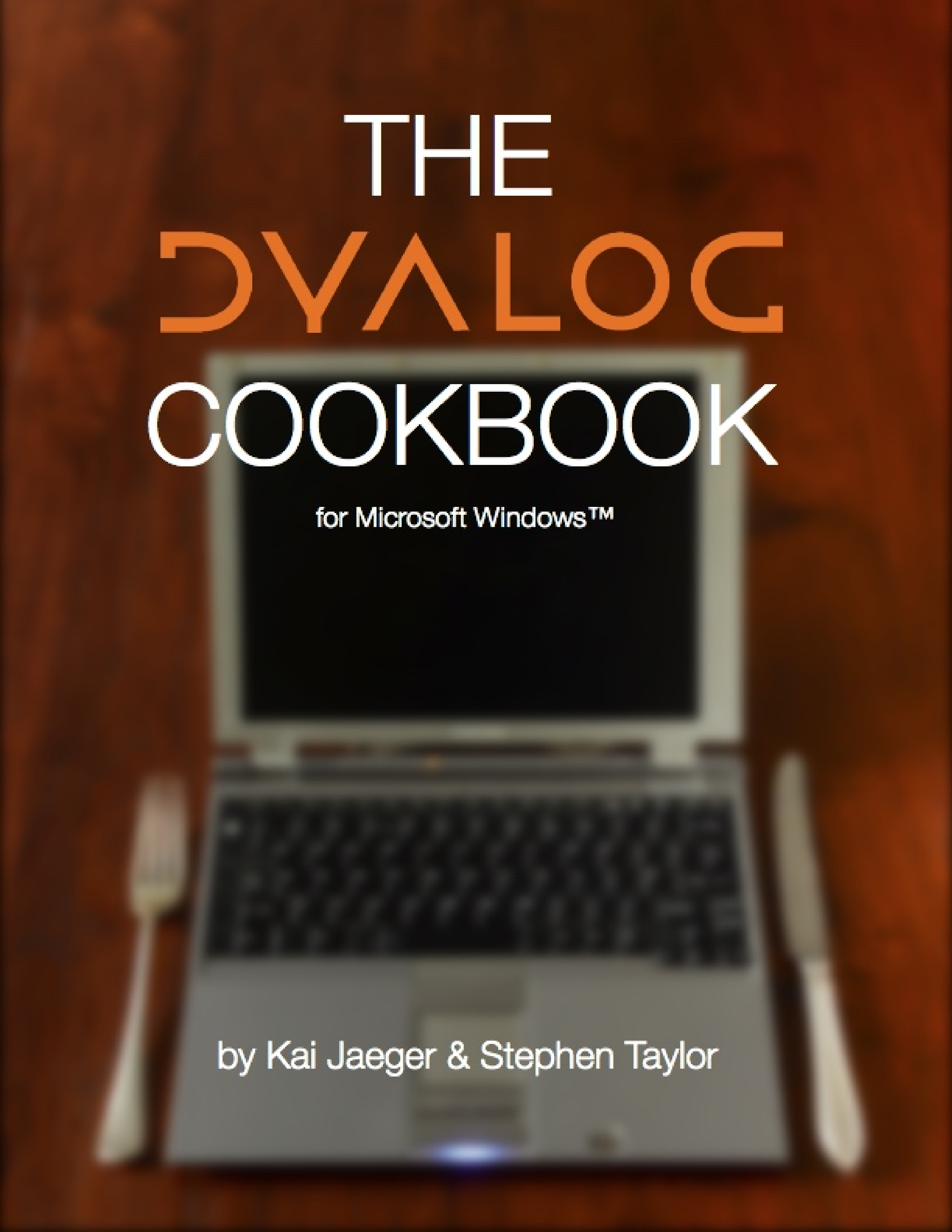
THE DYALOG COOKBOOK

for Microsoft Windows™



by Kai Jaeger & Stephen Taylor

The Dyalog Cookbook

Kai Jaeger and Stephen Taylor

This book is for sale at http://leanpub.com/thedyalogcookbook

This version was published on 2017-08-16

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

1. 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.

1.1 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 and applications from the APLTree [2] library.

But the development tools introduced by Dyalog in recent years are still finding their places with development teams. 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.

[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. You will however find that whenever possible we keep the code platform independent. If we use platform dependent utilities we will mentioned it and explain why, and we might mention alternatives available on other platforms.

[2] You can download all members of the APLTree library from the APL Wiki: http://download.aplwiki.com/

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.

We will also improve the code along the way while explaining why exactly the changes are improvements.

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.

Working through the book, you get to understand how the implementation issues and the solutions to them work. In the first chapters 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, constantly improved along the way.

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.

Later on 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. This includes third-party tools, configuring your development environment and discussing user commands.

1.2 What you need to use the Dyalog Cookbook

- The Dyalog Version 16.0 Unicode interpreter or later.
- Good knowledge of APL - the Cookbook is by no means an introduction.
- 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 [3]. 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: how to call the static methods of a class (sufficient in most cases) or how to create an instance of a class and use its methods and properties.

  See *Dyalog Programmer's Reference Guide* for an introduction.
- A good understanding of SALT, Dyalog's built-in code management system that allows you to load and save scripts either automatically in the background or at will.
- Internet access. Not necessarily all the time but probably most of the time. Not only because it gives you access to the APL wiki and the Dyalog forum (see below) but mainly for acessing the APLTree tools and the web site that is associated with this book: https://cookbook.dyalog.com.

  However, we also tried to write the book in a way so that you can just read it if that's what you prefer.

We have not attempted to 'dumb down' our use of the language for readers with less experience. In 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.

[3]These days seasoned programmers often have strong opinions about whether to use an object-oriented approach or a functional approach, or to mix the both. We have seen friendships going bust on discussing these issues. In this book we take a mixed approach, and we will discuss the pros and cons of each of them.

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.

In case you still need help the Dyalog Forum provides access to a competent and friendly community around Dyalog.

1.3 Conventions

Note that we assume `⎕IO←1` and `⎕ML←1`, not because we are making a statement, but because that's the Dyalog default. That keeps the Cookbook in sync with the Dyalog documentation.

Getting deeper

In case we want to discuss a particular issue in more detail but we are not sure whether the reader is ready for this, now or ever, we format the information this way.

Sometimes we need to warn you, for example in order to avoid common traps. This is how that would look like.

Sometimes we want to provide a tip, and here's how that would look like.

When we refer to a text file, e.g. something with the extension .txt, then we refer to it as a TXT. We refer to a dyalog script (*.dyalog) as a DYALOG. We refer to a dyapp script (*.dyapp) as a DYAPP. You get the pattern.

1.4 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.

We jealously claim any errors as entirely our own work.

Kai Jaeger & Stephen Taylor

2. 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. Some of the things we are going to add to this application will seem like overkill, but keep in mind that we use this application just as a very simple example for all the techniques we are going to introduce.

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 introduce a DYAPP script to assemble an active workspace from them. Using scripts to store your source code has many advantages: You can use a traditional source code management system rather than having your code and data stored in a binary blob. Changes that you make to your source code are saved immediately, rather than relying on your remembering to save the workspace at some suitable point in your work process. Finally, you don't need to worry about crashes in your code or externally called modules and also any corruption of the active workspace which might prevent you from saving it.

Corrupted workspaces

The *workspace* (WS) is where the APL interpreter manages all code and all data in memory. The Dyalog tracer / debugger has extensive edit-and-continue capabilities; the downside is that these have been known to occasionally corrupt the workspace.

The interpreter checks WS integrity every now and then; how often can be influenced by setting certain debug flags; see the appendix "Workpace integrity, corruptions and aplcores" for details.

2.1 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.

Crash workspaces

A crash workspace is a workspace that was saved by a function what was initiated by error trapping, typically by a setting of `⎕TRAP`. It's a snapshot of the workspace at the moment an unforseen problem triggered error trapping to take over. It's usually very useful to analyze such problems.

Note that a workspace cannot be saved when more than one thread is running.

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 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, something we will discuss in the chapter *Handling Errors.*

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!

2.2 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 anyway.

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, Git 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.

Source code management with acre

Some members of the APL community prefer to use a source code management system that is tailored to solve the needs of an APL programmer, or a team of APL programmers: acre. APL code is very compact, teams are typically small, and work on APL applications tends to be very oriented towards functions rather than modules. Other aspects of working in APL impact the importance of features of the SCM that you use. acre is an excellent alternative to Git etc., and it is available as Open Source; we will discuss acre in its own appendix.

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) for the application. The help system will be an ordinary — non-scripted — namespace. We us a *build script* (DYAPP) to assemble the application as well as the development environment.

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.

2.3 The LetterCount workspace

We suppose you already have a workspace in which your program runs. We don't have your 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.

So we'll begin with the LetterCount 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 code. You can download it from the book's web site: https://cookbook.dyalog.com.

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.

2.4 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 current 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 current 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 create a workspace MyApp.dws from scripts: a DYAPP and some DYALOGs. We'll call it Version 1.

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

Investigating the workspace LetterCount

Let's load the workspace `LetterCount` and investigate it a bit.

Function `TxtToCsv` takes the filepath of a TXT and writes a sibling CSV [1] 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.'
      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
```

[1]With version 16.0 Dyalog has introduced a system function `⎕CSV` for both importing from and exporting to CSV files.

Note that we use a variable `∆` here. Not exactly a memorable or self-explaining name. However, we use `∆` whenever we collect data for temporary use.

`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 – `toUppercase`, `join` and `map` are utilities, of general use.

Note that we have some functions that start with lowercase characters while others start with uppercase characters. In a larger application you might want to be able to tell data from calls to functions and operators by introducing consistent naming conventions. Which one you settle for is less important then choosing something consistent. And don't forget to put it into a document any programmer joining the team is supposed to read first.

`toUppercase` uses the fast case-folding I-beam introduced in Dyalog 15.0 (also available in 14.0 & 14.1 from revision 27141 onwards).

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

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.

The APLTree library is an open source project hosted in the APL wiki. It attempts to provide solutions for many every-day problems a Dyalog APL programmer might run into. In the Cookbook we will use many of its members. For details see http://aplwiki.com/CategoryAplTree.

Start at the root namespace (`#`). We're going to be conservative about defining names in `#`. 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 `#` with other objects you don't know anything about right now.

So your program will be a self-contained object in `#`. 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 `#`. If you're using classes or namespaces that other systems might also use, define them in `#`. 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 `#`, eg `#.Utilities`.

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

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

`⍟` denotes a namespace, `○` a class. These are the characters (among others) you can use to tell the editor what kind of object you wish to create, so for a class `)ed ○ Foo`. Press F1 with the cursor on `)ed` in the session for details.

Note that we keep the user interface (`UI`) separate from the business logic. This is considered good practise because whatever you believe right now, you will almost certainly consider to exchange a particular type of UI (say .NET Windows forms) against a different one (say HTML+JavaScript). This is difficult in any case but much easier when you separate them right from the start. However, our application is so simple that we collect all its code in a namespace script `MyApp` in order to save one level in the namespace hirarchy.

If this were to be a serious project then you would not do this even if the amount of code is small: application tend to change and grow over time, sometimes significantly. Therefore you would be better prepared to have, say, a namespace `MyApp` that contains, say, a namespace script `engine` with all the code.

The objects in `#` 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. `status←(bar>3) #.Utilities.means 'ok' #.Utilities.else 'error'`

⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ ⌧TODO⌧ Andy made a good point here that this is a diversion he was not happy with. Maybe this should be moved elsewhere? ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧

The last one is pretty horrible. It needs some explanation.

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

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

Some equivalents in Dyalog:

- Control structure

  ```
  :If bar>3
      status←'ok'
  :Else
      status←'error'
  :EndIf
  ```

- Pick -1-

  ```
  status←(⎕IO+bar>3)⊃'error' 'ok'
  ```

- Pick -2-

  ```
  status←⊃(bar>3)⌽'error' 'ok'
  ```

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

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

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

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

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

  We use the term "alias" her for a reference pointing to a particular script or namespace. In this context it is important to note that after executing `C←#.Constants` the alias `C` is *identical* to `#.Constants`, therefore ‘ 1 ←→ C≡#.Constants‘.

```
    C←#.Constants ⋄ U←#.Utilities
```

allowing it to be written as

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

What style you prefer is mainly a matter of personal taste, and indeed even the authors do not necessarily agree on this. There are however certain rules you should keep in mind:

Execution time

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

In this approach two user defined functions are called. Not much overhead but don't go for this if the line is, say, executed thousands of times within a loop.

Keep the end user in mind

The authors have done pair programming for years with end users being the second party. For a user a statement like:

```
taxfree←(dob>19491231) U.means 35000 U.else 50000
```

is easily readable despite it being formed of APL primitives and user defined functions. In an agile environment when the end user is supposed to discuss business logic with implementors this can be a big advantage.

For classes however there is another way to do this: include the namespace `#.Utilities`. In order to illustrate this let's assume for a moment that `MyApp` is not a namespace but a class.

```
:Clase MyApp
:Include Utilities
...
:EndClass
```

This requires the namespace `#.Utilities` to be a sibling of the assumed class `MyApp`. Now within the class you can do

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

yet Shift+Enter in the Tracer or the Editor still works, and any changes would go into `#.Utilities`.

More about :Include

When a namespace is :Included, the interpreter will execute functions from that namespace as if they had been defined in the current class. However, the actual *code* is shared with the original namespace. For example, this means that if the code of `means` or `else` is changed while tracing into it from the `MyApp` class those changes are reflected in `#.Utilities` immediately (and any other classes that might have :Included it.

Most of the time, this works as you expect it to, but it can lead to confusion, in particular if you were to `)COPY #.Utilities` from another workspace. This will change the definition of the namespace, but the class has pointers to functions in the old copy of `#.Utilities`, and will not pick up the new definitions until the class is fixed again. If you were to edit these functions while tracing into the `MyApp` class, the changes will not be visible in the namespace. Likewise, if you were to `)ERASE #.Utilities`, the class will continue to work until the class itself is edited, at which point it will complain that the namespace does not exist.

Let's assume that in a WS `C:\Test_Include` we have just this code:

```
:Class Foo
:Include Goo
:EndClass

:Namespace Goo
∇ r←Hello
    :Access Public Shared
      r←'World'
    ∇
:EndNamespace
```

Now we do this:

```
Foo.Hello
world
      )Save
Saved...
      ⎕EX 'Goo'
      Goo
VALUE ERROR
      Foo.Hello
world
)copy c:\Test_Include Goo
copied...
```

If you would at this stage edit `Goo` and change `'world'` to `'Universe'` and then call again `Foo.Hello` it would still print `world` to the session.

If you experience this sort of confusion, it is a good idea to re-fix your classes (in this case `Foo`). Building a fresh WS from source files might be even better.

Be careful with diamonds

The `:If - :Then - :else` solution could have been written this way:

```
:If bar>3 ⋄ status←'ok' ⋄ :Else ⋄ status←'error' ⋄ :EndIf
```

There is one major problem with this: when executing the code in the Tracer the line will be executed in one go. If you think you might want to follow the control flow and trace into the individual expressions, you should spread the control structure over 5 lines.

In general: if you have a choice between a short and a long expression then your are advised to go for the short one unless the long one offers an incentive like improved readability, better debugging or faster execution speed; only a short program has a chance of being bug free.

Diamonds can be useful in some situations, but in general it's a good idea to avoid them.

Diamonds

In some circumstances diamonds are quite useful:

- To make sure that no thread switch takes place between two statements. Something like

  ```
  tno←filename ⎕nTIE 0 ⋄ l←ρ⎕nread tno 80 (⎕nsize tno) ⋄ ⎕nuntie tno
  ```

  is guaranteed to be executed as a unit. Depending on the circumstances this can be really important.
- Make multiple assignments on a single line as in `⎕IO←1 ⋄ ⎕ML←3 ⋄ ⎕PP←20`. Not for variable settings, just system stuff.
- Assignments to `⎕LX` as in `⎕LX←#.FileAndDirs.PolishCurrentDir ⋄ ⎕←Info`.
- To make dfns more readable as in `{w←ω ⋄ ((w='¯')/w)←'-' ⋄ ω}`. There is really no reason to make this a multi-line dfn.

  (Note that from version 16 onwards you can achieve the same result with `{'-'@(⍸ω='¯')⊣ω}`)
- You *cannot* trace into a one-line dfn. This can be quite useful. For example, this function:

  ```
   OnConfigure←{(4↑ω),((⊃α){(0∊ρα):ω ⋄ α⌈ω}ω[4]),((⊃⌽α){(0∊ρα):ω ⋄ α⌈ω}ω[5])}
  ```

  makes sure that a GUI Form (window) is not going to be smaller than a minimum size defined by `α`.

  You don't want to have a multi-line dfn here because then you won't be able to trace into any `⎕DQ` (or `Wait`) statement any more; the number of "Config" events is simply overwhelming. Thanks to the `⋄` we can solve the task on a single line and prevent the Tracer from ever entering the dfn.

Why not use `⎕PATH`?

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

```
status←(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 in our experience its use quickly leads to confusion about which functions are called or edited, and where new names are created. We recommend that you avoid `⎕PATH` if reasonable alternatives are available.

Convert the WS LetterCount into a single scripted namespace.

If your own application is already using scripted namespaces and/or classes then you can skip this, of course.

We assume you have downloaded the WS and saved it as `Z:\code\v00\LetterCount`.

Note that all the stuff in that WS lives in `#`. We have to change that so that all the stuff lives in a single namespace `MyApp`. In order to achieve that execute the following steps:

1. Start an instance of Dyalog
2. Execute `)ns MyApp` in order to create a namespace `MyApp` in the workspace.
3. Execute `)cs MyApp` in order to change *into* `MyApp`, making `MyApp` effectively the *current namespace.*
4. Execute `)copy Z:\code\v00\LetterCount` in order to copy all functions and the single variable into the current namespace which happens to be `#.MyApp`.
5. Execute `)copy Z:\code\v00\LetterCount ⎕IO ⎕ML`

   This makes sure that we really use the same values for important system variables as the WS by copying their values into the namespace `#.MyApp`.
6. Execute `]save #.MyApp Z:\code\v01\MyApp -makedir -noprompt`

The last step will save the contents of the namespace `#.MyApp` into `Z:\code\v01\MyApp.dyalog`. In the case that the folder `v01` or any of its parents do not already exist the –makedir option will cause them to be created. `-noprompt` makes sure that `]save` does not ask any questions.

