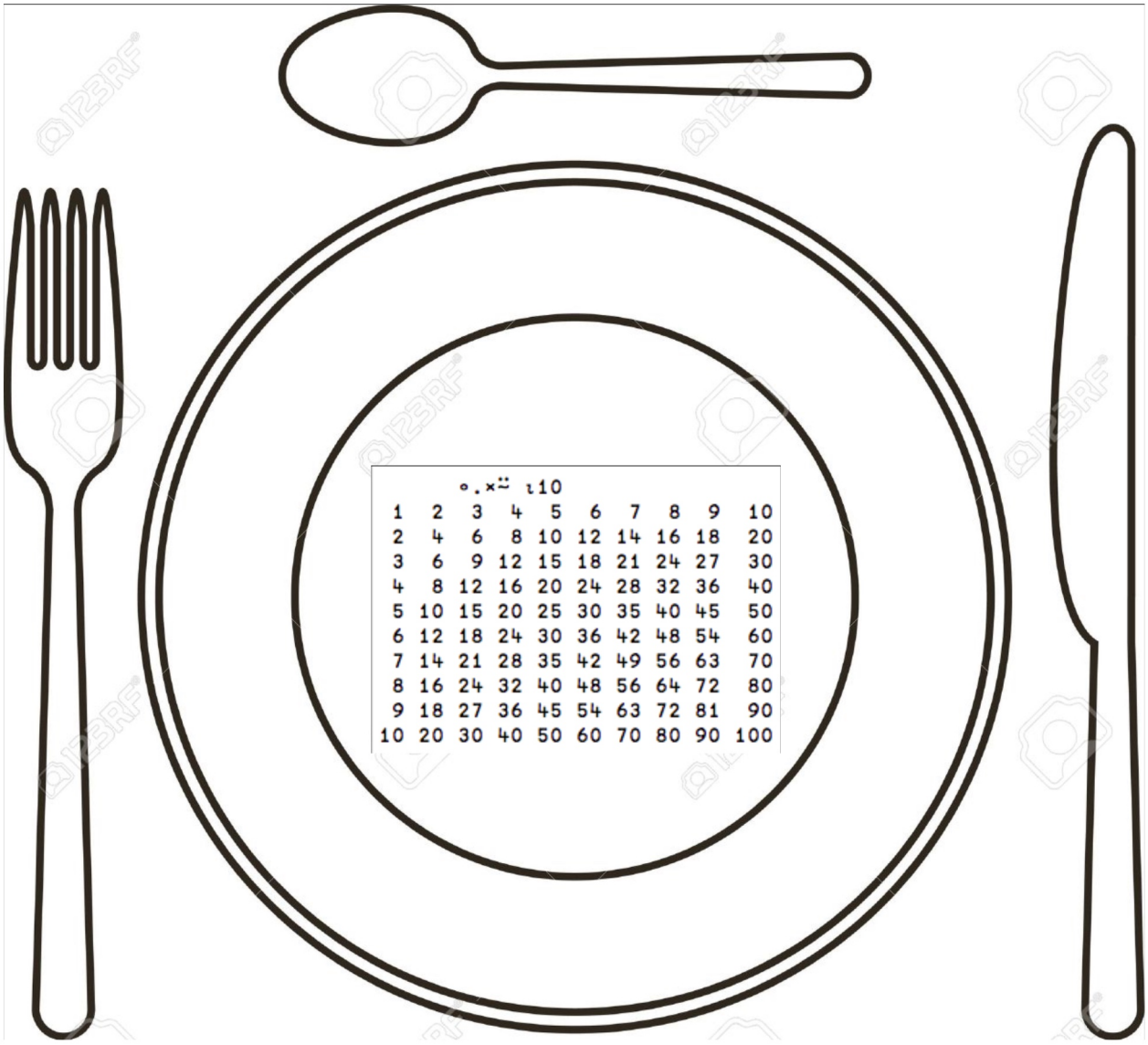

The Dyalog Cookbook

for Windows

Stephen Taylor & Kai Jaeger

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 an 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.

Many of these issues are entangled with each other. We'll arrive at the best solutions by way of some interim solutions, so you get to understand on the way the issues and how their solutions 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 on the book website at dyalog.com. Watch out: they are interim solutions. By all means adapt the last as a framework for your application, but do so knowing how it works!

