←⊂'Coord' 'Pixel'
  ∆,←⊂'Posn'(50 70)
  ∆,←⊂'Size'(400 500)
  ∆,←⊂'Caption' 'Frequency Counter'
  ∆,←⊂'MaxButton' 0
  ∆,←⊂'FontObj'ui.Font
  ∆,←⊂'IconObj'ui.Icon
  ui.∆form←⎕NEW'Form'∆
  ui.∆form.ui←ui
∇
```

But notice key moves in the last two lines. When the form is created, its reference is assigned to a new property of the UI namespace: `∆form`. And, as will all its children, the form is given, as property `ui`, a reference to the UI namespace.

It follows, from any control `obj` in the UI, the form can be referred to as `obj.ui.∆form`.

We'll see this first in creating the menubar.

Controls

Here we create a menubar as a child of the form, which we can refer to as `ui.∆form`. A reference to the menubar is saved in the UI namespace under the name `MB`.

```
∇ ui←CreateMenubar ui
  ui.MB←ui.∆form.⎕NEW⊂'Menubar'

  ui←CreateFileMenu ui
  ui←CreateLanguageMenu ui

  ui.∆MenuCommands.onSelect←⊂'OnMenuCommand'
  ui.∆MenuCommands.ui←ui
∇
```

When both menus have been made, the callback `OnMenuCommand` is set for all the objects in the list `ui.∆MenuCommands`. Presumably that list was populated as a side effect of `CreateFileMenu` and/or `CreateLanguageMenu`. Just so:

```
∇ ui←CreateFileMenu ui
  ui.MenuFile←ui.MB.⎕NEW'Menu'(⊂'Caption' '&File')

  ui.Quit←ui.MenuFile.⎕NEW'MenuItem'(⊂'Caption'('Quit',(⎕UCS 9),'Alt+F4'))
  ui.∆MenuCommands,←ui.Quit
∇
```

Just so: the menu item Quit is created as a child of the File menu, and a reference to it appended to `ui.∆MenuCommands`.

The Language menu has to be created dynamically from the languages defined in `#.MyApp.ALPHABETS`.

In principle we have a serious potential problem here. We're assigning menu items to alphabet names in the UI. The alphabet names are drawn from (among other sources) INI files. They could conflict with names defined during `CreateGui`. Although that seems highly unlikely, we should encapsulate the language names in their own namespace. For now, we've left a comment on the line that might break, and wrapped the assignment in a for-loop rather than using the *each* operator.

```
∇ ui←CreateLanguageMenu ui;alph;mi
  ui.MenuLanguage←ui.MB.⎕NEW'Menu'(⊂'Caption' '&Language')

  :For alph :In U.m2n M.ALPHABETS.⎕NL 2
      mi←ui.MenuLanguage.⎕NEW'MenuItem'(⊂'Caption'alph)
      alph ui.{⍎⍺,'←⍵'}mi ⍝ FIXME possible conflict with control names
      ui.∆LanguageCommands,←mi
  :EndFor
  ui.∆LanguageCommands.Checked←ui.∆LanguageCommands∊ui⍎M.PARAMETERS.alphabet
  ui.∆MenuCommands,←ui.∆LanguageCommands
∇
```

The Language menu items use the `Checked` property to display the current selection. By listing them in the property `∆LanguageCommands`, we can set `Checked` in a single test.

Callbacks and the event queue

A *callback* function receives as right argument information about the event that triggered it, and a reference to the object that fired it. The callback takes its own action and returns a result that tells `⎕DQ` what else to do before moving on to the next event. A result of `0` tells `⎕DQ` to do nothing more.

We've set a single callback function `OnMenuCommand` on all the menu items. In this skeleton interface, a 'portmanteau' function such as `OnMenuCommand` looks a bit excessive. After all, it immediately decides whether it has been invoked from the Quit menu item or one of the Language menu items. Simpler to set one callback on the Quit menu item and a different one on all the Language menu items.

But with many more menu items that strategy produces a 'cloud' of tiny callback functions. More legible to have a single 'portmanteau' callback for all menu items.