This is how the script would look like:

```
:Namespace MyApp
⍝ === VARIABLES ===

Accents←2 28⍴'ÁÂÃÀÄÅÇĐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝAAAAAACDEEEEIIIINOOOOOOUUUUY'


⍝ === End of variables definition ===

(⎕IO ⎕ML ⎕WX ⎕PP ⎕DIV)←1 1 3 15 1

 CountLetters←{
     ⍝ Table of letter frequency in txt
     {⍺(≢⍵)}⌸⎕A{⍵/⍨⍵∊⍺}(↓Accents)map toUppercase ⍵
 }

∇ noOfBytes←TxtToCsv fullfilepath;NINFO_WILDCARD;NPUT_OVERWRITE;tgt;files;path;stem;txt;enc;nl;lines\
;csv
     ⍝ Write a sibling CSV of the TXT located at fullfilepath,
     ⍝ containing a frequency count of the letters in the file text.
 NINFO_WILDCARD←NPUT_OVERWRITE←1 ⍝ constants
 fullfilepath~←'"'
 csv←'.csv'
 :Select 1 ⎕NINFO fullfilepath
 :Case 1 ⍝ folder
     tgt←fullfilepath,'\total',csv
     files←⊃(⎕NINFO⍠NINFO_WILDCARD)fullfilepath,'\*.txt'
 :Case 2 ⍝ file
     (path stem)←2↑⎕NPARTS fullfilepath
     tgt←path,stem,csv
     files←,⊂fullfilepath
 :EndSelect
     ⍝ assume txt<<memory
 (txt enc nl)←{(⊃,/1⊃¨⍵)(1 2⊃⍵)(1 3⊃⍵)}⎕NGET¨files
 lines←','join¨↓⍕¨CountLetters txt
     ⍝ use encoding and NL from first source file
 noOfBytes←(lines enc nl)⎕NPUT tgt NPUT_OVERWRITE
     ⍝Done
∇

 join←{
     α←⎕UCS 13 10
     (-≢α)↓⊃,/ω,¨⊂α
 }

 map←{
     (old new)←α
     nw←∪ω
     (new,nw)[(old,nw)⍳ω]
 }

 toUppercase←{1(819⌶)ω}

:EndNamespace
```

There might be minor differences depending on the version of the `]save` user command and the version of SALT you are actually using.

This is the easiest way to convert any ordinary workspace into one or more scripted namespaces.

We start improving based on this version.

2.5 Project Gutenberg

We'll raid Project Gutenberg 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
```

2.6 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 it is a file it will write a sibling CSV. If it is a folder it will read all TXT files in the folder, count the letter frequencies and write them as a CSV file sibling to that folder. Simple enough. Here we go.

2.7 Building from a DYAPP

In your text editor open a new document.

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++ works well but make sure that the editor converts TAB into spaces; by default it does not, and Dyalog does not like TAB characters.

You can even make sure that Windows will call Notepad++ when you enter "notepad.exe" into a console window or double-click a TXT file: google for "notepad replacer".

Here's how the object tree will look:

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

We've saved the very first version as `z:\code\v01\MyApp.dyalog`. Now we take a copy of that and save it as `z:\code\v02\MyApp.dyalog`. Alternatively you can download version 2 from the book's website of course.

Note that compared with version 1 we will improve in several ways:

- We create a DYAPP which will assemble the workspace for us.
- We define all constants we need in a scripted namespace `Constants` which has a sub-namespace `NINFO` which in turn has a sub-namespace `TYPES`.
- The three utility functions go into their own separate namespace script `Utilities`.

The file tree will look like this:

```
z:\code\v02\Constants.dyalog
z:\code\v02\MyApp.dyalog
z:\code\v02\Utilities.dyalog
z:\code\v02\MyApp.dyapp
```

`MyApp.dyapp` looks like this if we take the simple approach:

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

This is the `Constants.dyalog` script:

```
: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
```

Note that we use uppercase here for the names of the "constants" (they are of course not really constants but ordinary variables so far). It is a common convention in most programming languages to use uppercase letters for constants.

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

This is the `Utilities.dyalog` script:

```
:Namespace Utilities
      map←{
          (old new)←⍺
          nw←∪⍵
          (new,nw)[(old,nw)⍳⍵]
      }
    toLowercase←0∘(819⌶)
    toUppercase←1∘(819⌶)
:EndNamespace
```

Finally the `MyApp.dyalog` script:

```
:Namespace MyApp

   (⎕IO ⎕ML ⎕WX ⎕PP ⎕DIV)←1 1 3 15 1

⍝ === Aliases

    U←##.Utilities ⋄ C←##.Constants

⍝ === VARIABLES ===

    Accents←↑'ÁÂÃÀÄÅÇĐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ' 'AAAAAACDEEEEIIIINOOOOOOUUUUY'

⍝ === End of variables definition ===

      CountLetters←{
          {⍺(≢⍵)}⌸⎕A{⍵⌿⍨⍵∊⍺}(↓Accents)U.map U.toUppercase ⍵
      }

    ∇ noOfBytes←TxtToCsv fullfilepath;csv;stem;path;files;lines;nl;enc;tgt;tbl
   ⍝ Write a sibling CSV of the TXT located at fullfilepath,
   ⍝ containing a frequency count of the letters in the file text.
      fullfilepath~←'"'
      csv←'.csv'
      :Select C.NINFO.TYPE ⎕NINFO fullfilepath
      :Case C.TYPES.DIRECTORY
          tgt←fullfilepath,'total',csv
          files←⊃(⎕NINFO⍠'Wildcard' 1)fullfilepath,'\*.txt'
      :Case C.TYPES.FILE
          (path stem)←2↑⎕NPARTS fullfilepath
          tgt←path,stem,csv
          files←,⊂fullfilepath
      :EndSelect
      (tbl enc nl)←{(⊂⍪⊃⍵)1↓⍵)}(CountLetters ProcessFiles) files
      lines←{⍺,',',⍕⍵}/⊃{⍺(+/⍵)}⌸/↓[1]tbl
      noOfBytes←(lines enc nl)⎕NPUT tgt C.NPUT.OVERWRITE
    ∇

    ∇(data enc nl)←(fns ProcessFiles) files;txt;file
   ⍝ Reads all files and executes `fns` on the contents. `files` must not be empty.
      data←⍬
      :For file :In files
          (txt enc nl)←⎕NGET file
          data,←⊂fns txt
      :EndFor
    ∇

:EndNamespace
```

This version comes with a number of improvements. Let's discuss them in detail:

- We address `Utilities` as well as `Constants` with "`##.`": that works as long as they are siblings of `MyApp`. "`#.`" would of course work as well but is inferior. For example, one day you might want to convert this application into a user command; then `##.` will continue to work while `#.` might or might not work, depending on what happens to be in the workspace at the time of execution. The same would be true if making it a ASP.NET application: those have no concept of a "root" at all.
- We have changed the assignment of the `Accents` variable so that we don't need to know the length of it any more.
- It is good programming practise to *not* use any numeric constants in your code. `TxtToCsv` tried to avoid this to some extend by assigning descriptive names at the start and using 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 `#`. We have also replaced the remaining integers in the `:Case` statements by references to symbolic variables in `Constants.NINFO.Type`.
- We have replaced the line `(txt enc nl)←{(⊃,/1⊃¨⍵)(1 2⊃⍵)(1 3⊃⍵)}⎕NGET¨files` by a `:For` loop and moved the code into `ProcessFiles`. Why?
  - We have two fewer local variables.
  - It keeps `TxtToCsv` nice and short.
  - When something goes wrong with a file (corrupted, missing rights, tied by another proces…) then the original version would not even allow you to identify easily what file is causing the problem. Now you would know exactly which file is causing a problem without any effort.
  - Tracing through a function can be painful in case there is any kind of loop involved while you are not interested in the loop at all.

Now it is simply a choice of whether you want to trace *into* `ProcessFiles` or not.
In short: way more often than not it is a good idea to move loops (`:For`, `:Repeat`, `:While`) into their own function (or operator) doing just the loop.

- `ProcessFile` is an operator rather than a function. Currently it takes the function `CountLetters` as operand, but it could be any other function that's supposed to do something useful with the contents of those files. Therefore having `ProcessFiles` as an operator is more general.
- Because of `enc` and `nl` we have to have two lines anyway, but if we weren't interested in `enc` and `nl` a one-liner would do: `tbl⍪←CountLetters ⊃⎕NGET file`. Is this a good idea?

  The answer is no. In case you have to inspect what comes from several files because one (or more) of them contain something unexpected you want to be able to check what you've got, one by one. By separating it on two lines you can open an edit window on `txt`, put a stop vector on line 5 and you can easily check on the contents of one file after the other.
- Although the system settings are done in `MyApp` that's not exactly ideal because these settings should be set for everything in the WS, in particular `#`.

  Naturally this is important for the "Utilities" script because as soon as we introduce a function into it that depends on either `⎕IO` or `⎕ML` we might or might not be in trouble.

  But there is more to it: when we execute a statement like `ref←#.⎕NS ''` (note the `#.`!) then the (unnamed) namespace created by this statement would inherit all the system settings from its parent `#`. Now we can safely assume that you have configured your session according to your needs. However, when you start the app with a double-click on the DYAPP then you might be on a different machine with different settings. In that case you have no idea what you are going to get.

  It is therefore safer - and strongly recommended - to make sure that the setting is well-defined. We will come back to this later.

If you see any namespaces called `SALT_Data` ignore them. They are part of how SALT manages meta data for scripted objects.

We have converted the saved workspace to a bunch of text files, and invented a DYAPP that assembles the workspace from the DYALOGs. But we have not saved a workspace; we will always assemble a workspace from scripts.

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
  - C
  - CountLetters
  - TxtToCsv
  - U
- Utilities
  - map
  - toLowercase
  - toUppercase
```

Note that `MyApp` contains `C` and `U`. That means that the code in the script got executed in the process of assembling the WS, otherwise they wouldn't exist. That's nice because when you type `#.MyApp.C.` then autocomplete pops in and suggests all the names contained in `Constants`.

We have reached our goal:

- Everything is now stored in text files
- With a double-click on `MyApp.dyapp` we can assemble the WS.
- Along the way we have improved the quality of the code, making it more readable and easier to debug.

3. Package MyApp as an executable

Now we will make some adjustments in order to make `MyApp` ready for being packaged as an EXE. It will run from the command line and it will run 'headless' – without a user interface (UI).

Copy all files in `z:\code\v02\` to `z:\code\v03\`. Alternatively you can download version 3 from https://cookbook.dyalog.com.

3.1 Output to the session log

In a runtime interpreter or an EXE, there is no APL session, and output to the session which would have been visible in a development system will simply disappear. If we want the user to be able to view this output, we need to write it to a log file.

But how do we find out where we need to make changes? We recommended that you think about this from the start, and make sure that all *intentional* output goes through a "Log" function, or at least use an explicit `⎕←` so that output can easily be located in the source.

Unwanted output to the session

What can you do if you have output appearing in the session and you don't know where in your application it is being generated? The easiest way is to associate a callback function with the `SessionPrint` event as in:

```
  '⎕se' ⎕WS 'Event' 'SessionPrint' '#.Catch'
  #.⎕FX ↑'what Catch m'  ':If 0∊⍴what' '. ⍝ !' ':Else' '⎕←what' ':Endif'
  ⎕FX 'test arg'  '⎕←arg'
  test 1 2 3
⍎SYNTAX ERROR
Catch[2] . ⍝ !
```

You can even use this to investigate what is about to be written to the session (the left argument of `Catch`) and make the function stop when it reaches the output you are looking for. In the above example we check for anything that's empty.

Don't try the `⎕se.onSessionPrint←'#.Catch'` syntax with `⎕SE`; just stick with `⎕WS` as in the above example.

Don't forget to clear the stack after `Catch` crashed because if you don't and instead call `test` again it would behave as if there was no handler associated with the `SessionPrint` event.

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

3.2 Reading arguments from the command line

`TxtToCsv` needs an argument. The EXE we are about to create must take it from the command line. We'll give `MyApp` a function `StartFromCmdLine`. We will also introduce `SetLX`: the last line of the DYAPP will run it to set `⎕LX`:

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

In `MyApp.dyalog`:

```
:Namespace MyApp

(⎕IO ⎕ML ⎕WX ⎕PP ⎕DIV)←1 1 3 15 1

    ∇r←Version
    ⍝ * 1.0.0
    ⍝   * Runs as a stand-alone EXE and takes parameters from the command line.
      r←(⍕⎕THIS) '1.0.0' 'YYYY-MM-DD'
    ∇
    ...
    ⍝ === VARIABLES ===

    Accents←'ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ' 'AAAAAACDEEEEIIIINOOOOOOUUUUY'

⍝ === End of variables definition ===

      CountLetters←{
          {⍺(≢⍵)}⌸⎕A{⍵⌿⍨⍵∊⍺}Accents map toUppercase ⍵
      }
    ...
    ∇ {r}←SetLX dummy
    ⍝ Set Latent Expression (needed in order to export workspace as EXE)
     r←⍬
     ⎕LX←'#.MyApp.StartFromCmdLine #.MyApp.GetCommandLineArgs ⍬'
    ∇

    ∇ {r}←StartFromCmdLine arg
    ⍝ Run the application; arg = usually command line parameters .
       r←⍬
       r←TxtToCsv arg~''''
    ∇
    ∇ r←GetCommandLineArgs dummy
      r←⊃¯1↑1↓2 ⎕NQ'.' 'GetCommandLineArgs' ⍝ Take the last one
    ∇

:EndNamespace
```

Changes are emphasized.

Now MyApp is ready to be run from the Windows command line, with the name of the file to be processed following the command name.

Notes:

- By introducing a function `Version` we start to keep track of changes.
- `Accents` is now a vector of text vectors (vtv). There is no point in making it a matrix when `CountLetters` (the only function that consumes `Accents`) requires a vtv anyway. We were able to simplify `CountLetters` as a bonus.
- Functions should return a result, even `StartFromCmdLine` and `SetLX`. Always. Otherwise, by definition, they are not functions.

  If there is nothing reasonable to return as a result, return `⍬` as a shy result as in `StartFromCmdLine`. Make this a habit. It makes life easier in the long run. One example is that you cannot call a “function” from a dfn that does not return a result. Another one is that you cannot provide it as an operand to the `⍣` (power) operator.
- *Always* make a function monadic rather than niladic even if the function does not require an argument right now.

  It is way easier to change a monadic function that has ignored its argument so far to one that actually requires an argument than to change a niladic function to a monadic one later on, especially when the function is called in many places, and this is something you *will* run into; it’s just a matter of time.
- `GetCommandLineArgs` ignores its right argument. It makes that very clear by using the name “dummy”. (“ignored” would be fine as well while “to_be_ignored” would not because that might well be a list of objects to be ignored by the function) When you change this at one stage or another then of course you have to change that name to something meaningful.
- Make sure that a `⎕LX` statement can be executed from anywhere. That requires the path to be fully qualified, therefore `#.MyApp` rather than `MyApp`. Make that a habit too. You won’t regret it when later on you want to execute the statement:

  ```
  ⍎⎕LX
  ```

  when you are not in root.
- You may wonder whether `#.MyApp.(StartFromCmdLine GetCommandLineArgs ⍬)` is better than `#.MyApp.StartFromCmdLine #.MyApp.GetCommandLineArgs ⍬` because it is shorter. Good point, but there is a drawback: you cannot <Shift+Enter> on either of the two functions within the shorter expression but you can with the longer one.
- Currently we allow only one file (or folder) to be specified. That’s supposed to be the last parameter specified on the command line. We’ll improve on this later.

We're now nearly ready to create the first version of our EXE. Note that in Dyalog's "File" menu this is called "Export" for some reason.

1. Double-click the DYAPP in order to create the WS.
2. From the "File" menu pick "Export".
3. Pick `Z:\code\v03` as the destination folder [1].
4. From the list "Save as type" pick "Standalone Executable".
5. Set the "File name" as `MyApp`.
6. Check the "Runtime application" check box
7. Make sure that the "Console application" check box is *not* ticked.
8. Click "Save".

You should see a message: *File Z:\code\v03\MyApp.exe successfully created.* This occasionally (rarely) fails for no obvious reason. If it does fail just try again and you should be fine. If it keeps failing then the by far most common reason is that the EXE is running - you cannot replace an EXE while it is running.

Although you cannot replace a running exe what you *can* do is to rename it; that's possible. You can then create a new EXE with the original name.

In case you wonder what a "Console application" actually is:

- It sets the `IMAGE_SUBSYSTEM_WINDOWS_CUI` flag in the header of the EXE. The effect is that, when called *on a command line* (also known as the console), it will wait for the program to return.
- You can access the variable `ERRORLEVEL`. Yes, this implies that without ticking the check box "Console application" you *cannot* access this environment variable.
- When double-clicked a console window pops up.

Note that it catches the return code and assigns it to the environment variable "ERRORLEVEL" in any case.

Note that you cannot really debug a console application with Ride; for details see the "Debugging a stand-alone EXE" chapter.

If you do not tick "Console application", the program is started as a separate process and you cannot catch the return code.

It is therefore recommended not to tick the "Console application" check box unless you have a good reason to do so.

Use the *Version* button to bind to the EXE information about the application, author, version, copyright and so on. These pieces of information will show in the "Properties/Details" tab of the resulting EXE. Note that in order to use the cursor keys or "Home" or "End" *within* a cell the "Version" dialog box requires you to enter "in-cell" mode by pressing F2.

[1]Note that in the Cookbook the words "folder" and "directory" are used interchangeably.

You might specify an icon file to replace the Dyalog icon with your own one.

Let’s run it. From a command line:

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

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

4. 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 file/folder:

```
Z:\code\v03\MyApp.exe Z:\texts\Does_not_exist
```

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 file or folder `Z:\texts\Does_not_exist`. That triggered an error in the APL code. The interpreter tried to display an error message and looked for input from a developer from the session. But a runtime task has no session, so at that point the interpreter popped the alert message and `MyApp` died.

Prior to version 16.0, as soon as you close the message box a CONTINUE workspace was created in the current directory. Such a CONTINUE WS can be loaded and investigated, making it easy to figure out what the problem is. However, this is only true if it is a single-threaded application since workspaces cannot be saved when more than one thread is running.

With version 16.0 you can still force the interpreter to drop a CONTINUE workspace by enabling the old behaviour with `2704⌶ 1` while `2704⌶ 0` would disable it, but that's the default anyway.

Note that for analyzing purposes a CONTINUE workspace must be loaded in an already running instance of Dyalog. In other words: don't double-click a CONTINUE! The reason is that `⎕DM` and `⎕DMX` are overwritten in the process of booting SALT, meaning that you loose the error message. You *may* be able to recreate them by re-executing the failing line but that might be dangerous, or fail in a different way when executed without the application having been initialised properly.

The next version of `MyApp` could do better by having the program recording what happens to a log file.

Save a copy of `Z:\code\v03` as `Z:\code\v04` or copy `v04` from the Cookbook's website.

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\apltree`.[1] Now edit `Z:\code\v04\MyApp.dyapp` to include some library code:

[1] You can download all members of the APLTree library from the APL Wiki: http://download.aplwiki.com/

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

and run the DYAPP to recreate the `MyApp` workspace.

Getting help with any APLTree members

Note that you can ask for a detailed documentation for how to use the members of the APLTree project by executing:

```
]ADoc APLTreeUtils
```

You'll find more information regarding `ADoc` in the chapter "Documentation – the Doc is in".

The `Logger` class and its dependencies will now be included when we build `MyApp`:

- `APLTreeUtils` is a namespace that contains some functions needed by most applications. All members of the APLTree library depend on it.
- `FilesAndDirs` is a class that offers methods for handling files and directories.
- `OS` contains a couple of OS-independent methods for common tasks. `KillProcess` is just an example. `FilesAndDirs` needs `OS` under some circumstances.