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. Much of the utility code in the Cookbook is packaged into 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.

Acknowledgements

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

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[^ext] 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. So 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, SourceForge, 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
ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ
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.

`caseUp` 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. 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.

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
\-○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 appliation that uses `#.Utilities`. Everything else is encapsulated within `MyApp`. Here's how to refer in in the MyApp code to these different categories of object.

1. `log←⎕NEW #.Logger`
2. `queue←⎕NEW TaskQueue`
3. `tbl←Engine.CountLetters txt`
4. `r←ok,ok #.Utilities.means r #.Utilities.else 'error'`

That last is pretty horrible, but we can improve it by defining aliases within `#.MyApp`:

```
(C U)←#.(Constants Utilities)
```

allowing (4) to be written as

```
r←ok,ok U.means r 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:

```
r←ok,ok means r 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 wikth 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[1] 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.

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

```
      ↑⊃(⎕NINFO⍠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
```

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++[2] works well.

Here's how the object tree will look:

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

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

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

The file tree will look like this:

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

So `z:\code\v00\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
      WILDCARD←1
   :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

    ⍝ Ubiquitous functions that for local purposes
    ⍝  effectively extend the language
    ⍝ Treat as reserved words: do not shadow

    caseDn←{0(819⌶)⍵}
    caseUp←{1(819⌶)⍵}


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

:EndNamespace
```

And another:

```
:Namespace MyApp

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

⍝ === VARIABLES ===

    ACCENTS←↑'ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ' 'AAAAAACDEEEEIIIINOOOOOOUUUUY'

⍝ === End of variables definition ===

    CountLetters←{
      {⍺(≠⍵)}⌸A{⍵≠⍨⍵∈⍺}(↓ACCENTS)U.map U.caseUp ⍵
    } ⍝ 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←1 ⎕NINFO fullfilepath
      :Case 1 ⍝ folder
          tgt←fullfilepath,csv
          files←⊃(⎕NINFO⍠⎕C.NINFO.WILDCARD)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
    - WILDCARD
  - NPUT
    - OVERWRITE
- LocalAPL
  - caseDn
  - caseUp
  - map
- MyApp
  - ACCENTS
  - CountLetters
  - TxtToCsv
```

If you also see containers `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.*

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:\>MyApp.exe 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`. Now edit `Z:\code\v02\MyApp.dyapp` to include some library code:

```
Target #
Load ..\AplTree\APLTreeUtils
Load ..\AplTree\Logger
Load ..\AplTree\WinFile
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.

```
⍝ Aliases (referents must be defined previously)
    (A L W)←#.(APLTreeUtils Logger WinFile) ⍝ 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 `full-filepath`. 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 the EXE's containing folder. We know that exists: *executo ergo sum.*

In developing and testing MyApp, we create the active workspace by running `MyApp.dyapp`. That sets the Current Directory as the DYAPP's container. That too must exist.

```
      #.WinFile.Cd ''
Z:\code\v02
```

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

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

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;Δ;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
      'CREATE!'W.CheckPath'Logs' ⍝ ensure subfolder of current dir
      Δ←L.CreatePropertySpace
      Δ.path←'Logs\' ⍝ subfolder of current directory
      Δ.encoding←'UTF8'
      Δ.filenamePrefix←'MyApp'
      Δ.refToUtils←#
      Log←⎕NEW L(,⊂Δ)
      Log.Log'Started MyApp in ',W.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, and in the Windows command line run the new `MyApp.exe`.

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

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

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

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

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

```
      t 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.