```
∇ Z←OnMenuCommand(obj xxx);ui
  ui←GetRef2ui obj
  :Select obj
  :Case ui.Quit
      ⎕NQ ui.∆form'Close'
  :CaseList ui.∆LanguageCommands
      M.PARAMETERS.alphabet←obj.Caption
      ui.∆LanguageCommands.Checked←ui.∆LanguageCommands=obj
  :EndSelect
  Z←0
∇
```

The first move of the callback finds the UI namespace. This should be simply `obj.ui` but in case the `ui` property has not been defined for the invoking control, we use `GetRef2ui`, which either returns the property or searches the object's ancestors until it finds it. (Because the `ui` property was defined for the form itself, we know any search will at worst terminate there.)

```
GetRef2ui←{9=⍵.⎕NC'ui':⍵.ui ⋄ ∇ ⍵.##}
```

Object references are scalars, so the expression `ui.∆LanguageCommands=obj` yields a simple Boolean vector.

Quitting the UI

⎕DQ on the form was started by Run. (Using the cover function DQ, which provides a shell for future logging, tracing and debugging.)

In the OnMenuComamnd callback, if obj was the Quit menu, a Close event is enqueued for the form. When the callback exits, that Close event is the next one ⎕DQ encounters.

When ⎕DQ encounters the Close event for its argument, it closes the object and exits. That terminates DQ. The Shutdown function deletes the form explicitly, rather than relying on Windows to do so when Run leaves the execution stack and the UI namespace in its local variable ui vanishes.

D functions

Most of the UI functions can be written as Dfns and some writers prefer this form. Here as examples are a constructor and a callback.

```
CreateGui←{
    ui←⍵

    ui.∆LanguageCommands←''
    ui.∆MenuCommands←''

    ui←CreateForm ui
    ui←CreateMenubar ui
    ui←CreateEdit ui
    ui←CreateStatusbar ui

    ui
}

OnMenuCommand←{
    (obj xxx)←⍵
    ui←GetRef2ui obj
    obj=ui.Quit:0⊣⎕NQ ui.∆form'Close'
    M.PARAMETERS.alphabet←obj.Caption
    ui.∆LanguageCommands.Checked←ui.∆LanguageCommands=obj
    0
}
```

Providing help

(Choice between CHM generated by say *Help and Manual* or the APLTreeHelp namespace.)

Writing an installer

Working with other processes

Managing your source code

Storing data

Native and component files

XML

JSON

Relational databases

User interface – WPF

User interface – HTTP

Deploying as a Web application

Deploying as a Web service

2. Professional programming practices

Minimise semantic distance

Modern programs are much bigger as are machine memories. In general the scarcest resource is human attention. For the *human* reader, assigning a name to a value (defining a variable) equates to *remember this*. Remember this value, just calculated, by this name. The more such associations the reader has to remembers, and the longer she has to remember them, the greater the demand, the cognitive load, upon the reader.

Semantic distance is the 'distance' in the code between assigning a variable and using it in another expression. Minimise the distance. Use the variable as soon as possible after setting it. Don't ask your reader to remember an association over 10 lines of code if she need only remember it over two.

No names

The minimum semantic distance is zero. This is when you 'use' a value in the same expression in which you calculated it. If you can do this – and have no further need to refer to the value – then avoid assigning it a name at all. Even a short name, such as `x`, is a demand that the reader remember it until at least the end of the function.

You can often use anonymous D-functions (lambdas) within a function to avoid defining variables in it. Assignments *within* the D-function disappear when the D-function leaves the stack, so it's clear the reader has nothing to remember.

For example, suppose you want to apply a function `foo` to the second element of the result of an expression before passing the lot onto another function `bar`. You might write:

```
triple←expression
(2⊃triple)←foo 2⊃triple
bar triple
```

But you can avoid defining `triple`, for which you have no further use and thus no need for the reader to remember, with an anonymous D-function:

```
bar {(a b c)←⍵ ⋄ a (foo b) c} expression
```

Or even an operator:

```
bar 0 1 0 foo{⍺⍺⍣⍺⊢⍵}¨expression
```

Delta, the Heraclitean variable

Sometimes you need a name even though your reader need remember it only until the next line. You might need multiple lines to construct an array:

```
∆←('Dog' 'Mammal')('Cat' 'Mammal')('Carp' 'Fish')
∆,←('Eagle' 'Bird')('Viper' 'Reptile')('Rabbit' 'Mammal')
RegisterAnimals ∆
```

Here we follow a convention that the variable `∆` need not be remembered past the following line.

Stay DRY – don’t repeat yourself

If you say something twice (or more) and later have to change it, you have to change it everywhere you said it. Here are some refactoring techniques for removing duplications from your code.

Define variables once – if that

A man with two watches never knows what time it is. On this principle, we avoid redefining variables.

This seems counterintuitive. The point of a variable is that its content can vary. Early programming practice, working with scarce memory, positively encouraged ‘re-use’ of variables, treating them as bins or containers in which to park data.

Experience has taught us great respect for our capacity for confusion. When tracing and debugging, it is a great comfort to know that having found the definition of a variable, there are no other definitions to consider.

Instead of thinking of assignment as storing some data in a named bin, think of it as assigning a name to the data. Giving the same name to different data sounds lke what it is: a recipe for confusion.

There are two obvious exceptions. Where variables are redefined in a loop we can read the definition as provisional, just for one iteration.

The other is where different possible values are most clearly expressed in control structures. In this case minimise the semantic distance between the different definitions.

```
:if test1
    x←'foo'
:Else
    x←'bar'
:EndIf
```

The above is better expressed using ‘syntactic sweeteners’ (see next section)if the alternative results cost little or nothing to evaluate. But if the alternative definitions of `x` are non-trivial to evaluate (or even just lengthy to express), the control strucure serves better.

APL.local

Every developer, and every development group, has utility functions uses throughout the code base. Most of these can and should be grouped into topic-specific namespaces, such as the FilesAndDirs namespace from the APLTree library. Aliasing the namespaces, such as `F←#.FilesAndDirs` allows utility functions to be called in abbreviated form, e.g. `F.Dir` instead of `#.FilesAndDirs.Dir`.

But some functions are so ubiquitous and general it makes better sense to treat them as your local extensions to the language itself. For example the simplified conditional functions `means` and `else` are 'syntax sweeteners' that allow you to write

```
:if a=b
   Z←'this'
:else
   Z←'that'
:endif
```

more legibly as `Z←(a=b) means 'this' else 'that'`

Functions `means` and `else` could be defined in say a namespace `#.Utilities` and abbreviated to `U`, permitting

```
Z←(a=b) U.means 'this' U.else 'that'
```

But you might reasonably prefer to omit even the `U.` prefix.

Defining ubiquitous utilities

Here is how to define your `Utilities` namespace in the workspace root so that you can refer to them without prefixes.

Every function or object you include in the root as your 'local extension' to the APL language effectively becomes a reserved word in your 'local' APL dialect. Be conservative and define functions this way only when you have found them ubiquitous and indispensable!

FIXME `:Require` in scripts…

Some utilities you might like to make ubiquitous

Hooray for arrays

You’ve already discovered how APL lets you write code that is helpfully light on ‘ceremony’. For example, you don’t have to declare variable types or loop through collections. Here are some more advanced array programming techniques that might give you a further lift.

Functional is funky

Functional style makes your code easier to test, debug and re-use.

Complex data structures

Some ways array programmers commonly use to represent more complex data structures, such as trees, dictionaries and tables.

Passing parameters perfectly

The ghastliness of globals

Nothing is handier when developing your code than keeping bits of information in global variables. They are like the Post-It™ notes or scraps of paper on your real-world desktop. They have no place in your application. Here's why professional programmers keep the global symbol table as empty as possible – and how.

Coding with class

Classes are like micro-workspaces, and a great way of organising your code and data into modules.