Let's get the program to log what it's doing. Within `MyApp`, some changes. First we introduce some more aliases for the new modules:

```
⍝ === Aliases (referents must be defined previously)

    F←##.FilesAndDirs ⋄ A←##.APLTreeUtils   ⍝ from the APLTree lib
```

Note that `APLTreeUtils` comes with the functions `Uppercase` and `Lowercase`. We have those already in the `Utilities` namespace. This violates the DRY principle. We should get rid of one version and use the other everywhere. But how to choose?

First of all, almost all APLTree projects rely on `APLTreeUtils`. If you want to use this library then we cannot get rid of `APLTreeUtils`.

The two different versions both use the Dyalog `⌶` function, so comparing functionality and speed won't help.

However, `APLTreeUtils` is in use for more than 10 years now, it comes with a comprehensive set of test cases and it is documented in detail. That makes the choice rather easy.

Therefore we remove the two functions from `Utilities` and change `CountLetters`:

```
      CountLetters←{
          {⍺(≢⍵)}⌸⎕A{⍵⌿⍨⍵∊⍺}Accents U.map A.Uppercase ⍵
      }
```

That works because the alias `A` we've just introduced points to `APLTreeUtils`.

4.1 Where to keep the logfiles?

Where is `MyApp` to write the logfile? We need a folder 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:

```
Z:\code\v04\MyApp.exe Z:\texts\en
```

Current directory is `Z:\` and therefore that's where the logfiles will appear.

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 to know where they are. We will put them into a sub folder `Logs` within the current directory.

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 for us. That too is sure to exist.

```
      #.FilesAndDirs.PWD
Z:\code\v04
```

Now we set up the parameters needed to instantiate the Logger class. First we use the Logger class' shared `CreateParms` method to get a parameter space with an initial set of default parameters. You can use the built-in method `∆List` to display its properties and their defaults:

```
      #.Logger.CreateParms.∆List''
  active                 1
  autoReOpen             1
  debug                  0
  encoding            ANSI
  errorPrefix      *** ERROR
  extension            log
  fileFlag               1
  filename
  filenamePostfix
  filenamePrefix
  filenameType        DATE
  path
  printToSession         0
  timestamp
```

We then modify those where the defaults don't match our needs and use the parameter space to create the Logger object. For this we create a function `OpenLogFile`:

```
∇ instance←OpenLogFile path;logParms
  ⍝ Creates an instance of the "Logger" class.
  ⍝ Provides methods `Log` and `LogError`.
  ⍝ Make sure that `path` (that is where log files will end up) does exist.
  ⍝ Returns the instance.
  logParms←##.Logger.CreateParms
  logParms.path←path
  logParms.encoding←'UTF8'
  logParms.filenamePrefix←'MyApp'
  'CREATE!'F.CheckPath path
  instance←⎕NEW ##.Logger(,⊂logParms)
∇
```

Notes:

- We need to make sure that the current directory contains a sub-directory `Logs`. That's what the method `FilesAndDirs.CheckPath` will ensure when the left argument is the string `'Create!'`.
- We change the default encoding – that's "ANSI" – to "UTF-8". Note that this has pros and cons: it allows us to write APL characters to the log file but it will also cause potential problems with any third-party tools dealing with log files, because many of them only support ANSI characters.

  Although we've changed it here for demonstration purposes we recommend sticking with ANSI unless you have a *very* good reason not to. When we introduce proper error handling in chapter 6, we will do away with the need for having APL characters in the log file.
- Since we have not changed either `autoReOpen` (1) or `filenameType` ("DATE") it tells the `Logger` class that it should automatically close a log file and re-open a new one each day at 23:59:59. It also defines (together with `filenamePrefix`) the name of the log file.
- If we would run `OpenLogFile` and allow it to return its result to the session window then something similar to this would appear:

```
[Logger:Logs\MyApp_20170211.log(¯87200436)]
```

- "Logger" is the name of the class the object was instantiated from.
- The path between `:` and `(` tell us the actual name of the log file. Because the `filenameType` is "DATE" the name carries the year, month and day the log file was opened.
- The negative number is the tie number of the log file.

We create a function `Initial` (short for "Initialize") which calls `OpenLogFile` and returns the `Logger` instance:

```
∇ {MyLogger}←Initial dummy
⍝ Prepares the application.
   #.⎕IO←1 ⋄ #.⎕ML←1 ⋄ #.⎕WX←3 ⋄ #.⎕PP←15 ⋄ #.⎕DIV←1
⍝
```

At the moment `Initial` is not doing too much, but that will change. Note that we took the opportunity to make sure that all the system settings in `#` are set according to our needs. `MyApp` sets these variables for itself but within `Initial` we now make sure that `#` uses the same values as well, no matter what the session defaults are.

We also need to change `ProcessFile`:

```
∇ data←(fns ProcessFiles)files;txt;file
⍝ was: (data enc nl)←(fns ProcessFiles)files;txt;file
⍝ Reads all files and executes `fns` on the contents.
   data←⍬
   :For file :In files
       txt←'flat' A.ReadUtf8File file
       ⍝ was: (txt enc nl)←⎕NGET file
       data,←⊂fns txt
   :EndFor
∇
```

We use `APLTreeUtils.ReadUtf8File` rather than `⎕NGET` because it optionally returns a flat string without a performance penalty, although that is only an issue with really large files. This is achieved by passing "flat" as the (optional) left argument to `ReadUtf8File`. We ignore encoding and the new line character and allow it to default to the current operating system. As a side effect `ProcessFiles` won't crash anymore when `files` is empty because `enc` and `nl` have disappeared from the function.

Now we have to make sure that `Initial` is called from `StartFromCmdLine`:

```
∇ {r}←StartFromCmdLine arg;MyLogger
⍝ Needs command line parameters, runs the application.
   r←⍬
   MyLogger←Initial ⍬
   MyLogger.Log'Started MyApp in ',F.PWD
   MyLogger.Log #.GetCommandLine
   r←TxtToCsv arg~''''
   MyLogger.Log'Shutting down MyApp'
∇
```

Note that we now log the full command line. In an application that receives its parameters from the command line this is an important thing to do.

We take the opportunity to move code from `TxtToCsv` to a new function `GetFiles`. This new function will take the command line argument and return a list of files which may contain zero, one or many filenames:

```
 ∇ (target files)←GetFiles fullfilepath;csv;target;path;stem
 ⍝ Investigates `fullfilepath` and returns a list with files
 ⍝ May return zero, one or many filenames.
   fullfilepath~←'"'
   csv←'.csv'
   :If F.Exists fullfilepath
       :Select C.NINFO.TYPE ⎕NINFO fullfilepath
       :Case C.TYPES.DIRECTORY
           target←F.NormalizePath fullfilepath,'\total',csv
           files←⊃F.Dir fullfilepath,'\*.txt'
       :Case C.TYPES.FILE
           (path stem)←2↑⎕NPARTS fullfilepath
           target←path,stem,csv
           files←,⊂fullfilepath
       :EndSelect
       target←(~0∊⍴files)/target
   :Else
       files←target←''
   :EndIf
 ∇
```

We have to make sure that `GetFiles` is called from `TxtToCsv`. Note that moving code from `TxtToCsv` to `GetFiles` allows us to keep `TxtToCsv` nice and tidy and the list of local variables short. In addition we have added calls to `MyLogger.Log` in appropriate places:

```
∇ rc←TxtToCsv fullfilepath;files;tbl;lines;target
⍝ Write a sibling CSV of the TXT located at fullfilepath,
⍝ containing a frequency count of the letters in the file text
   (target files)←GetFiles fullfilepath
   :If 0∊⍴files
       MyLogger.Log'No files found to process'
       rc←1
   :Else
       tbl←⊃⍪/(CountLetters ProcessFiles)files
       lines←{⍺,',',⍕⍵}/{⍵[⍒⍵[;2];]}⊃{⍺(+/⍵)}⌸/↓[1]tbl
       A.WriteUtf8File target lines
       MyLogger.Log(⍕⍴files),' file',((1<⍴files)/'s'),' processed:'
       MyLogger.Log' ',↑files
       rc←0
   :EndIf
∇
```

Notes:

- We are now using `FilesAndDirs.Dir` rather than the Dyalog primitive `⎕NINFO`. Apart from offering recursive searches (a feature we don't need here) the `Dir` function also normalizes the separator character. Under Windows it will always be a backslash while under Linux it is always a slash character.

  Although Windows itself is quite relaxed about the separator and accepts a slash as well as a backslash, as soon as you call something else in one way or another you will find that slashes are not accepted. An example is any setting to `⎕USING`.
- We use `APLTreeUtils.WriteUtf8File` rather than `⎕NPUT` for several reasons:
  1. It will either overwrite an existing file or create a new one for us no questions asked.
  2. It will try several times in case something goes wrong. This is often helpful when a slippery network is involved.
- We could have written `A.WriteUtf8File target ({⍺,',',⍕⍵}/⊃{⍺(+/⍵)}⌸/↓[1]tbl)`, avoiding the local variable `lines`. We didn't because this variable might be helpful in case something goes wrong and we need to trace through the `TxtToCsv` function.
- Note that `MyLogger` is a global variable rather than being passed as an argument to `TxtToCsv`. We will discuss this issue in detail in the "Configuration settings" chapter.

Finally we change `Version`:

```
∇r←Version
⍝ * 1.1.0:
⍝   * Can now deal with non-existent files.
⍝   * Logging implemented.
⍝ * 1.0.0
⍝   * Runs as a stand-alone EXE and takes parameters from the command line.
  r←(⍕⎕THIS) '1.1.0' '2017-02-26'
∇
```

The foreseeable error that aborted the runtime task – an invalid filepath – has now been replaced by a message saying no files were found.

We have also changed the explicit result. So far it has returned the number of bytes written. In case something goes wrong ("file not found" etc.) it will now return `¯1`.

We can now test `TxtToCsv`:

```
      #.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:\
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
```

Alternatively you could set the parameter `printToSession` – which defaults to 0 – to 1. That would let the `Logger` class write all the messages not only to the log file but also to the session. That can be quite useful for test cases or during development. You could even prevent the `Logger` class to write to the disk at all by setting `fileFlag` to 0.

The `Logger` class is designed to never break your application – for obvious reasons. The drawback of this is that if something goes wrong like the path becoming invalid because the drive got removed you would only notice by trying to look at the log files. You can tell the `Logger` class that it should **not** trap all errors by setting the parameter `debug` to 1. Then `Logger` would crash if something goes wrong.

Let's see if logging works also for the exported EXE. Run the DYAPP to rebuild the workspace. Export as before and then run the new `MyApp.exe` in a Windows console.

```
Z:\code\v04\MyApp.exe Z:\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:\code\v04\MyApp.exe Z:\texts\de
```

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

```
      ↑⎕NGET 'Logs\MyApp_20160406.log'
2017-02-26 10:54:01 *** Log File opened
2017-02-26 10:54:01 (0) Started MyApp in Z:\code\v04
2017-02-26 10:54:01 (0) Source: G:\Does_not_exist
2017-02-26 10:54:01 (0) No files found to process
2017-02-26 10:54:26 *** Log File opened
2017-02-26 10:54:26 (0) Source: "Z:\texts\en\ageofinnocence.txt"
2017-02-26 10:54:26 (0) Started MyApp in Z:\code\v04
2017-02-26 10:54:26 (0) 1 file processed.
2017-02-26 10:58:07 (0) Z:/texts/en/ageofinnocence.txt
2017-02-26 10:54:35 *** Log File opened
2017-02-26 10:54:35 (0) Started MyApp in Z:\code\v04
2017-02-26 10:54:35 (0) Source: "Z:\texts\en\"
2017-02-26 10:54:35 (0) 9 files processed.
2017-02-26 10:58:07 (0) Z:/texts/en/ageofinnocence.txt
...
```

In case you wonder what the `(0)` in the log file stands for: this reports the thread number that has written to the log file. Since we do not use threads this is always `(0)` = the main thread the interpreter is running in.

One more improvement in `MyApp`: we change the setting of the system variables from

```
:Namespace MyApp

    (⎕IO ⎕ML ⎕WX ⎕PP ⎕DIV)←1 1 3 15 1
    ....
```

to the more readable:

```
:Namespace MyApp

    ⎕IO←1 ⋄ ⎕ML←1 ⋄ ⎕WX←3 ⋄ ⎕PP←15 ⋄ ⎕DIV←1
    ....
```

4.2 Watching the log file with LogDog

So far we have used modules from the APLTree project: classes and namespace scripts that might be useful when implementing an application.

APLTree also offers applications that support the programmer during her work without becoming part of the application. One of those applications is the LogDog. Its purpose is simply to watch a log file and make sure that any changes are immediately reflected by the GUI. This is useful for us since now the log file is the major source of information about how the application is doing.

In order to use LogDog you first need to download it from http://download.aplwiki.com. We assume that you download it into the default download location. For a user "JohnDoe" that would be `C:\Users\JohnDoe\Downloads`.

LogDog does not come with an installer. All you have to do is to copy it into a folder where you have the right to add, delete and change files. That means `C:\Program Files` and `C:\Program Files (x86)` are not an option. If you want to install the application just for your own user ID then this is the right place:

```
"C:\Users\JohnDoe\AppData\Local\Programs\LogDog
```

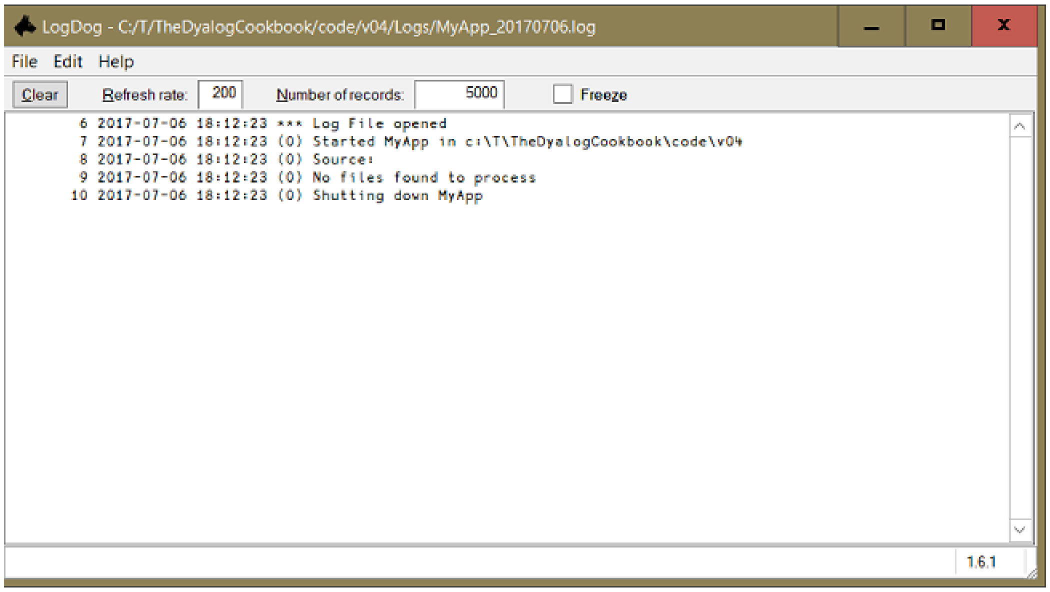
If you want to install it for all users on your PC then we suggest that you create this folder:

```
"C:\Users\All users\Local\Programs\LogDog
```

Of course `C:\MyPrograms\LogDog` might be okay as well.

You start LogDog by double-clicking the EXE. You can then consult LogDog's help for how to open a log file. We recommend to go for the "Investigate folder" option. The reason is that every night at 24:00 a new log file with a new name is created. To put any new(er) log file on display you can issue the "Investigate folder" menu command again.

Once you have started LogDog on the `MyApp` log file you will see something like this:



```
 6 2017-07-06 18:12:23 *** Log File opened
 7 2017-07-06 18:12:23 (0) Started MyApp in c:\T\TheDyalogCookbook\code\v04
 8 2017-07-06 18:12:23 (0) Source:
 9 2017-07-06 18:12:23 (0) No files found to process
10 2017-07-06 18:12:23 (0) Shutting down MyApp
```

LogDog GUI

Note that LogDog comes with an auto-scroll feature, meaning that the latest entries at the bottom of the file are always visible. If you don't want this for any reason just tick the "Freeze" check box.

From now on we will assume that you have LogDog always up and running, so that you will get immediate feedback on what is going on when `MyApp.exe` runs.

4.3 Where are we

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. But before we are doing this we will disuss how to configure `MyApp`.

Destructors versus the Tracer

When you trace through `TxtToCsv` the moment you leave the function the Tracer shows the function `Cleanup` of the `Logger` class. The function is declared as a destructor.

In case you wonder why that is: a destructor (if any) is called when the instance of a class is destroyed (or shortly thereafter). `MyLogger` is localized in the header of `TxtToCsv`, meaning that when `TxtToCsv` ends, this instance of the `Logger` class is destroyed and the destructor is invoked. Since the Tracer was up and running, the destructor makes an appearance in the Tracer.

5. Configuration settings

We are going to want our logging and error handling to be configurable. In fact, we will soon have lots of state settings; thinking more widely, an application's configuration includes all kinds of state: e.g., folders for log files and crashes, a debug flag, a flag for switching off error trapping, an email address to report to, you name it.

A variety of mechanisms for permanently storing configuration settings exists: Under Microsoft Windows we have the Windows Registry, and there are a number of cross-platform file formats to consider: XML, JSON - and good old INI files. We will discuss these in detail.

5.1 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; some would say abused. However, for quite some time it was considered bad to have application-specific config files. Everything was expected to go into the Windows Registry. The pendulum started to swing back the other way now for several years, and we see application-specific config files becoming ever more common. 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 classes WinRegSimple and WinReg provide methods for handling the Windows Registry. We will discuss them in their own chapter.

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

Note that the Windows Registry is still an excellent choice for saving user-specific stuff like preferences, themes, recent files etc. However, you have to make sure that your user has permission to write to the Windows Registry - that's by no means a certainty.

5.2 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.

By using the APLTree class `IniFiles` we get as a bonus additional features:

- Data types: a key can carry either a text vector or a number.
- Nested vectors: a key can carry a vector of text vectors.
- Merge INI files: specify more than one INI file.
- Local variables (place holders).

We will discuss all these features as we go along.

5.3 INI files it is!

Where to save an INI file

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, for all users.
These must be coded into the application ("Convention over configuration"), 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.

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

As part of the user's profile
The Windows environment variable `APPDATA` points to the individual user's roaming profile, so we might look there for a `MyApp\MyApp.ini` file. "Roaming" means that no matter which computer a user logs on to in a Windows Domain [1] her personal settings, preferences, desktop etc. roams with her. The Windows environment variable `LOCALAPPDATA` on the other hand defines a folder that is saved just locally. Typically `APPDAATA` points to something like `C:\Users\{username}\AppData\Roaming` and `LOCALAPPDATA` to `C:\Users\{username}\AppData\Local`.