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
⍝ Read command parameters, run the application
 ⎕OFF TxtToCsv 2>2↑⎕2 ⎕NQ'.' 'GetCommandLineArgs'
 ∇
```

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;Δ;xxx;Log;Error;files;tgt
 ⍝ Write a sibling CSV of the TXT located at fullfilepath,
 ⍝ containing a frequency count of the letters in the file text
  'CREATE!'W.CheckPath'Logs' ⍝ ensure subfolder of current dir
  Δ←L.CreatePropertySpace
  Δ.path←'Logs\' ⍝ subfolder of current directory
  Δ.encoding←'UTF8'
  Δ.filenamePrefix←'MyApp'
  Δ.refToUtils←#
  Log←⎕NEW L(,⊂Δ)
  Log.Log'Started MyApp in ',W.PWD
  Log.Log'Source: ',fullfilepath

  Error←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 `∼×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. `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←1 ⎕NINFO fullfilepath)∈C.NINFO.TYPE.(DIRECTORY FILE)
      (files exit)←(Error 'INVALID_SOURCE')('')
  :ElseIf ~⎕NEXISTS fullfilepath
      (files exit)←(Error 'SOURCE_NOT_FOUND')('')
  :Else
      :Select type
      :Case C.NINFO.TYPE.DIRECTORY
          files←⊃(⎕NINFO⍠C.NINFO.WILDCARD)fullfilepath,'\*.txt'
      :Case C.NINFO.TYPE.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←Error '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←Error'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. 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\MyApp.dyapp`:

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

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

```
(A H L W)←#.(APLTreeUtils HandleError Logger WinFile) ⍝ from APLTree
```

Define a new exit code constant:

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

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

```
∇ StartFromCmdLine
 ⍝ Read command parameters, run the application
  ⎕TRAP←0 'E' '#.HandleError.Process ''''' ⍝ trap unforeseen problems
  ⎕OFF TxtToCsv 2>2↑⎕2 ⎕NQ'.' 'GetCommandLineArgs'
∇
```

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:

```
  ...
Error←Log∘{code←EXIT⍠⍵ ⋄ code⍪⍺.LogError code ⍵}

isDev←'Development'≡4↑'.'⎕WG'APLVersion'
⍝ refine trap definition
#.ErrorParms←H.CreateParms
#.ErrorParms.errorFolder←W.PWD
#.ErrorParms.returnCode←EXIT.APPLICATION_CRASHED
#.ErrorParms.(logFunctionParent logFunction)←Log'Log'
#.ErrorParms.trapInternalErrors←~isDev
:If 'Development'≡4↑'.'⎕WG'APLVersion'
    ⎕TRAP⍴⎕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 stop[3] in `CompileFiles` after reading the first file.

```
    :EndTrap
    . ⍝ DEBUG
:Until (≠files)<i←i+1
```

This will definitely break. It is not caught by any of the other traps. Export the workspace as before, run it from the DOS command shell…

[3]The poets among us love that the tersest way to bring a function to a full stop is to type one.

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

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:

```
FIXME
```

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.

Remove the deliberate error from `#.MyApp` and save your work.

Discussion

[SJT] What's a convenient way to show the exit codes returned to the Windows command shell?

Initialising with INI files

Testing: the sound of breaking glass

Our app here is as simple as we could make it – just join a folder of text files into a single TXT. For many purposes, simply tieing all the source files to a vector of tie numbers `srcs` and creating a new file on tie number `tgt` would reduce the app to a single line:

```
(⎕NREAD¨srcs,¨82,¨⎕NSIZE¨srcs) ⎕NAPPEND¨ tgt
```

Pretty much all the other code has been written to package that 'app' 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 is liable to 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[4], author of *Extreme Programming Explained*. We've written code to allow for ways in which the file system might surprise us. We should write tests to discover if that code works. We'll eventually discover conditions we haven't foreseen and write fixes for them. Now those conditions too join the things we think might break, and get added to the test suite.

[4] in conversation with one of the authors.

Testing in different versions of Windows

User interface – native

A simple UI with native Dyalog forms

Forms

Controls

Callbacks and the event queue

Extended controls

Providing help

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

Writing an installer

Working with other processes

Launching tasks

Running as a Windows service

Sharing files

Communicating through TCP/IP

Managing your source code

Documentation

FIRE

CompareSimple

Useful user commands

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

Delta, the Heraclitean variable

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{αα⍤α⊢ω}¨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:

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

Here we follow a convention that the variable `Δ` 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.

The obvious exception is where different possible values are most clearly expressed in control structures, in which case minimise the semantic distance between the different definitions.

```
:if test1
:ElseIf test2 ⋄ x←'foo'
:Else ⋄ x←'bar'
:EndIf
```

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 WinFile namespace from the APLTree library. Aliasing the namespaces, such as `W←#.WinFile` allows utility functions to be called in abbreviated form, e.g. `W.Dir` instead of `#.WinFile.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!

…

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.