Note that when a user logs on to another computer all the files in `APPDATA` are syncronized first. Therefore it is not too good an idea to save a logfile in `APPDATA` that will eventually grow large – that should go into `LOCALAPPDATA`.

[1]https://en.wikipedia.org/wiki/Windows_domain

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

1. Defaults in the program code
2. Overwrite from ALLUSERSPROFILE if any
3. Overwrite from USERPROFILE
4. Overwrite from an INI specified in command line, if any
5. Overwrite with the command line

However, for the Cookbook we keep things simple: we look for an INI file that is a sibling of the DYAPP or the EXE for now but will allow this to be overwritten via the command line with something like `INI='C:\MyAppService\MyApp.ini`. We need this when we make MyApp a Windows Scheduled Task, or run it as a Windows Service.

Let's start

Save a copy of `Z:\code\v04` as `Z:\code\v05` or copy `v05` from the Cookbook's website. We add one line to `MyApp.dyapp`:

```
...
Load ..\AplTree\FilesAndDirs
leanpub-insert-start
Load ..\AplTree\IniFiles
leanpub-insert-end
Load ..\AplTree\OS
...
```

and run the DYAPP to recreate the `MyApp` workspace.

You can read `IniFiles`'s documentation in a browser with `]ADoc #.IniFiles`.

Our INI file

This is the contents of the newly introduced `code\v05\MyApp.ini`:

```
localhome = '%LOCALAPPDATA%\MyApp'

[Config]
Debug       = ¯1    ; 0=enfore error trapping; 1=prevent error trapping;
Trap        = 1     ; 0 disables any :Trap statements (local traps)

Accents     = ''
Accents     ,='ÁÂÃÀÄÅÇĐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ'
Accents     ,='AAAAAACDEEEEIIIINOOOOOOUUUUY'

[Folders]
Logs        = '{localhome}\Log'
Errors      = '{localhome}\Errors'
```

If you have not copied `v05` from the website make sure you create an INI file with this contents as a sibling of the DYAPP.

Notes:

- The `IniFiles` class offers some unique features. Those are discussed below. This is by no means a violation of the standard because for INI files there is no such thing.
- Assignments above the first section – which is `[Config]` – are variables local to the INI file. We can refer to them by putting curly brackets (`{}`) around their names as with `{localhome}` but they have no other purpose. You can see that `localhome` is referred to twice in the `[Folders]` section, and why that is useful.
- `IniFiles` supports two data types: character and number. Everything between two quotes is character, everything else is assumed to be a number.
- `Debug` is set to ¯1 – it is indeed going to be a numeric value because there are no quotes involved. `debug` defines whether the application runs in debug mode or not. Most importantly `debug←1` will switch off global error trapping, something we will soon introduce. `¯1` means that the INI file does not set the flag. Therefore it will later in the application default to 1 in a development environment and to 0 in a runtime evenvironment. By setting this to either 1 or 0 in the INI file you can force it to be a particular value.
- `Trap` can be used to switch off error trapping globally. It will be used in statements like `:Trap Config.Traps/0`. What `Config` is we will discuss in a minute.
- `Accents` is initialized as an empty vector but then values are added with `,=`. That means that `Accents` will be a vtv: a vector of text vectors. Since we define the default to be the same as what the INI file contains anyway it makes not too much sense but it illustrates a second – better?! – way of defining it.
- `Logs` defines the folder were MyApp will save its log files.
- `Errors` defines the folder were MyApp will save crash information later on when we establish global error handling.

Initialising the workspace

We create a new function `CreateConfig` for that:

```
∇ Config←CreateConfig dummy;myIni;iniFilename
⍝ Instantiate the INI file and copy values over to a namespace `Config`.
  Config←⎕NS''
  Config.⎕FX'r←∆List' 'r←{0∊ρω:0 2ρ'''' ⋄ ω,[1.5]⍎¨ω}'' ''~¨⍨↓⎕NL 2'
  Config.Debug←A.IsDevelopment
  Config.Trap←1
  Config.Accents←'ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ' 'AAAAAACDEEEEIIIINOOOOOOUUUUY'
  Config.LogFolder←'./Logs'
  Config.DumpFolder←'./Errors'
  iniFilename←'expand'F.NormalizePath'MyApp.ini'
  :If F.Exists iniFilename
      myIni←⎕NEW ##.IniFiles(,⊂iniFilename)
      Config.Debug{¯1≡ω:α ⋄ ω}←myIni.Get'Config:debug'
      Config.Trap←⊃Config.Trap myIni.Get'Config:trap'
      Config.Accents←⊃Config.Accents myIni.Get'Config:Accents'
      Config.LogFolder←'expand'F.NormalizePath⊃Config.LogFolder myIni.Get'Folders:Logs'
      Config.DumpFolder←'expand'F.NormalizePath⊃Config.DumpFolder myIni.Get'Folders:Errors'
  :EndIf
  Config.LogFolder←'expand'F.NormalizePath Config.LogFolder
  Config.DumpFolder←'expand'F.NormalizePath Config.DumpFolder
∇
```

What the function does:

- It creates an unnamed namespace and assigns it to `Config`.
- It fixes a function `∆List` inside `Config`.
- It populates `Config` with the defaults for all the settings we are going to use. Remember, we might not find an INI file.
- It creates a name for the INI file and checks whether it exists. If that's the case then it instatiates the INI file and then copies all values it can find from the INI file to `Config`, overwriting the defaults.

Notes:

- The `Get` function requires a section and a key as right argument. They can be provided either as a two-item vector as in `'Config' 'debug'` or as a text vector with section and key separated by a colon as in `'Config:debug'`.
- `Get` requires a given section to exist, otherwise it will throw an error. It is tolerant in case a given key does not exist in case a left argument is provided: in that case the left argument is considered a default and returned by `Get`. In case the key does not exist *and* no left argument was specified an error is thrown.
- In case you cannot be sure whether a section/key combination exists (a typical problem when after an update a newer version of an application hits an old INI file) you can check with the `Exist` method.

The built-in function `∆List` comes handy when you want to check the contents of `Config`:

```
      Config.∆List
 Accents      ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ  AAAAAACDEEEEIIIINOOOOOOUUUUY
 Debug                                                               0
 DumpFolder                         C:\Users\kai\AppData\Local\MyApp\Log
 LogFolder                          C:\Users\kai\AppData\Local\MyApp\Log
 Trap                                                                1
```

Now that we have moved `Accents` to the INI file we can get rid of these lines in the `MyApp` script:

```
⍝ === VARIABLES ===
    Accents←'ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ' 'AAAAAACDEEEEIIIINOOOOOOUUUUY'
⍝ === End of variables definition ===
```

Where should we call `CreateConfig` from? Surely that has to be `Initial`:

```
∇ (Config MyLogger)←Initial dummy
⍝ Prepares the application.
  #.⎕IO←1 ⋄ #.⎕ML←1 ⋄ #.⎕WX←3 ⋄ #.⎕PP←15 ⋄ #.⎕DIV←1
  Config←CreateConfig ⍬
  MyLogger←OpenLogFile Config.LogFolder
  MyLogger.Log'Started MyApp in ',F.PWD
  MyLogger.Log #.GetCommandLine
  MyLogger.Log↓⎕FMT Config.∆List
∇
```

Note that we also changed what `Initial` returns: a vector of length two, the namespace `Config` but also an instance of the `MyLogger` class.

`Initial` was called within `StartFromCmdLine`, and we are not going to change this but we must change the call as such because now it returns something useful:

```
∇ {r}←StartFromCmdLine arg;MyLogger;Config
⍝ Needs command line parameters, runs the application.
  r←⍬
  (Config MyLogger)←Initial ⍬
  r←TxtToCsv arg~''''
∇
```

Although both `MyLogger` as well as `Config` are kind of global and not passed as arguments it helps to assign them this way rather then hide the statement that creates them somewhere down the stack. This way it's easy to see where they are coming from.

Specifying an INI file on the command line

We could pass the command line parameters as arguments to `Initial` and investigate whether it carries any `INI=` statement. If so the INI file specified this way should take precedence over any other INI file. However, we keep it simple here.

We now need to think about how to access `Config` from within `TxtToCsv`.

What we think about when we think about encapsulating state

The configuration parameters, including `Accents`, are now collected in the namespace `Config`. That namespace is not passed explicitly to `TxtToCsv` but is needed by `CountLetters` which is called by `TxtToCsv`. We have two options here: we can pass a reference to `Config` to `TxtToCsv`, for example as left argument, and `TxtToCsv` in turn can pass it to `CountLetters`. The other option is that `CountLetters` just assumes the `Config` is around and has a variable `Accents` in it:

```
CountLetters←{
    {⍺(≢⍵)}⌸⎕A{⍵⌿⍨⍵∊⍺}Config.Accents U.map A.Uppercase ⍵
}
```

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

Passing everything through function arguments does not come with a performance penalty. 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 `G` as a left argument of `TxtToCsv`, which then simply gets passed to `CountLetters`. 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) don't want to specify when working in session mode.

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 global variables*. (Or semi-global variables. [2]) 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.

One of the main reasons why globals should be used with great care is that they can easily be confused with local variables with similar or – worse – the same name. If you need to have global variables then we suggest to encapsulating them in a dedicated namespace `Globals`. With a proper search tool like Fire [3] it is easy to get a report on all lines referring to anything in `Globals`, or set it.

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.

The IniFiles class

We have used the most important features of the `IniFiles` class, but it has more to offer. We just want to mention some major topics here.

[2]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'.

[3]Fire stands for *Find and Replace*. It is a powerful tool for both search and replace operations in the workspace. It is also a member of the APLTree Open Source Library. For details see http://aplwiki.com/Fire. Fire is discussed in the chapter "Useful user commands".

- The `Get` method can be used to list sections of even all sections with all key-value pairs. The following can be done when you trace into the `Initial` function to the point where the instance of the `Logger` class got instantiated:

```
      myIni.Get 'Config' ⍬
 Debug                                                               0
 Trap                                                                1
 Accents   ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ  AAAAAACDEEEEIIIINOOOOOOUUUUY
      Display myIni.Get_ ⍬ ⍬
 CONFIG
        Debug                                                              ¯1
        Trap                                                                1
        Accents   ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ  AAAAAACDEEEEIIIINOOOOOOUUUUY
 FOLDERS
        Logs                                     %LOCALAPPDATA%\MyApp\Log
        Errors                                   %LOCALAPPDATA%\MyApp\Log
```

  `Get` returns a matrix with three columns:
  1. Contains per row a section name or an empty vector
  2. Contains a key or an empty vector
  3. Contains either a value or an empty vector.
- Instead of using the `Get` method you can also use indexing:

```
      myIni[⊂'Config:debug']
0
      myIni['Config:debug' 'Folders:']
 0   %LOCALAPPDATA%\MyApp\Log  %LOCALAPPDATA%\MyApp\Log
```

- You can actually assign a value to a key with the index syntax and save the INI file by calling the `Save` method. However, you should *only* use this to write default values to an INI file, typically in order to create one. An INI file is not a database and should not be abused as such.
- We instantiated the `IniFiles` class with the statement `myIni←⎕NEW ##.IniFiles(,⊂iniFilename)` but you can actually specify more than just one INI file. Let's assume that your computer's name is "Foo" then this:

```
myIni←⎕NEW ##.IniFiles('MyApp.ini' 'Foo.ini')
```

  would create a new instance which contains all the definitions of *both* INI files. In case of a name conflict the last one wins. Here this would mean that machine specific definitions would overwrite more general ones.
- Sometimes it is more appropriate to have a namespace representing the INI file as such, with sub namespaces representing the sections and variables within them representing the keys and values. This can be achieved by using the instance method `Convert`. See `]ADoc #.IniFiles` for details.

  Here we give a simple example:

```
      q←myIni.Convert ⎕ns''
      q.List ''
CONFIG   Accents   ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ  AAAAAACDEEEEIIIINOOOOOOUUUUY
CONFIG   Debug                                                               ¯1
CONFIG   Trap                                                                 1
FOLDERS  Errors                                        %LOCALAPPDATA%\MyApp\Log
FOLDERS  Logs                                          %LOCALAPPDATA%\MyApp\Log
       q.RIDE.Debug
¯1
```

Final steps

We need to change the `Version` function:

```
∇ r←Version
   ⍝ * 1.2.0:
   ⍝   * The application now honours INI files.
   ⍝ * 1.1.0:
   ⍝   * Can now deal with non-existent files.
   ⍝   * Logging implemented.
   ⍝ * 1.0.0
   ⍝   * Runs as a stand-alone EXE and takes parameters from the command line.
      r←(⍕⎕THIS)'1.2.0' '2017-02-26'
∇
```

And finally we create a new standalone EXE as before and run it to make sure that everything keeps working. (Yes, we need test cases)

6. Debugging a stand-alone EXE

Imagine the following situation: when MyApp is started with a double-click on the DYAPP and then tested everything works just fine. When you create a stand-alone EXE from the DYAPP and execute it with some appropriate parameter it does not create the CSV files. In this situation obviously you need to debug the EXE. In this chapter we'll discuss how to achieve that. In addition we will make `MyApp.exe` return an exit code.

For debugging we are going to use Ride. (If you don't know what Ride is refer to the documentation) If enabled you can use Ride to hook into a running interpreter, interrupt any running code, investigate and even change that code.

6.1 Configuration settings

We introduce a `[RIDE]` section into the INI file:

```
[Ride]
Active      = 1
Port        = 4599
Wait        = 1
```

By setting `Active` to 1 and defining a `Port` number for the communication between Ride and the EXE you can tell MyApp that you want "to give it a ride". Setting `Wait` to 1 lets the application wait for a Ride. That simply means it enters an endless loop.

That's not always appropriate of course, because it allows anybody to read your code. If that's something you have to avoid then you have to find other ways to make the EXE communicate with Ride, most likely by making temporary changes to the code. The approach would be in both cases the same. In MyApp we keep things simple and allow the INI file to rule whether the user may Ride into the application or not.

Copy `Z:\code\v05` to `Z:\code\v06` and then run the DYAPP to recreate the `MyApp` workspace.

Note that 4502 is Ride's default port, and that we've settled for a different port, and for good reasons. Using the default port leaves room for mistakes. Using a dedicated port rather than just using the default minimises the risk of connecting with the wrong application.

6.2 The "Console application" flag

In case you've exported the EXE with the "console application" check box ticked there is a problem: although you will be able to connect to the EXE with Ride, all output goes into the console window. That means that you can enter statements in Ride but any response from the interpreter goes to the console window rather than Ride.

For debugging purposes it is therefore recommended to create the EXE with the check box unticked.

6.3 Code changes

We want to make the Ride configurable. That means we cannot do it earlier than after having instantiated the INI file. But not long after either, so we change `Initial`:

```
∇ (Config MyLogger)←Initial dummy
⍝ Prepares the application.
  #.⎕IO←1 ⋄ #.⎕ML←1 ⋄ #.⎕WX←3 ⋄ #.⎕PP←15 ⋄ #.⎕DIV←1
  Config←CreateConfig ⍬
  CheckForRide Config.(Ride WaitForRide)
  MyLogger←OpenLogFile Config.LogFolder
  MyLogger.Log'Started MyApp in ',F.PWD
  MyLogger.Log #.GetCommandLine
  MyLogger.Log↓⎕FMT Config.∆List
∇
```

We have to make sure that `Ride` makes it into `Config`, so we establish a default 0 (no Ride) and overwrite with INI settings.

```
∇ Config←CreateConfig dummy;myIni;iniFilename
  Config←⎕NS''
  Config.⎕FX'r←∆List' 'r←{0∊⍴⍵:0 2⍴'''' ⋄ ⍵,[1.5]⍎¨⍵}'' ''~¨⍨↓⎕NL 2'
  Config.Debug←A.IsDevelopment
  Config.Trap←1
  Config.Accents←'ÁÂÃÀÄÅÇÐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ' 'AAAAAACDEEEEIIIINOOOOOOUUUUY'
  Config.LogFolder←'./Logs'
  Config.DumpFolder←'./Errors'
  Config.Ride←0         ⍝ If not 0 the app accepts a Ride & treats Config.Ride as port number.
  Config.WaitForRide←0 ⍝ If 1 `CheckForRide` will enter an endless loop.
  iniFilename←'expand'F.NormalizePath'MyApp.ini'
  :If F.Exists iniFilename
      myIni←⎕NEW ##.IniFiles(,⊂iniFilename)
      Config.Debug{¯1≡⍵:⍺ ⋄ ⍵}←myIni.Get'Config:debug'
      Config.Trap←⊃Config.Trap myIni.Get'Config:trap'
      Config.Accents←⊃Config.Accents myIni.Get'Config:Accents'
      Config.LogFolder←'expand'F.NormalizePath⊃Config.LogFolder myIni.Get'Folders:Logs'
      Config.DumpFolder←'expand'F.NormalizePath⊃Config.DumpFolder myIni.Get'Folders:Errors'
      :If myIni.Exist'Ride'
      :AndIf myIni.Get'Ride:Active'
          Config.Ride←⊃Config.Ride myIni.Get'Ride:Port'
          Config.WaitForRide←⊃Config.Ride myIni.Get'Ride:Wait'
      :EndIf
  :EndIf
  Config.LogFolder←'expand'F.NormalizePath Config.LogFolder
  Config.DumpFolder←'expand'F.NormalizePath Config.DumpFolder
∇
```

We add a function `CheckForRide`:

```
∇ {r}←CheckForRide (ridePort waitFlag);rc
 ⍝ Depending on what's provided as right argument we prepare for a Ride
 ⍝ or we don't. In case `waitFlag` is 1 we enter an endless loop.
  r←1
  :If 0<ridePort
      rc←3502⌶0
      :If ~rc∊0 ¯1
          11 ⎕SIGNAL⍨'Problem switching off Ride, rc=',⍕rc
      :EndIf
      rc←3502⌶'SERVE::',⍕ridePort
      :If 0≠rc
          msg←'Problem setting the Ride connecion string to SERVE::'
          msg,←,(⍕ridePort),', rc=',⍕rc
          11 ⎕SIGNAL⍨msg
      :EndIf
      rc←3502⌶1
      :If ~rc∊0 ¯1
          11 ⎕SIGNAL⍨'Problem switching on Ride, rc=',⍕rc
      :EndIf
      {}{_←⎕DL ⍵ ⋄ ∇ ⍵}⍣(⊃waitFlag)⊣1
  :EndIf
∇
```

Notes:

- `ridePort` will be either the port to be used for communicating with Ride or 0 if no Ride is required.
- The optional left argument defaults to 0. If it is 1 then the function waits for Ride to hook on.
- In this specific case we pass a reference to `Config` as argument to `CheckForRide`. There are two reasons for that:
  - `CheckForRide` really needs `Config`.
  - We have nothing else to pass but we don't want to have niladic functions around (except in very special circumstances).
- We catch the return codes from the calls to `3502⌶` and check them on the next line. This is important because the calls may fail for several reasons; see below for an example. In case something goes wrong the function signals an error.
- With `3502⌶0` we switch Ride off, just in case. That way we make sure that we can execute `→1` while tracing `CheckForRide` at any point if we wish to; see "Restartable functions" underneath this list.
- With `3502⌶'SERVE::',⍕ridePort` we establish the Ride parameters: type ("SERVE"), host (nothing between the two colons, so it defaults to "localhost") and port number.
- With `3502⌶1` we enable Ride.
- With `{_←⎕DL ⍵ ⋄ ∇ ⍵}1` we start an endless loop: wait for a second, then calls itself (∇) recursively. It's a dfn, so there is no stack growing on recursive calls.
- We could have passed `Config` rather than `Config.(Ride WaitForRide)` to `CheckForRide`. By *not* doing this we allow the function `CheckForRide` to be tested independently from `Config`. This is an important point. There is value in keeping the function independent in this way, but if you suspect that later you will most likely be in need for other parameters in `Config` then the flexibility you gain this way outperforms the value of keeping the function independent from `Config`.

Finally we amend the `Version` function:

```
∇r←Version
   ⍝ * 1.3.0:
   ⍝   * MyApp gives a Ride now, INI settings permitted.
   ...
∇
```

Before we can actually try to Ride into

Now you can start Ride, enter "localhost" and the port number as parameters, connect to the interpreter or stand-alone EXE etc. and then select "Strong interrupt" from the "Actions" menu in order to interrupt the endless loop; you can then start debugging the application. Note that this does not require the development EXE to be involved: it may well be a runtime EXE. However, of course you need a development license in order to be legally entitled to Ride into an application run by the RunTime EXE (DyalogRT.exe).

DLLs required by Ride

Prior to version 16.0 one had to copy the files "ride27_64.dll" (or "ride27_32.dll") and "ride27ssl64.dll" (or "ride27ssl32.dll") so that they are siblings of the EXE. From 16.0 onward you must copy the Conga DLLs instead. Failure in doing that will make `3502⌶1` fail. Note that "2.7" refers to the version of Conga, not Ride. Prior to version 3.0 of Conga every application (interpreter, Ride, etc.) needed to have their own copy of the Conga DLLs, with a different name. Since 3.0 Conga can serve several applications in parallel. We suggest that you copy the 32-bit and the 64-bit DLLs over to where your EXE lives.

In case you forgot to copy "ride27ssl64.dll" and/or "ride27ssl32.dll" then you will see an error "Can't find Conga DLL". This is because the OS does not bother to tell you about dependencies. You need a tool like DependencyWalker for finding out what's really missing. Note that we said "OS" because this is *not* a Windows-only problem.

Restartable functions

Not only do we try to exit functions at the bottom, we also like them to be "restartable". What we mean by that is that we want a function – and its variables – to survive `→1`whenever that is possible; sometimes it is not like a function that starts a thread and *must not* start a second one for the same task, or a file was tied etc. but most of the time it is possible to achieve that. That means that something like this should be avoided:

```
∇r←MyFns arg
r←⍬
:Repeat
    r,← DoSomethingSensible ⊃arg
:Until 0∊⍴arg←1↓arg
```

This function does not make much sense but the point is that the right argument is mutilated; one cannot restart this function with `→1`. Don't do something like that unless there are very good reasons. In this example a counter is a better way to do this. It’s also faster.

7. Handling errors

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

7.1 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.

We'll start with the second item from the list above: quitting and passing an exit code.

Inspecting Windows exit codes

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

```
Z:\code\v05\MyApp.exe Z:\texts\en

echo Exit Code is %errorlevel%
Exit Code is 0

MyApp.exe Z:\texts\does_not_exist

echo Exit Code is %errorlevel%
Exit Code is 101
```

but only if you ticked the check box "Console application" in the "Export" dialog box. We don't want to do this if we can help it because we cannot Ride into an application with this option active. Therefore we are going to execute our stand-alone EXE from now on with the help of the APLTree class `Execute`.

Copy `Z:\code\06` to `Z:\code\07`.

For the implementation of global error handling we need APLTree's `HandleError` class. For calling the exported EXE we need the `Execute` class. Therefore we add both to the DYAPP. Edit `Z:\code\v07\MyApp.dyapp`:

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

7.2 Foreseen errors

For those we check in the code and quit when something is wrong but pass an error code to the calling environment.

First we define in `#.MyApp` a child namespace of exit codes:

```
    :Namespace EXIT
        OK←0
        INVALID_SOURCE←111
        SOURCE_NOT_FOUND←112
        UNABLE_TO_READ_SOURCE←113
        UNABLE_TO_WRITE_TARGET←114
          GetName←{
              l←' '~¨⍨↓⎕NL 2
              ind←({⍎¨l}l)⍳⍵
              ind⊃l,⊂'Unknown error'
          }
    :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 can convert the numeric value back to the symbolic name with the function `GetName`:

```
      EXIT.GetName EXIT.INVALID_SOURCE
INVALID_SOURCE
```

This is useful when we want to log an error code: the name is telling while the number is meaningless.

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. The exit codes defined in `EXIT` are specific to `MyApp`, so are better defined there.

Now the result of `TxtToCsv` gets passed to `⎕OFF` to be returned to the operating system as an exit code.

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

In case you wonder about `⎕OFF`: that's actually a niladic function. Being able to provide a "right argument" is therefore kind of cheating because there can't be any. This is a special case in the Dyalog parser.

Note that in this particular case we invent a local variable `rc` although strictly speaking this is not necessary. We learned from experience that you should not call several functions on a single line with the left-most being `Off`. If you do this anyway you will regret it one day, it's just a matter of time.

Now we have to introduce a function `Off`:

```
∇ Off exitCode
  :If 0<⎕NC'MyLogger'
      :If exitCode=EXIT.OK
          MyLogger.Log'MyApp is closing down gracefully'
      :Else
          MyLogger.LogError exitCode('MyApp is unexpectedly shutting down: ',EXIT.GetName exitCode)
      :EndIf
  :EndIf
  :If A.IsDevelopment
      →
  :Else
      ⎕OFF exitCode
  :EndIf
∇
```

Note that `⎕OFF` is actually only executed when the program detects a runtime environment, otherwise it just quits. Although the workspace is much less important these days you still don't want to loose it by accident.

We modify `GetFiles` so that it checks its arguments and the intermediary results:

```
∇ (rc target files)←GetFiles fullfilepath;csv;target;path;stem;isDir
⍝ Checks argument and returns a list of files (or a single file).
   fullfilepath~←'"'
   files←target←''
   :If 0∊⍴fullfilepath
       rc←EXIT.INVALID_SOURCE
       :Return
   :EndIf
   csv←'.csv'
   :If 0=F.Exists fullfilepath
       rc←EXIT.SOURCE_NOT_FOUND
   :ElseIf ~isDir←F.IsDir fullfilepath
   :AndIf ~F.IsFile fullfilepath
       rc←EXIT.INVALID_SOURCE
   :Else
       :If isDir
           target←F.NormalizePath fullfilepath,'\total',csv
           files←⊃F.Dir fullfilepath,'/*.txt'
       :Else
           (path stem)←2↑⎕NPARTS fullfilepath
           target←path,stem,csv
           files←,⊂fullfilepath
       :EndIf
       target←(~0∊⍴files)/target
       rc←EXIT.OK
   :EndIf
∇
```

Note that we have replaced some constants by calls to functions in `FilesAndDirs`. You might find this easier to read.

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, and we have learned a great respect for our capacity to get confused. However, here we don't mind exiting the function with `:Return` on line 5. It's obvious why that is and it saves us one level of nesting regarding the control structures. Also, there is no tidying up at the end of the function that we would miss with `:Return`.

`ProcessFile` now traps some errors:

```
∇ data←(fns ProcessFiles)files;txt;file
 Reads all files and executes `fns` on the contents.
  data←⍬
  :For file :In files
      :Trap Config.Trap/FileRelatedErrorCodes
          txt←'flat'A.ReadUtf8File file
      :Case
          MyLogger.LogError'Unable to read source: ',file
          Off EXIT.UNABLE_TO_READ_SOURCE
      :EndTrap
      data,←⊂fns txt
  :EndFor
∇
```

In the line with the `:Trap` we call a niladic function (exception to the rule!) which returns all error codes that are related to problems with files:

```
∇ r←FileRelatedErrorCodes
⍝ Returns all the error codes that are related to files and directories.
⍝ Useful to trap all those errors.
  r←12 18 20 21 22 23 24 25 26 28 30 31 32 34 35
∇
```

Doesn't that contradict our policy to avoid meaningless constants in the code? It does indeed. Let's fix this. There is a class `EventCodes` available on the APLTree that contains symbolic names for all these error numbers. The symbolic names are taken from the help page you get when you press `F1` on `⎕TRAP`. Add this class to your DYAPP file:

```
...
Load ..\AplTree\Logger
Load ..\AplTree\EventCodes.dyalog
Load Constants
Load Utilities
Load MyApp
Run #.MyApp.SetLX ⍬
```

The `EventCodes` class comes with a method `GetName` that, when fed with an integer, returns the corresponding symbolic name. We can use that to convert return codes to meaningful names:

```
      #.EventCodes.GetName¨ #.MyApp.FileRelatedErrorCodes
HOLD_ERROR  FILE_TIE_ERROR  FILE_INDEX_ERROR  FILE_FULL  FILE_NAME_ERROR...
```

We can convert this into something that will be useful when we change the function `FileRelatedError-Codes`:

```
      ⍪'r,←E.'∘,¨#.EventCodes.GetName¨#.MyApp.FileRelatedErrorCodes
 r,←E.HOLD_ERROR
 r,←E.FILE_TIE_ERROR
 r,←E.FILE_INDEX_ERROR
 r,←E.FILE_FULL
 ...
```

Now we can change `FileRelatedErrorCodes` by copying what we've just printed to the session into the function:

```
∇ r←FileRelatedErrorCodes;E
⍝ Returns all the error codes that are related to files and directories.
⍝ Useful to trap all those errors.
  r←''
  E←##.EventCodes
  r,←E.HOLD_ERROR
  r,←E.FILE_TIE_ERROR
  r,←E.FILE_INDEX_ERROR
  r,←E.FILE_FULL
  r,←E.FILE_NAME_ERROR
  r,←E.FILE_DAMAGED
  r,←E.FILE_TIED
  r,←E.FILE_TIED_REMOTELY
  r,←E.FILE_SYSTEM_ERROR
  r,←E.FILE_SYSTEM_NOT_AVAILABLE
  r,←E.FILE_SYSTEM_TIES_USED_UP
  r,←E.FILE_TIE_QUOTA_USED_UP
  r,←E.FILE_NAME_QUOTA_USED_UP
  r,←E.FILE_SYSTEM_NO_SPACE
  r,←E.FILE_ACCESS_ERROR_CONVERTING_FILE
∇
```

Why don't we just :Trap all errors?

`:Trap 0` would trap all errors - way easier to read and write, so why don't we do this?

Well, for a very good reason: trapping everything includes such basic things like a VALUE ERROR, which is most likely introduced by a typo or by removing a function or an operator in the false believe that it is not called anywhere. We don't want to trap those, really. The sooner they come to light the better. For that reason we restrict the errors to be trapped to whatever might pop up when it comes to dealing with files and directories.

That being said, if you really have to trap *all* errors (occasionally this makes sense) then make sure that you can switch it off with a global flag as in `:Trap trapFlag/0`: if `trapFlag` is 1 then the trap is active, otherwise it is not.

It's a different story when you try to catch any possible error at a very early stage of an application. That scenario will be discussed soon.

Back to `ProcessFiles`. Note that 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 to reset the trap to avoid an open loop. But be careful when you call other functions: in case they crash the `:Trap` would catch the error!

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.

Note that here for the first time we take advantage of the `[Config]Trap` flag defined in the INI file, which translates to `Config.Trap` at this stage. With this flag we can switch off all “local” error trapping, sometimes a measure we need to take to get to the bottom of a problem.

Finally we need to amend `TxtToCsv`:

```
∇ exit←TxtToCsv fullfilepath;∆;isDev;Log;LogError;files;target;success
 ⍝ Write a sibling CSV of the TXT located at fullfilepath,
 ⍝ containing a frequency count of the letters in the file text
 ⍝ Returns one of the values defined in `EXIT`.
  (rc target files)←GetFiles fullfilepath
  :If rc=EXIT.OK
      :If 0∊⍴files
          MyLogger.Log'No files found to process'
      :Else
          tbl←⊃⍪/(CountLetters ProcessFiles)files
          lines←{⍺,',',⍕⍵}/{⍵[⍒⍵[;2];]}⊃{⍺(+/⍵)}⌸/↓[1]tbl
          :Trap Config.Trap/FileRelatedErrorCodes
              A.WriteUtf8File target lines
              success←1
          :Case
              MyLogger.LogError'Writing to <',target,'> failed, rc=',(⍕⎕EN),'; ',⊃⎕DMX
              rc←EXIT.UNABLE_TO_WRITE_TARGET
              success←0
          :EndTrap
          :If success
              MyLogger.Log(⍕⍴files),' file',((1<⍴files)/'s'),' processed:'
              MyLogger.Log' ',↑files
          :EndIf
      :EndIf
  :EndIf
∇
```

Note that 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.

Logging file related errors

Logging errors related to files in a real-world application requires more attention to detail: `⎕DMX` provides more information that can be very useful:

- `Message`
- `OSErrors`
- `InternalLocation`

7.3 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 replicable, we can easily track it down using the development interpreter. But the error might not be 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 any `:Trap` statements. 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.

Define a new `EXIT` code constant:

```
    ....
    OK←0
    APPLICATION_CRASHED←104
    INVALID_SOURCE←111
    ...
```

104? Why not 4, the standard Windows code for a crashed application? The distinction is useful. An exit code of 104 will tell us MyApp's trap caught and reported the crash. An exit code of 4 tells you even the trap failed!

We want to establish general error trapping as soon as possible, but we also need to know where to save crash files etc. That means we start right after having instantiated the INI file, because that's where we get this kind of information from. For establishing error trapping we need to set `⎕TRAP`. Because we want to make sure that any function down the stack can pass a certain error up to the next definition of `⎕TRAP` (see the `⎕TRAP` help, options "C" and "N") it is vitally important not only set to set but also to *localyze* `⎕TRAP` in `StartFromCmdLine`

```
∇ {r}←StartFromCmdLine arg;MyLogger;Config;rc;⎕TRAP
⍝ Needs command line parameters, runs the application.
   r←⍬
   (Config MyLogger)←Initial ⍬
   ⎕WSID←'MyApp'
   ⎕TRAP←(Config.Debug=0) SetTrap Config
   rc←TxtToCsv arg~''''
   Off rc
∇
```

We need to set `⎕WSID` because the global trap will attempt to save a workspace in case of a crash. We set `⎕TRAP` by assigning the result of `SetTrap`, so we we need to create that function now:

```
∇ trap←{force}SetTrap Config
⍝ Returns a nested array that can be assigned to `⎕TRAP`.
  force←{0<⎕NC ⍵:⍎⍵ ⋄ 0}'force'
  #.ErrorParms←##.HandleError.CreateParms
  #.ErrorParms.errorFolder←⊃Config.Get'Folders:Errors'
  #.ErrorParms.returnCode←EXIT.APPLICATION_CRASHED
  #.ErrorParms.logFunction←MyLogger.Log
  #.ErrorParms.windowsEventSource←'MyApp'
  #.ErrorParms.addToMsg←' --- Something went terribly wrong'
  trap←force ##.HandleError.SetTrap '#.ErrorParms'
∇
```

Notes:

- First we generate a parameter space with default values by calling `HandleError.CreateParms`.
- We then overwrite some of the defaults:
  - Where to save crash information.
  - The return code.
  - What function to use for logging purposes.
  - Name of the source to be used when reporting the problem to the Windows Event Log (empty=no reporting at all).
  - Additional message to be added to the report send to the Windows Event Log. If you don't know anything about the Windows Event Log yet: it's going to be discussed in its own chapter.
- We specify `ErrorParms` as a global named namespace for two reasons:
  - Any function might crash, and we need to "see" the namespace with the parameters needed in case of a crash, so it has to be a global in `#`.
  - The `⎕TRAP` statement allows us to call a function and to pass parameters except references, so it has to be a named namespace.

Let's investigate how this will work; trace into `#.MyApp.StartFromCmdLine ''`. When you reach line 4 `Config` exists, so now you can call `MyApp.SetTrap` with different left arguments:

```
      SetTrap Config
 0 1000 S
      0 SetTrap Config
 0 1000 S
      1 SetTrap Config
 0 E #.HandleError.Process '#.ErrorParms'
      #.ErrorParms.∆List
 addToMsg
 checkErrorFolder                                                       1
 createHTML                                                             1
 customFns
 customFnsParent
 enforceOff                                                             0
 errorFolder                        C:\Users\kai\AppData\Local\MyApp\Errors
 logFunction                                                          Log
 logFunctionParent   [Logger:C:\Users\...\MyApp_20170305.log(¯70419218)]
 off                                                                   1
 returnCode                                                          104
 saveCrash                                                             1
 saveErrorWS                                                           1
 saveVars                                                              1
 signal                                                                0
 trapInternalErrors                                                    1
 trapSaveWSID                                                          1
 windowsEventSource
```

Test the global trap

We can test this: we could insert a line with a full stop[1] into, say, `CountLettersIn`. But that is awkward: we don't really want to change our source code in order to test error trapping; many applications crashed in production because a programmer forgot to remove a break point before going live. Therefore we invent an additional setting in the INI file:

```
[Config]
Debug       = ¯1    ; 0=enfore error trapping; 1=prevent error trapping;
Trap        = 1     ; 0 disables any :Trap statements (local traps)
ForceError  = 1     ; 1=let TxtToCsv crash (for testing global trap handling)
...
```

That requires two minor changes in `CreateConfig`:

```
∇ Config←CreateConfig dummy;myIni;iniFilename
...
Config.ForceError←0
      iniFilename←'expand'F.NormalizePath'MyApp.ini'
      :If F.Exists iniFilename
          myIni←⎕NEW ##.IniFiles(,⊂iniFilename)
          Config.ForceError←myIni.Get'Config:ForceError'
```

We change `TxtToCsv` so that it crashes in case `Config.ForceError` equals 1:

[1]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.)

```
∇ rc←TxtToCsv fullfilepath;files;tbl;lines;target
⍝ Write a sibling CSV of the TXT located at fullfilepath,
⍝ containing a frequency count of the letters in the file text.
⍝ Returns one of the values defined in `EXIT`.
   MyLogger.Log'Source: ',fullfilepath
   (rc target files)←GetFiles fullfilepath
   {~⍵:r←⍬ ⋄ 'Deliberate error (INI flag "ForceError"'⎕SIGNAL 11}ForceError
...
```

The dfns `{~⍵:r←⍬ ⋄ ...` uses a guard to signal an error in case `⍵` is true and does nothing at all but return a shy result otherwise. In order to test error trapping we don't even need to create and execute a new EXE; instead we just set `ForceError` to 1 and then call `#.MyApp.StartFromCmdLine` from within the WS:

```
      #.MyApp.StartFromCmdLine 'Z:\texts\ulysses.txt'
⍎SYNTAX ERROR
TxtToCsv[6] . ⍝ Deliberate error (INI flag "ForceError")
           ^
```

That's exactly what we want: error trapping should not interfere when we are developing.

To actually test error trapping we need to set the `Debug` flag in the INI file to 0. That will `MyApp` tell that we want error trapping to be active, no matter what environment we are in. Change the INI file accordingly and execute it again.

```
      )reset
      #.MyApp.StartFromCmdLine 'Z:\texts\ulysses.txt'
HandleError.Process caught SYNTAX ERROR
```

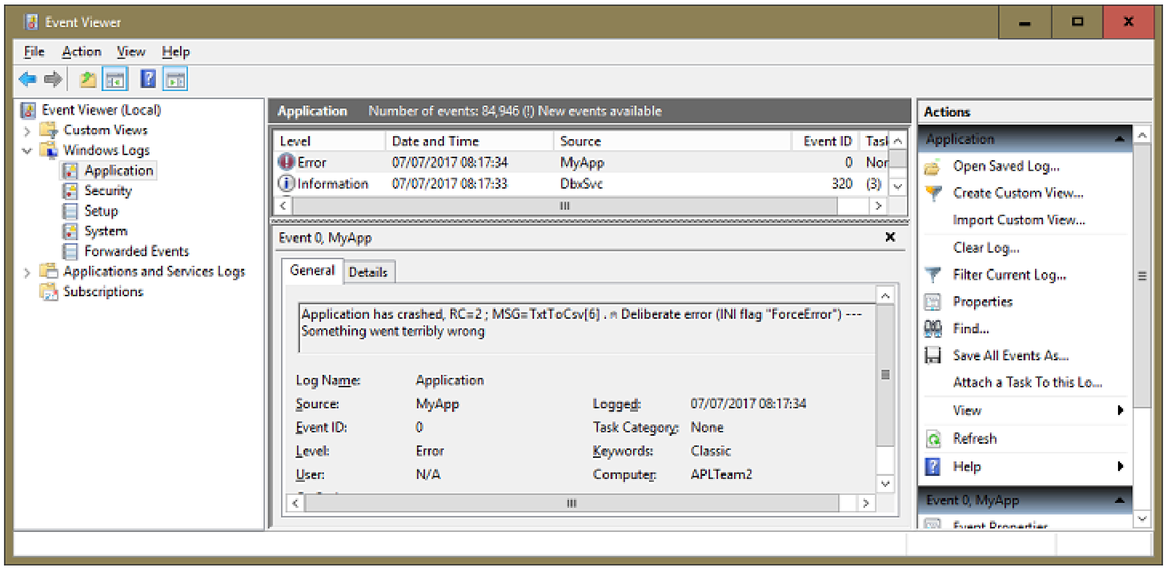
Note that `HandleError` has not executed `⎕OFF` because we executed this in a development environment.

That's all we see in the session, but when you check the folder `#.ErrorParms.errorFolder` you will find that indeed there were three new files created in that folder for this crash. (Note that in case you traced through the code there would be just two files: the workspace is missing. The reason is that with the Tracer active the current workspace cannot be saved, at least not with a pending stack. Generally there are two reasons why no workspace will be saved:

- Any open edit or trace window
- More than one thread was running at the moment of the crash.

This is not strictly true. When `HandleError` detects multiple threads it tries to kill all of them. By definition that won't work because a) it cannot kill the main thread (0) and b) it cannot kill its own thread. However, if it happens to run in the main thread at that very moment it will get rid of all other running threads and be able to save a crash workspace afterwards as a result.

Because we've defined a source for the Windows Event Log `HandleError` has reported the error accordingly:



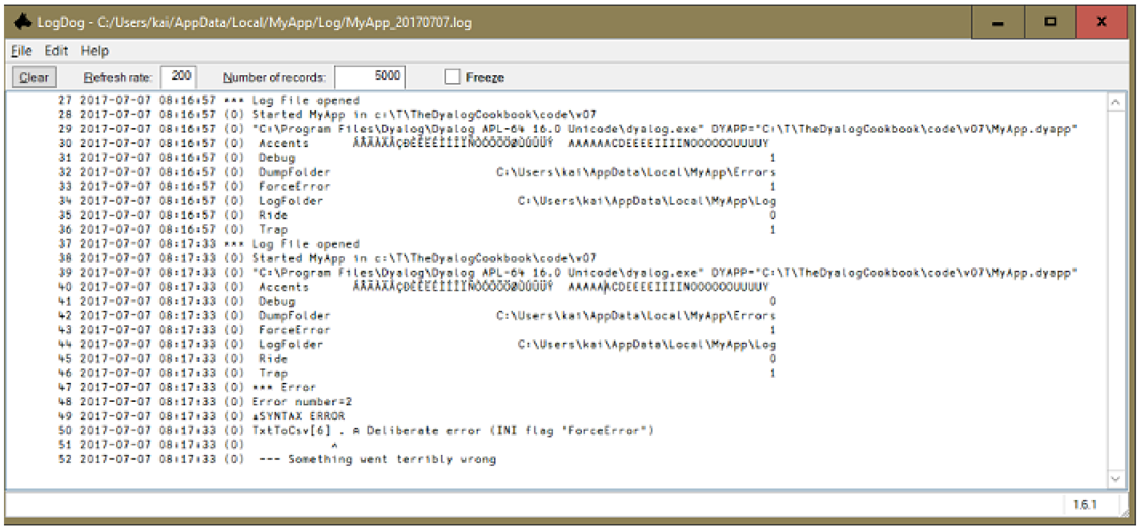
```
Application has crashed, RC=2 ; MSG=TxtToCsv[6] . ⍝ Deliberate error (INI flag "ForceError") ---
Something went terribly wrong

Log Name:     Application
Source:       MyApp            Logged:         07/07/2017 08:17:34
Event ID:     0                Task Category:  None
Level:        Error            Keywords:       Classic
User:         N/A              Computer:       APLTeam2
```

Windows Event Log

As mentioned earlier, the Windows Event Log is discussed in a later chapter.

As already mentioned, the Windows Event Log will be discussed in its own chapter.

We also find evidence in the log file that something broke; see LogDog:



```
27 2017-07-07 08:16:57 *** Log File opened
28 2017-07-07 08:16:57 (0) Started MyApp in c:\T\TheDyalogCookbook\code\v07
29 2017-07-07 08:16:57 (0) "C:\Program Files\Dyalog\Dyalog APL-64 16.0 Unicode\dyalog.exe" DYAPP="C:\T\TheDyalogCookbook\code\v07\MyApp.dyapp"
30 2017-07-07 08:16:57 (0)  Accents      ÀÁÂÃÄÅÇÐÈÉÊËÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ  AAAAAACDEEEEIIIINOOOOOOUUUUY
31 2017-07-07 08:16:57 (0)  Debug                                                1
32 2017-07-07 08:16:57 (0)  DumpFolder              C:\Users\kai\AppData\Local\MyApp\Errors
33 2017-07-07 08:16:57 (0)  ForceError                                           1
34 2017-07-07 08:16:57 (0)  LogFolder                  C:\Users\kai\AppData\Local\MyApp\Log
35 2017-07-07 08:16:57 (0)  Ride                                                 0
36 2017-07-07 08:16:57 (0)  Trap                                                 1
37 2017-07-07 08:17:33 *** Log File opened
38 2017-07-07 08:17:33 (0) Started MyApp in c:\T\TheDyalogCookbook\code\v07
39 2017-07-07 08:17:33 (0) "C:\Program Files\Dyalog\Dyalog APL-64 16.0 Unicode\dyalog.exe" DYAPP="C:\T\TheDyalogCookbook\code\v07\MyApp.dyapp"
40 2017-07-07 08:17:33 (0)  Accents      ÀÁÂÃÄÅÇÐÈÉÊËÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ  AAAAAACDEEEEIIIINOOOOOOUUUUY
41 2017-07-07 08:17:33 (0)  Debug                                                0
42 2017-07-07 08:17:33 (0)  DumpFolder              C:\Users\kai\AppData\Local\MyApp\Errors
43 2017-07-07 08:17:33 (0)  ForceError                                           1
44 2017-07-07 08:17:33 (0)  LogFolder                  C:\Users\kai\AppData\Local\MyApp\Log
45 2017-07-07 08:17:33 (0)  Ride                                                 0
46 2017-07-07 08:17:33 (0)  Trap                                                 1
47 2017-07-07 08:17:33 (0) *** Error
48 2017-07-07 08:17:33 (0) Error number=2
49 2017-07-07 08:17:33 (0) ⍎SYNTAX ERROR
50 2017-07-07 08:17:33 (0) TxtToCsv[6] . ⍝ Deliberate error (INI flag "ForceError")
51 2017-07-07 08:17:33 (0)               ^
52 2017-07-07 08:17:33 (0)  --- Something went terribly wrong
```

The log file

This is done automatically by the `HandleError` class for us because we provided the name of a logging function and a ref pointing to the instance where that log function lives.

We also have an HTML 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 a4ll the variables visible at the moment of the crash. Some of this has got to help.

Note that the crash files names are simply the WSID plus the timestamp prefixed by an underscore:

```
      ⍪{⊃,/1↓⎕NPARTS⍵}¨⊃#.FilesAndDirs.Dir #.ErrorParms.errorFolder,'\'
 MyApp_20170307111141.dcf
 MyApp_20170307111141.dws
 MyApp_20170307111141.html
```

Save your work and re-export the EXE.

7.4 Crash files

What's *in* those crash files?

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

```
MyApp_20170307111141

Version:   Windows-64 16.0 W Development
⎕WSID:           MyApp
⎕IO:           1
⎕ML:           1
⎕WA:           62722168
⎕TNUMS:            0
Category:
EM:                SYNTAX ERROR
HelpURL:
EN:                2
ENX:           0
InternalLocation:        parse.c 1739
Message:
OSError:   0 0
Current Dir:        ...code\v07
Command line:        "...\Dyalog\...\dyalog.exe" DYAPP="...code\v07\MyApp.dyapp"
Stack:

#.HandleError.Process[22]
#.MyApp.TxtToCsv[6]
#.MyApp.StartFromCmdLine[6]
Error Message:

⍎SYNTAX ERROR
TxtToCsv[6] . ⍝ Deliberate error (INI flag "ForceError")
           ^
```

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

```
      (#.ErrorParms.errorFolder,'/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 real problem – the full stop in `MyApp.TxtToCsv` – is some levels down in the stack.

```
      )SI
#.HandleError.SaveErrorWorkspace[7]*
#.HandleError.Process[28]
#.MyApp.TxtToCsv[6]*
#.MyApp.StartFromCmdLine[6]
```

You can clear `HandleError` off the stack with a naked branch arrow; note the `*` on the first and third line. When you do so, you'll find the original global trap restored. Disable it. Otherwise any error you produce while running code will trigger `HandleError` again!

```
      →
      )SI
#.MyApp.TxtToCsv[6]*
#.MyApp.StartFromCmdLine[6]
      ⎕TRAP
  0 E #.HandleError.Process '#.ErrorParms'
      ⎕TRAP←0/⎕TRAP
```

We also want to check whether the correct return code is returned. For that we have to call the EXE, but we don't do this in a console window for reasons we have discussed earlier. Instead we use the `Execute` class which provides two main methods:

- `Process` allows use to catch a program's standard output.
- `Application` allows us to catch a program's exit code.

```
      ⎕←2⊃#.Execute.Application 'Myapp.exe '"Z:\texts\ulysses.txt"'
104
```

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!

7.5 About #.ErrorParms

We've established `#.ErrorParms` as a namespace, and we have explained why: `HandleError.Process` needs to see `ErrorParms` not matter the circumstances, otherwise it cannot work. Since we construct the workspace from scratch when we start developing it cannot do any harm because we quit as soon as the work is done.

Or can it? Let's check. First change the INI file so that it reads:

```
...
Trap         = 1    ; 0 disables any :Trap statements (local traps)
ForceError   = 0    ; 1=let TxtToCsv crash (for testing global trap handling)
...
```

Now double-click the DYAPP, call `#.MyApp.StartFromCmdLine ''` and then execute:

```
      ⎕nnames
C:\Users\kai\AppData\Local\MyApp\Log\MyApp_20170309.log
```

The log file is still open! Now that's what we expect to see as long as `MyLogger` lives, but that is kept local in `#.MyApp.StartFromCmdLine`, so why is this? The culprit is `ErrorParms`! In order to allow `HandleError` to write to our log file we've provided not only the name of the log file but also a reference pointing to the instance the log function is living in:

```
      #.ErrorParms.logFunctionParent
[Logger:C:\Users\kai\AppData\Local\MyApp\Log/MyApp_20170309.log(¯76546889)]
```

In short: we have indeed a good reason to get rid of `ErrorParms` once the program has finished. But how? `⎕SHADOW` to the rescue! With `⎕SHADOW` we can declare a variable to be local from within a function. Mainly useful for localyzing names that are constructed in one way or another we can use it to make `ErrorParms` local within `StartFromCmdLine`. For that we add a single line:

```
∇ {r}←StartFromCmdLine arg;MyLogger;Config;rc;⎕TRAP
⍝ Needs command line parameters, runs the application.
  r←⍬
  #.⎕SHADOW'ErrorParms'
  ⎕WSID←'MyApp'
....
```

Note that we've put `#.` in front of `⎕SHADOW`; that is effectlively the same as having a header `StartFromCmd-Line;#.ErrorParms` only that this is syntactically impossible to do. With `#.⎕SHADOW` it works. When you now try again a double-click on the DYAPP and then call `#.MyApp.StartFromCmdLine` you will find that no file is tied any more, and that `#.ErrorParms` is not hanging around either.

7.6 Very early errors

At the moment there is a possibility that `MyApp` will crash and the global trap is not catching it. This is because we establish the global trap only after having instantiated the INI file: only then do we know where to write the crash files, how to log the error etc. But an error may well occur before that!

Naturally there is no perfect solution available here but we can at least try to catch such errors. For this we establish a `⎕TRAP` with default settings very early, and we make sure that `⎕WSID` is set even earlier, otherwise any attempt to save the crash WS will fail.

```
∇ {r}←StartFromCmdLine arg;MyLogger;Config;rc;⎕TRAP
⍝ Needs command line parameters, runs the application.
  r←⍬
  ⎕WSID←'MyApp'
  ⎕TRAP←1 #.HandleError.SetTrap ⍬
  .
  #.⎕SHADOW'ErrorParms'
  ....
```

Note that we use the `SetTrap` function `HandleError` comes with. It accepts a parameter space as right argument, but it also accepts an empty vector. In the latter case it falls back to the defaults.

For testing purposes we have provided a `1` as left argument, which enforces error trapping even in a development environment. In the following line we break the program with a full stop.

When you now call `#.MyApp.StartFromCmdLine ''` then the error is caught. Of course no logging will take place but it will still try to save the crash files. Since no better place is known it will try to create a folder `MyApp\Errors` in `%LOCALAPPDATA%`.

You can try this now but make sure that when you are ready you remove the line with the full stop from `MyApp.StartFromCmdLine` and also remove the `1` provided as left argument to `HandleError.SetTrap`.

7.7 HandleError in detail

`HandleError` can be configured in many ways by changing the defaults provided by the `CreateParms` method. There is a table with documentation available; execute `]ADoc #.HandleError` and then scroll to `CreateParms`. Most of the parameters are self-explaining but some need background information.

```
      #.HandleError.CreateParms.∆List
 addToMsg
 checkErrorFolder          1
 createHTML                1
 customFns
 customFnsParent
 enforceOff                0
 errorFolder         Errors/
 logFunction
 logFunctionParent
 off                       1
 returnCode                1
 saveCrash                 1
 saveErrorWS               1
 saveVars                  1
 signal                    0
 trapInternalErrors        1
 trapSaveWSID              1
 windowsEventSource
```

`signal`

By default `HandleError` executes `⎕OFF` in a runtime environment. That’s not always the best way to deal with an error. In a complex application it might be the case that just one command fails, but the rest of the application is doing fine. In that case we would be better off by setting `off` to 0 and signal an numeric code that can be caught by yet another `⎕TRAP` that simply allows the user to explore other commands in the application.

`trapInternalErrors`
This flag allows you to switch off any error trapping *within* `HandleError`. This can be useful in case something goes wrong. It's can be useful when working on or debugging `HandleError` itself.

`saveCrash`, `saveErrorWS` **and** `saveVars`
While `saveCrash` and `saveVars` are probably always 1 setting `saveErrorWS` to 0 is perfectly reasonable in case you know upfront that any attempt to save the error WS will fail, for example because your application is multi-threaded. Another good reason to avoid saving a workspace is to keep your code from spying eyes.

`customFns` **and** `customFnsParent`
This allows you to make sure that `HandleError` will call a function of your choice. For example, you can use this to send an email or a text to a certain address.

8. Testing: the sound of breaking glass (Unit tests)

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

All the other code has been written to configure the application, package it 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[1], 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.

8.1 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[2], 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.

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.

[1]Kent Beck, in conversation with one of the authors.
[2]APL has no *string* datatype. We use the word as a casual synonym for *character vector*.

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 recommend it given that the circumstances are right. For example, if you know from the start *exactly* what your program is supposed to do then TDD is certainly an option. If you start prototyping in order to find out what the user actually wants the program to do TDD is no option at all.

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!

8.2 Readability

Reading and understanding APL code is more difficult than in other programming language due to the higher abstraction level and the power of APL's primitives. However, as long as you have an example were the function is fed with correct data it's always possible to decipher the code. Things can become very nasty indeed if an application crashes because inappropriate data arrives at your function. However, before you can figure out whether the data is appropriate or not you need to understand the code - a chicken-egg problem.

That's when test cases can be very useful as well, because they demonstrate which data a function is expected to process. It also emphasizes why it is important to have test cases for all the different types of data (or parameters) a function is supposed to process. In this respect test cases should be exhaustive.

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 the 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.

8.3 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 `{⍵/⍳≢⍵}` [3] 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\v07` as `Z:\code\v08`.

8.4 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`.

Create `Z:\code\v08\Tests.dyalog`:

```
    :Namespace Tests

    :EndNamespace
```

Save this as `Z:\code\v08\Tests.dyalog` and include both scripts in the DYAPP:

[3] With version 16.0 the same can be achieved with the new primitive `⍸flags`.

```
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 you might want to execute `]ADoc #.Tester` to see the documentation for the Tester class if you are in doubt about any of the methods and helpers in `Tester` later on.

8.5 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 [4] you can write both.

In this chapter we'll tackle unit tests.

8.6 Speed

Unit tests should execute *fast*: developers often want to execute them even when still working on a project in order to make sure that they have not broken anything, or to find out what they broke. When executing the test suite takes too long it defeats the purpose.

Sometimes it cannot be avoided that tests take quite a while, for example when testing GUIs. In that case it might be an idea to create a group of tests that comprehend not all but just the most important ones. Those can then be executed while actually working on the code base while the full-blown test suite is only executed every now and then, maybe only before checking in the code.

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

8.7 Preparing: helpers

The first thing we are going to do is to establish a number of helpers in `Tests` that the `Tester` class provides. We can simply call `Tester.EstablishHelpersIn` and provide a ref to the namespace hosting our test cases as right argument:

```
    )cs #
#
    Tester.EstablishHelpersIn #.Tests
    #.Tests.⎕nl 3
FailsIf
G
GoToTidyUp
L
ListHelpers
PassesIf
Run
RunBatchTests
RunBatchTestsInDebugMode
RunDebug
RunThese
∆Failed
∆Inactive
∆LinuxOnly
∆LinuxOrMacOnly
∆LinuxOrWindowsOnly
∆MacOnly
∆MacOrWindowsOnly
∆NoAcreTests
∆NoBatchTest
∆OK
∆WindowsOnly
```

The helpers can be categorized:

- Those starting their names with a `∆` character are niladic functions that return a result. They act like constants in other programming languages. APL does not have constants, but they can be emulated with niladic functions. (Strictly speaking they are not helpers)
- Those starting their names with `Run` are used to run all or selected test cases in slightly different scenarios.
- `FailsIf`, `PassesIf` and `GoToTidyUp` are used for flow control.
- Miscellaneous

Some of the helpers (`G`, `L` and `ListHelpers`) are just helpful while others, like all the `Run*` functions and the flow control functions, are essential. We need them to be around before we can execute any test case. The fact that we had to establish them with a function call upfront contradicts this. But there is an escape route: we add a line to the DYAPP:

```
...
Run #.MyApp.SetLX ⍬
Run #.Tester.EstablishHelpersIn #.Tests
```

Of course we don't need this when DYAPP is supposed to assemble the workspace for a productive environment; we will address this problem later.

We will discuss all helpers in detail, and we start with the flow control helpers.

Flow control helpers

Let's look at an example: `FailsIf` takes a Boolean right argument and returns either `0` in case the right argument is `1` or an empty vector in case the right argument is `0`:

```
      FailsIf 1
0
      FailsIf 0

      ⍴FailsIf 0
0
```

That means that the statement `→FailsIf 1` will jump to 0, exiting the function carrying the statement.

Since GoTo statements are rarely used these days because under most circumstances control structures are way better, it is probably worthwhile to mention that `→⍬` – as well as `→''` – makes the interpreter carry on with the next line. In other words the function just carries on. That's exactly what we want when the right argument of `FailsIf` is a `0` because in that case the test has not failed.

`PassesIf` is exactly the same thing but just with a negated argument: it returns a `0` when the right argument is `0` and an empty vector in case the right argument is `1`.

`GoToTidyUp` is a special case. It returns an empty vector in case the right argument is `0`. If the right argument is `1` it expects the function where it was called from to have a line that carries a label `∆TidyUp`; the line number of that label is then returned.

This is useful in case a test function needs to do some cleaning up, no matter whether it has failed or not. Imagine you need a temporary file for a test but want to delete it after carrying out the test case. In that case the bottom of your test function might look like this:

```
...
∆TidyUp:
    #.FilesAndDirs.DeleteFile tempFilename
```

When everything goes according to plan the function would eventually execute these lines anyway, but when a test case fails you need this:

```
    →GoToTidyUp expected≢result
```

Like `FailsIf` the test function would just carry on in case `expected≢result` returns a `0` but jump to the label `∆TidyUp` in case the test fails (=the condition is true).

But why are we using functions for all this anyway? We could do without, couldn't we? Yes, so far we could, but there is just one more thing. Stay with us…

8.8 Writing unit tests

We have automated the way the helpers are established in `Tests`. Now we are ready to implement the first test case.

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 with `map` in `#.Utilities`. We know by now that in general it works although even that needs to be confirmed by a test of course. What we don't know yet is whether it works under all circumstances. We also need to make sure that it complains when it is fed with inappropriate data.

To make writing test cases as easy as possible you can ask `Tester` for providing a test case template.

```
    ⎕←⍪#.Tester.GetTestFnsTemplate
  R←Test_000(stopFlag batchFlag);⎕TRAP
 ⍝ Model for a test function.
  ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
  R←∆Failed

 ⍝ Preconditions...
 ⍝ ...

  →PassesIf 1≡1
  →FailsIf 1≢1
  →GoToTidyUp 1≢1
  R←∆OK

 ∆TidyUp: ⍝ Clean up after this label
          ⍝ ...
```

The template covers all possibilities, and we will discuss all of them. However, for the time being we want to keep it simple, so we will delete quite a lot and also add three more functions:

```
:Namespace Tests
⎕IO←1 ⋄ ⎕ML←1
∇Initial
  U←##.Utilities ⋄ F←##.FilesAndDirs ⋄ A←##.APLTreeUtils
∇
∇ R←Test_001(stopFlag batchFlag);⎕TRAP
 ⍝ Is the length of the left argument of the `map` function checked?
  ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
  R←∆Failed
  :Trap 5
      {}(⊂⎕A)U.map'APL is great'
      →FailsIf 1
  :Else
      .
  :EndTrap
  R←∆OK
∇
∇ {r}←GetHelpers
  r←##.Tester.EstablishHelpersIn ⎕THIS
∇
∇ Cleanup dummy
  ⎕EX¨'AFU'
∇
:EndNamespace
```

What we changed:

- We renamed the template from `Test_000` to `Test_001`.
- We described in line 1 as thoroughly as possible what the test case is doing. The reason is that this line is later the only way to tell this test case from any other. In other words, it is really important to get this right. By the way: if you cannot describe in a single line what the test case is doing it's most likely doing too much.
- We set `⎕TRAP` so that any error 999 will stop the interpreter with a deliberate error; we will soon see why and what for.
- We also localize `⎕TRAP` so that any error 999 has an effect only within the test function. In fact any other error than 999 is passed through with the `N` option in order to allow `⎕TRAP`s further up the stack to take control.
- We initialize the explicit result `R` by assigning the value returned by the niladic function `∆Failed`. That allows us to simply leave the function in case a test fails: the result will then tell.
- We trap the call to `map` which we expect to fail with a length error because we provide a scalar as left argument.
- In case there is no error we call the function `∆FailsIf` and provide a `1` as right argument. That makes `∆FailsIf` return a `0` and therefore leave `Test_001`.

You might have noticed that we address, say, `Utilities` with `##.Utilities` rather than `#.Utilities`. Making this a habit is a good idea: currently it does not make a difference, but when you later decide to move everything in `#` into, say, a namespace `#.Container` (you never know!) then `##.` would still work while `#.` wouldn't.

The `:Else` part is not ready yet; the full stop will prevent the test function from carrying on when we get there.

Notes:

- We also added a function `GetHelpers` to the script. The reason is that the helpers will disappear as soon as we fix the `Test` script. And we are going to fix it whenever we change something or add a new test case. Therefore we need an easy way to get them back: `GetHelpers` to the rescue.
- The function `Initial` will be executed by the test framework before any test function is executed. This can be used to initialize stuff that all (or many) test cases need. Here we establish the references `A` (for the `APLTreeUtils` module), `F` (for the `FilesAndDirs` module) and `U` (for the `Utilities` module). `Initial` relies on naming conventions; in case there is a function around with that name it will be executed. More later.
- The function `Cleanup` will be executed by the test framework after all test functions have been executed. This can be used to clean up stuff that's not needed any more. Here we delete the references `A`, `F` and `U`. More later.

Ordinary namespaces versus scripted ones

There's a difference between an ordinary namespace and a scripted namespace: imagine you've called `#.Tester.EstablishHelpersIn` within an ordinary namespace. Now you change/add/delete test functions; that would have no effect on anything else in that namespace. In other words, the helpers would continue to exist.

When you change a namespace script on the other hand the namespace is re-created from the script, and that means that our helpers will disappear because they are not a part of the `Tests` script.

Let's call our test case. We do this by running the `Run` method first:

```
Run
--- Test framework "Tester" version 3.5.0 from 2017-07-16 --------------------------------
Searching for INI file Testcases.ini
  ...not found
Searching for INI file testcases_APLTEAM2.ini
  ...not found
Looking for a function "Initial"...
  "Initial" found and successfully executed
--- Tests started at YYYY-MM-DD hh:mm:ss on #.Tests --------------------------------------
# Test_001 (1 of 1) : Is the length of the left argument of the `map` function checked?
 ------------------------------------------------------------------------------------------
   1 test case executed
   0 test cases failed
   1 test case broken
```

```
Time of execution recorded on variable #.Tests.TestCasesExecutedAt in: YYYY-MM-DD hh:mm:ss
Looking for a function "Cleanup"...
  Function "Cleanup" found and executed.
*** Tests done
```

That's what we expect.

Note that there are INI files mentioned. Ignore this for the time being; we will discuss this later on.

What is a test case?!

You might wonder how `Run` established what is a test case and what isn't: that's achieved by naming conventions. Any test function *must* start their name with `Test_`. After that there are two possibilities:

1. In the simple case there are one or more digits after the `_`; nothing but digits. Therefore these all qualify as test cases: `Test_1`, `Test_01`, `Test_001` and so on. `Test_01A` however does not.
2. In case you have a large number of test cases you most probably want to group them in one way or another. You can add a group name after the first _ and add a second _ followed by one or more digits. Therefore `Test_map_1` is recognized as a test case, and so is `Test_Foo_9999`. `Test_Foo_Goo_1` however is not.

What if we want to look into a broken or failing test case? Of course in our current scenario – which is extremely simple – we could just trace into `Test_001` and find out what's going on, but if we take advantage of the many features the test framework is actually offering then we cannot do this (soon it will become clear why). However, there is a way to do this no matter whether the scenario is simple, reasonably complex or extremely complex: we call `RunDebug`:

```
RunDebug 0
--- Test framework "Tester" version 3.6.0 from  -------
Searching for INI file testcases_{computername}.ini
  ...not found
Searching for INI file Testcases.ini
  ...not found
Looking for a function "Initial"...
  "Initial" found and successfully executed
--- Tests started at YYYY-MM-DD hh:mm:ss on #.Tests -------------
SYNTAX ERROR
      . ⍝ Deliberate error
     ^
      )si
#.Tests.Test_001[6]*
⍎
#.Tester.ExecuteTestFunction[6]
```

```
#.Tester.ProcessTestCases[6]
#.Tester.Run__[39]
#.Tester.RunDebug[17]
#.Tests.RunDebug[3]
```

It stopped in line 6. Obviously the call to `FailsIf` has something to do with this, and so has the `⎕TRAP` setting, because apparently that's where the "Deliberate error" comes from. Indeed this is the case: all three flow control functions, `FailIf`, `PassesIf` and `GoToTidyUp` check whether they are running in debug mode and if that is the case then rather returning a result that indicates a failing test case they `⎕SIGNAL 999` which is then caught by the `⎕TRAP` which in turn first prints `⍝ Deliberate error` to the session and then hands over control to the user. You can now investigate variables or start the Tracer etc. in order to investigate why the test case failed.

The difference between `Run` and `RunDebug` is the setting of the first of the two flags provided as right argument to the test function: `stopFlag`. This is `0` when `Run` executes the test cases, but it is `1` when `RunDebug` is in charge. The three flow control functions `FailsIf`, `PassesIf` and `GoToTidyUp` all honour `stopFlag` - that's how it works.

Now sometimes you don't want the test function to go to the point where the error actually appears, for example in case the test function does a lot of precautioning, and you want to check this upfront because there might be something wrong with it, causing the failure. Note that so far we passed a `0` as right argument to `RunDebug`. If we pass a `1` instead then the test framework would stop just before it would execute the test case:

```
      RunDebug 1
--- Test framework "Tester" version 3.6.0 from YYYY-MM-DD -------
Searching for INI file Testcases.ini
  ...not found
Searching for INI file testcases_APLTEAM2.ini
  ...not found
Looking for a function "Initial"...
  "Initial" found and successfully executed
--- Tests started at YYYY-MM-DD hh:mm:ss on #.Tests -------------

ExecuteTestFunction[6]
      )si
#.Tester.ExecuteTestFunction[6]*
#.Tester.ProcessTestCases[6]
#.Tester.Run__[39]
#.Tester.RunDebug[17]
#.Tests.RunDebug[3]
```

You could now trace into `Test_001` and investigate. Instead enter `→0`. You should see something like this:

```
* Test_001 (1 of 1) : Is the length of the left argument of the `map` function checked?
 ----------------------------------------------------------------------------------------------------\
----------------------------------------------------------------------
   1 test case executed                                                                              \

   1 test case failed                                                                                \

   0 test cases broken                                                                               \

Time of execution recorded on variable #.Tests.TestCasesExecutedAt in: YYYY-MM-DD hh:mm:ss
Looking for a function "Cleanup"...
  Function "Cleanup" found and executed.
*** Tests done
```

Let's make sure that `map` is checking its left argument:

```
:Namespace Utilities
      map←{
          (,2)≢⍴⍺:'Left argument is not a two-element vector'⎕SIGNAL 5
          (old new)←⍺
          nw←∪⍵
          (new,nw)[(old,nw)⍳⍵]
      }
:EndNamespace
```

Now run `RunDebug 1`. Trace into `Test_001` and watch whether now any error 5 (LENGTH ERROR) is trapped, You should end up on line 8 of `Test_001`. Exchange the full stop by:

```
    →PassesIf'Left argument is not a two-element vector'≡⎕DMX.EM
```

⎕DM versus ⎕DMX

You have always used `⎕DM`, and it was perfectly fine, right? No need to switch to the (relatively) new `⎕DMX`, right? Well, the problem with `⎕DM` is that it is not thread save while `⎕DMX` is. That's why we suggest that you stop using `⎕DM` and use just `⎕DMX`. It also provides more and more precise information.

This checks whether the error message is what we expect. Trace through the test function and watch what it is doing. After having left the test function you may click the green triangle in the Tracer ("Continue execution of all threads").

Now what if you've executed, say, not one but 300 test cases with `Run`, and just one failed, say number 289? You expected them all to succeed but since one did not you need to check on the failing one. Calling `Run` as well as `RunDebug` would always execute *all* test cases found. The function `RunThese` allows you to run just the specified test functions:

```
      RunThese 1
--- Test framework "Tester" version 3.5.0 from 2017-07-16 --------------------------------
Searching for INI file Testcases.ini
  ...not found
Searching for INI file testcases_APLTEAM2.ini
  ...not found
Looking for a function "Initial"...
  "Initial" found and successfully executed
--- Tests started at YYYY-MM-DD hh:mm:ss on #.Tests --------------------------------------
  Test_001 (1 of 1) : Process a single file with .\MyApp.exe
 ------------------------------------------------------------------------------------------
   1 test case executed
   0 test cases failed
   0 test cases broken
Time of execution recorded on variable #.Tests.TestCasesExecutedAt in: YYYY-MM-DD hh:mm:ss
Looking for a function "Cleanup"...
  Function "Cleanup" found and executed.
*** Tests done
```

This would run just test case number 1. If you specify it as `¯1` it would stop just before actually executing the test case. Same as before since we have just one test function yet but take our word for it, it would execute just `Test_001` no matter how many other test cases there are.

We have discussed the functions `Run`, `RunDebug` and `RunThese`. That leaves `RunBatchTests` and `Run-BatchTestsInDebugMode`; what are they for? Imagine a test that would either require an enormous amount of effort to implement or alternatively you just build something up and then ask the human in front of the monitor: "Does this look alright?". That's certainly *not* a batch test case because it needs a human sitting in front of the monitor. If you know upfront that there won't be a human paying attention then you can prevent non-batch test cases from being executed by calling either `RunBatchTests` or `RunBatchTestsInDebugMode`.

But how does this work? We already learned that `stopFlag`, the first of the two flags passed to any test case as the right argument, is ruling whether any errors are trapped or not. The second flag is called `batchFlag`, and that gives you an idea what it's good for. If you have a test which interacts with a user (=cannot run without a human) then your test case would typically look like this:

```
 R←Test_001(stopFlag batchFlag);⎕TRAP
⍝ Check ...
 ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
 R←∆Failed
 :If batchFlag
     ⍝ perform the test
     R←∆OK
 :Else
     R←∆NoBatchTest
 :EndIf
```

The test function checks the `batchFlag` and tells via the explicit result that it did not execute because it is not suitable for batch testing.

One can argue whether the test case we have implemented makes much sense, but it allowed us to investigate the basic features of the test framework. We are now ready to investigate the more sophisticated features.

Of course we also need a test case that checks whether `map` does what it's supposed to do when appropriate arrays are passed as arguments, therefore we add this to `Tests`:

```
Namespace Tests

∇ R←Test_001(stopFlag batchFlag);⎕TRAP
...
∇

∇ R←Test_002(stopFlag batchFlag);⎕TRAP;Config;MyLogger
  ⍝ Check whether `map` works fine with appropriate data
  ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
  R←∆Failed
  (Config MyLogger)←##.MyApp.Initial ⍬
  →FailsIf'APL IS GREAT'≢Config.Accents U.map A.Uppercase'APL is great'
  →FailsIf'UßU'≢Config.Accents U.map A.Uppercase'üßÜ'
  R←∆OK
∇
...
```

Note how using the references `U` and `A` here simplifies the code greatly.

Now we try to execute this test cases:

```
      #.Tests.GetHelpers
      RunThese 2
--- Test framework "Tester" version 3.6.0 from YYYY-MM-DD ----------------
Searching for INI file testcases_{computername}.ini
  ...not found
Searching for INI file Testcases.ini
  ...not found
Looking for a function "Initial"...
  "Initial" found and successfully executed
--- Tests started at YYYY-MM-DD hh:mm:ss on #.Tests ----------------------
  Test_002 (1 of 1) : Check whether `map` works fine with appropriate data
 -------------------------------------------------------------------------
   1 test case executed
   0 test cases failed
   0 test cases broken
```

Works fine. Excellent.

Now let's make sure that the workhorse is doing okay; for this we add another test case:

```
:Namespace Tests
...
    ∇ R←Test_002(stopFlag batchFlag);⎕TRAP
...
    ∇ R←Test_003(stopFlag batchFlag);⎕TRAP;Config;MyLogger
    ⍝ Test whether `TxtToCsv` handles a non-existing file correctly
      ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
      R←∆Failed
      rc←##.MyApp.TxtToCsv 'This_file_does_not_exist'
      →FailsIf ##.MyApp.EXIT.SOURCE_NOT_FOUND≢rc
      R←∆OK
    ∇
...
```

Let's call this test:

```
      )CS #.Tests
#.Tests
      GetHelpers
      RunThese 3
...
VALUE ERROR
TxtToCsv[4] MyLogger.Log'Source: ',fullfilepath
            ^
```

Oops. `MyLogger` is undefined. In the envisaged use in production, it is defined by and local to `StartFromCmdLine`. That design followed Occam's Razor[5]: (entities are not to be needlessly multiplied) in keeping the log object in existence only while needed. But it now prevents us from testing `TxtToCsv` independently. So we'll refactor:

```
:Namespace Tests
...
    ∇ R←Test_003(stopFlag batchFlag);⎕TRAP
    ⍝ Test whether `TxtToCsv` handles a non-existing file correctly
      ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
      R←∆Failed
      ##.MyApp.(Config MyLogger)←##.MyApp.Initial ⍬
      rc←##.MyApp.TxtToCsv 'This_file_does_not_exist'
      →FailsIf ##.MyApp.EXIT.SOURCE_NOT_FOUND≢rc
      R←∆OK
    ∇
...
```

Note that now both `Config` and `MyLogger` exist within `MyApp`, not in `Tests`. Therefore we don't even have to keep them local within `Test_003`. They are however not part of the script, therefore they will cease to exist as soon as the script `Tests` is fixed again, very much like the helpers.

Let's try again:

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

```
      RunThese 3
--- Test framework "Tester" version 3.6.0 from YYYY-MM-DD -------------------------
Searching for INI file testcases_{computername}.ini
  ...not found
Searching for INI file Testcases.ini
  ...not found
Looking for a function "Initial"...
  "Initial" found and successfully executed
--- Tests started at 2017-03-22 15:56:41 on #.Tests -------------------------------
  Test_003 (1 of 1) : Test whether `TxtToCsv` handles a non-existing file correctly
 ----------------------------------------------------------------------------------
   1 test case executed
   0 test cases failed
   0 test cases broken
Time of execution recorded on variable #.Tests.TestCasesExecutedAt: YYYY-MM-DD hh:mm:ss
Looking for a function "Cleanup"...
  Function "Cleanup" found and executed.
```

Clearly we need to have one test case for every result the function `TxtToCsv` might return but we leave that as an exercise to you. We have more important test cases to write: we want to make sure that whenever we create a new version of the EXE it will keep working.

Let's rename the test functions we have so far:

- `Test_001` becomes `Test_map_01`
- `Test_002` becomes `Test_map_02`
- `Test_003` becomes `Test_TxtToCsv_01`

The new test cases we are about to add will be named `Test_exe_01` etc. For our application we could get away without grouping, but once you have more than, say, 20 test cases grouping is a must. Therefore we take the opportunity to demonstrate how this can be done.

The "Initial" function

We've already introduced a function `Initial` for establishing the references `A`, `U` and `F` before we execute any test cases. For testing the EXE we need a folder where we can store files temporarily. We add this to `Initial`:

```
:Namespace Tests
⎕IO←1 ⋄ ⎕ML←1
 ∇ R←Initial;list;rc
   U←##.Utilities ⋄ F←##.FilesAndDirs ⋄ A←##.APLTreeUtils
   ∆Path←F.GetTempPath,'\MyApp_Tests'
   F.RmDir ∆Path
   'Create!'F.CheckPath ∆Path
   list←⊃F.Dir'..\..\texts\en\*.txt'
   rc←list F.CopyTo ∆Path,'\'
   :If ~R←0∧.=⊃rc
       ⎕←'Could not create ',∆Path
```

```
    :EndIf
 ∇
...
```

`Initial` does not have to return a result but if it does it must be a Boolean. For "success" it should return a `1` and otherwise a `0`. If it does return `0` then no test cases are executed but if there is a function `Cleanup` it will be executed. Therefore `Cleanup` should be ready to clean up in case `Initial` was only partly or not at all successful.

We have changed `Initial` so that it now returns a result because copying the files over might fail for all sorts of reasons, and we cannot do without.

`Initial` may or may not accept a right argument. If it does it will be fed with a namespace that holds all the parameters.

What we do in `Initial` apart from creating the references:

- First we create a global variable `∆Path` which holds a path to a folder `MyApp_Tests` within the Windows temp folder.
- We then remove that folder in case it still exists from any previously failing test cases.
- We then create it.
- We ask for a list of all text files in the `texts\en\` folder.
- We copy all those files over to our temporary test folder.
- Finally we check the return code of the copy operation; `R` gets 1 (indicating success) only in case they were successful.

Machine-dependent initialisation

What if you need to initialise something (say a database connection) but it is somehow different depending on what machine the tests are executed on (IP address, user-id, password…)? The test framework tries to find two different INI files in the current directory: First it looks for `testcase.ini`. It then tries to find `testcase_{computername}.ini`. "computername" is what you get when you execute `⊣ 2 ⎕nq # 'GetEnvironment' 'Computername'`.

If it finds any of them (or both) it instantiates the `IniFile` class as `INI` on these INI files within the namespace that hosts your test cases. In case of a name clash the setting in `testcase_{computername}.ini` will win because it is the last one.

Now we are ready to test the EXE; create it from scratch. Our first test case will process the file "ulysses.txt":

```
:Namespace Tests
...
    ∇ R←Test_exe_01(stopFlag batchFlag);⎕TRAP;rc
      ⍝ Process a single file with .\MyApp.exe
      ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
      R←∆Failed
     ⍝ Precautions:
      F.DeleteFile⊃F.Dir ∆Path,'\*.csv'
      rc←##.Execute.Application'MyApp.exe ',∆Path,'\ulysses.txt'
      →GoToTidyUp ##.MyApp.EXIT.OK≠⊃rc
      →GoToTidyUp~F.Exists ∆Path,'\ulysses.csv'
      R←∆OK
     ∆TidyUp:
      F.DeleteFile⊃F.Dir ∆Path,'\*.csv'
    ∇
...
```

Notes:

- First we make sure that there are no CSV files in `∆Path`.
- Then we call the EXE and pass the filename of "ulysses" as a command line parameter.
- We check the return code and jump to `∆TidyUp` in case it's not what we expect.
- We then check whether there is now a file "ulysses.cvs" in `∆Path`.
- Finally we clean up and delete (again) all CSV files in `∆Path`.

Let's run our new test case:

```
      GetHelpers
      RunThese 'exe'
--- Test framework "Tester" version 3.6.0 from YYYY-MM-DD -----
Searching for INI file testcases_{computername}.ini
  ...not found
Searching for INI file Testcases.ini
  ...not found
Looking for a function "Initial"...
  "Initial" found and successfully executed
--- Tests started at YYYY-MM-DD hh:mm:ss on #.Tests -----------
  Test_exe_01 (1 of 1) : Process a single file with .\MyApp.exe
 ---------------------------------------------------------------
   1 test case executed
   0 test cases failed
   0 test cases broken
Time of execution recorded on variable #.Tests.TestCasesExecutedAt in: YYYY-MM-DD hh:mm:ss
Looking for a function "Cleanup"...
  Function "Cleanup" found and executed.
```

We need one more test case:

```
:Namespace Tests
...
∇ R←Test_exe_01(stopFlag batchFlag);⎕TRAP;rc
...
∇ R←Test_exe_02(stopFlag batchFlag);⎕TRAP;rc;listCsvs
  ⍝ Process all TXT files in a certain directory
  ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
  R←∆Failed
  ⍝ Precautions:
  F.DeleteFile⊃F.Dir ∆Path,'\*.csv'
  rc←##.Execute.Application'MyApp.exe ',∆Path,'\'
  →GoToTidyUp ##.MyApp.EXIT.OK≠⊃rc
  listCsvs←⊃F.Dir ∆Path,'\*.csv'
  →GoToTidyUp 1≠⍴listCsvs
  →GoToTidyUp'total.csv'≢A.Lowercase⊃,/1↓⎕NPARTS⊃listCsvs
  R←∆OK
 ∆TidyUp:
  F.DeleteFile⊃F.Dir ∆Path,'\*.csv'
∇
...
```

This one will process *all* TXT files in `∆Path` and create a file `total.csv`. We check whether this is the case and we are done. Almost: in a real world application we most likely would also check for a path that contains spaces in its name. We don't do this, instead we execute the full test suite:

```
      GetHelpers
      ⎕←⊃Run
--- Test framework "Tester" version 3.6.0 from YYYY-MM-DD ----------------------------
Searching for INI file testcases_{computername}.ini
  ...not found
Searching for INI file Testcases.ini
  ...not found
Looking for a function "Initial"...
  "Initial" found and successfully executed
--- Tests started at YYYY-MM-DD hh:mm:dd on #.Tests -------------------------------------
  Test_TxtToCsv_03 (1 of 5) : Test whether `TxtToCsv` handles a non-existing file correctly
  Test_exe_01 (2 of 5)      : Process a single file with .\MyApp.exe
  Test_exe_02 (3 of 5)      : Process all TXT files in a certain directory
  Test_map_01 (4 of 5)      : Is the length of the left argument of the `map` function checked?
  Test_map_02 (5 of 5)      : Check whether `map` works fine with appropriate data
 -------------------------------------------------------------------------------------
   5 test cases executed
   0 test cases failed
   0 test cases broken
Time of execution recorded on variable #.Tests.TestCasesExecutedAt in: YYYY-MM-DD hh:mm:ss
Looking for a function "Cleanup"...
  Function "Cleanup" found and executed.
0
```

Note that the function `Run` prints its findings to the session but also returns a result. That's a two-item vector:

1. Is a return code. `0` means "okay".
2. Is a vector of vectors that is identical with what's printed to the session.

Cleaning up

Although we have been careful and made sure that every single test case cleans up after itself (in particular those that failed), we have not removed the directory `∆Path` points to. We add some code to the `Cleanup` function in order to achieve that:

```
:Namespace Tests
...
∇ Cleanup dummy
  ⎕EX¨'AFU'
  :If 0<⎕NC'∆Path'
      ##.FilesAndDirs.RmDir ∆Path
      ⎕EX '∆Path'
  :EndIf
∇

:EndNamespace
```

This function now checks whether a global `∆Path` exists. If that's the case then the directory it is pointing to is removed and the global variable deleted. The `Tester` framework checks whether there is a function `Cleanup`. If that's the case the function is executed after the last test case has been executed. The function must be either monadic or niladic; in case it is a monadic function the right argument will be `⍬`. It must either return a shy result (ignored) or no result at all.

Markers

We've already mentioned elsewhere that it is useful to mark code in particular ways, like `⍝FIXME⍝` or `⍝TODO⍝`. It is an excellent idea to have a test case that checks for such markers. Before something makes it to a customer such strings should probably be removed from the code.

The "L" and "G" helpers

Now that we have three groups we can take advantage of the `G` and the `L` helpers:

```
      G
exe
map
TxtToCsv
      L''
 Test_exe_01       Process a single file with .\MyApp.exe
 Test_exe_02       Process all TXT files in a certain directory
 Test_map_01       Is the length of the left argument of the `map` function checked?               \

 Test_map_02       Check whether `map` works fine with appropriate data
 Test_TxtToCsv_01  Test whether `TxtToCsv` handles a non-existing file correctly
      L'ex'
 Test_exe_01  Process a single file with .\MyApp.exe
 Test_exe_02  Process all TXT files in a certain directory
```

8.9 TestCasesExecutedAt

Whenever the test cases were executed `Tester` notifies the time on a global variable `TestCasesExecutedAt` in the hosting namespace. This can be used in order to find out whether part of the code has been changed since the last time the cases were executed. However, in order to do this you have to make sure that the variable is either saved somewhere or added to the script `Tests`. For example, it could be handled by a cover function that calls any of `Tester`s `Run*` functions and then handled that variable.

8.10 Conclusion

We have now a test suite available that allows us at any stage to call it in order to make sure that everything still works. This is invaluable.

8.11 The sequence of tests

Please note that there is always the possibility of test cases being dependent on another, even if you try to avoid that. That might be by mistake or due to an unnoticed side effect. That doesn't mean that you shouldn't aim for making all test cases completely independent from one another. Watch out: a future version of `Tester` might come with an option that shuffles the test cases before executing them.

8.12 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 one of:

- a test machine for each OS (version of Windows) you support.
- 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.

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 10 GB 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 20 seconds or less.

8.13 Testing APLTree modules

By now we are using quite a number of modules from the APLTree project. Shouldn't we test them as well? After all if they break our application will stop working! Well, there are pros and cons:

Pro The modules have their own unit tests, and those are exhaustive. An update is published only after all the test cases have passed.

The modules are constantly adapted to new demands or changes in the environment etc. Therefore a new version of Windows or Dyalog won't break them, although you need to allow some time for this to happen. "Some time" just means that you cannot expect the APLTree modules to be ready on the day a new version of either Windows or Dyalog becomes available.

Contra
We cannot know whether those test cases cover the same environment(s) (different versions of Windows, different versions of Dyalog, domain-managed network or not, network drives or not, multi-threaded versus single-threaded, you name it) our application will run in.

That clearly means that we should incorporate the tests those modules come with into our own test suite, although we are sure that not too many people/companies using modules from the APLTree library are actually doing this.

Anyway, it's not difficult to do at all: every module has a workspace saved on GitHub that comes with all that's needed in order to carry out the test cases. All it requires it starting Dyalog (*your* version of Dyalog that is), load that workspace, execute `#.TestCases.Run` (because all modules of the APLTree library host their test cases in an ordinary (non-scripted) namespace, catch the result and return it with `⎕OFF` to the calling environment. As long as that is `0` that's all what's required.

If it's not `0` you start your version of Dyalog, load the workspace of the module with one or more failing test cases and run `#.TestCases.RunDebug 0` in order to investigate what went wrong.

9. 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

9.1 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.*

9.2 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 [1] 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.

9.3 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?

[1]Compile those from questions actually asked by users. It's a common mistake to put together "Question we would like our users to ask".

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*.

9.4 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.

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, do write an explanatory comment. Better to reserve trailing comments for other notes, such as `⍝FIXME⍝ slow for >1E7 elements`[2]. (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[3].

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

Note that Dyalog offers a special command for aligning comments: "AC". You can assign a keystroke to this command: open the "Configuration" dialog (Options / Configure…), select the "Keyboard Shortcuts" tab and sort the table with a click on the "Code" column, then look for "AC".

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.

9.5 ADoc

ADoc is an acronym for *automatic documentation* generation. It works on classes and namespaces.

In its most basic function, it lists methods, properties and fields (functions, operators and variableas) and requires no comments in the code. In its more powerful function, it composes from header comments an HTML page. Honouring Markdown conventions, it provides all the typographical features you need for documentation. If you don't know what Markdown is please read the Markdown article on Wikipedia[4] and `Markdown2Help`'s own help file. The time will be a good investment in any case because these days Markdown is used pretty much everywhere.

Previously only found as a class in the APLTree library, it is now shipped in Dyalog Version 16.0 as a user command.

Get a summary

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

[2]Be it `⍝FIXME⍝` or `⍝CHECKME⍝` or `⍝TODO⍝` - what matters is that you keep it consistent and searchable. That implies that the search term cannot be mistaken as something else by accident. For that reason `⍝TODO⍝` is slighty better better than `TODO`.

[3]Thanks to Roger Hui for this term.

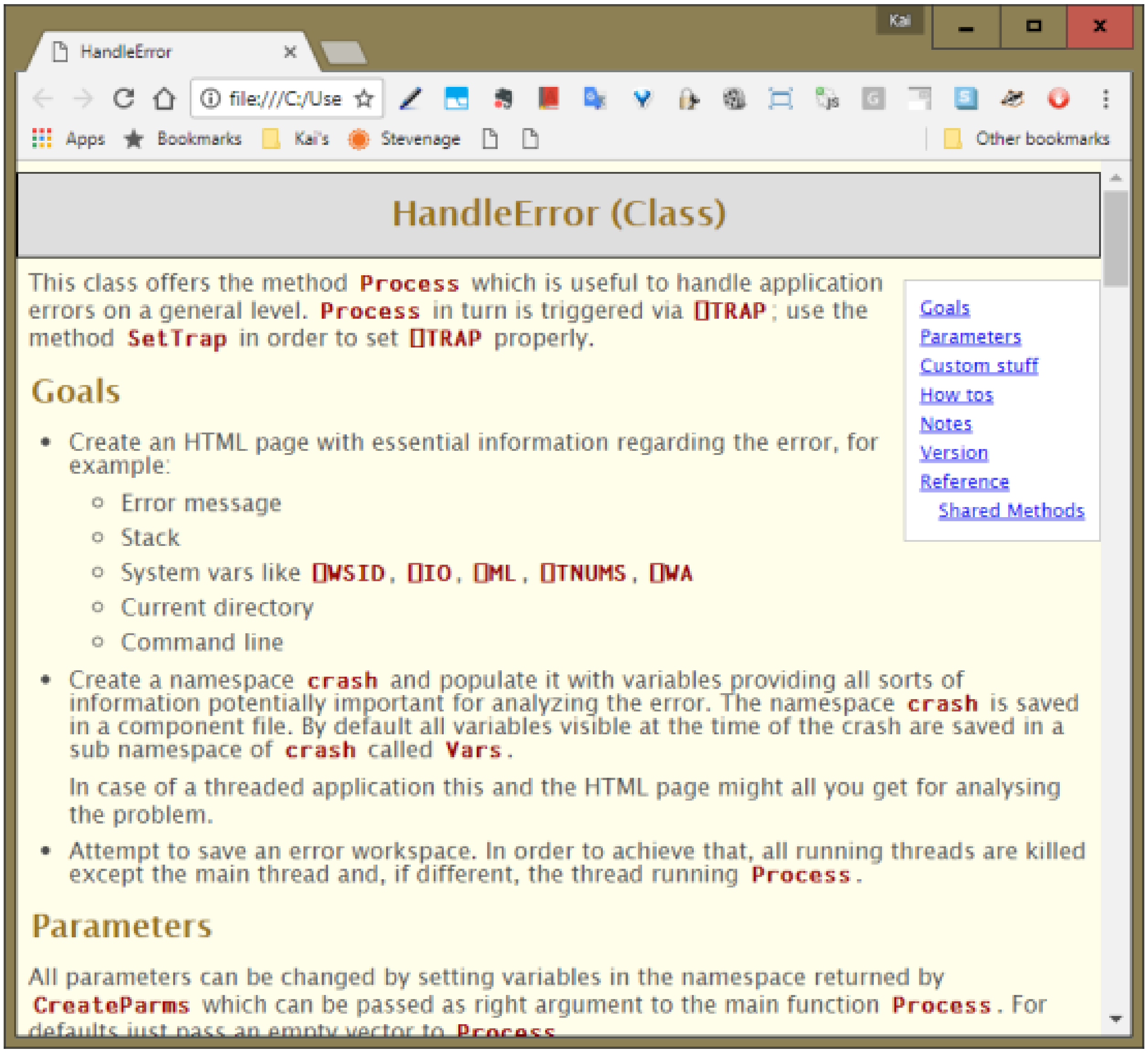
[4]https://en.wikipedia.org/wiki/Markdown

```
      ]ADoc #.HandleError -summary
*** HandleError (Class) ***

Shared Methods:
  CreateParms
  Process
  ReportErrorToWindowsLog
  SetTrap
  Version
```

For a more detailed list with arguments and results specify `]ADoc #.HandleError -summary=full`.

Get it all

```
]ADoc #.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
Parameters
Custom stuff
How tos
Notes
Version
Reference
Shared Methods

Goals

- Create an HTML page with essential information regarding the error, for example:
  - Error message
  - Stack
  - System vars like ⎕WSID, ⎕IO, ⎕ML, ⎕TNUMS, ⎕WA
  - Current directory
  - Command line
- 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 all variables visible at the time of the crash are saved in a sub namespace of crash called Vars.

  In case of a threaded application this and the HTML page might all you get for analysing the problem.
- 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

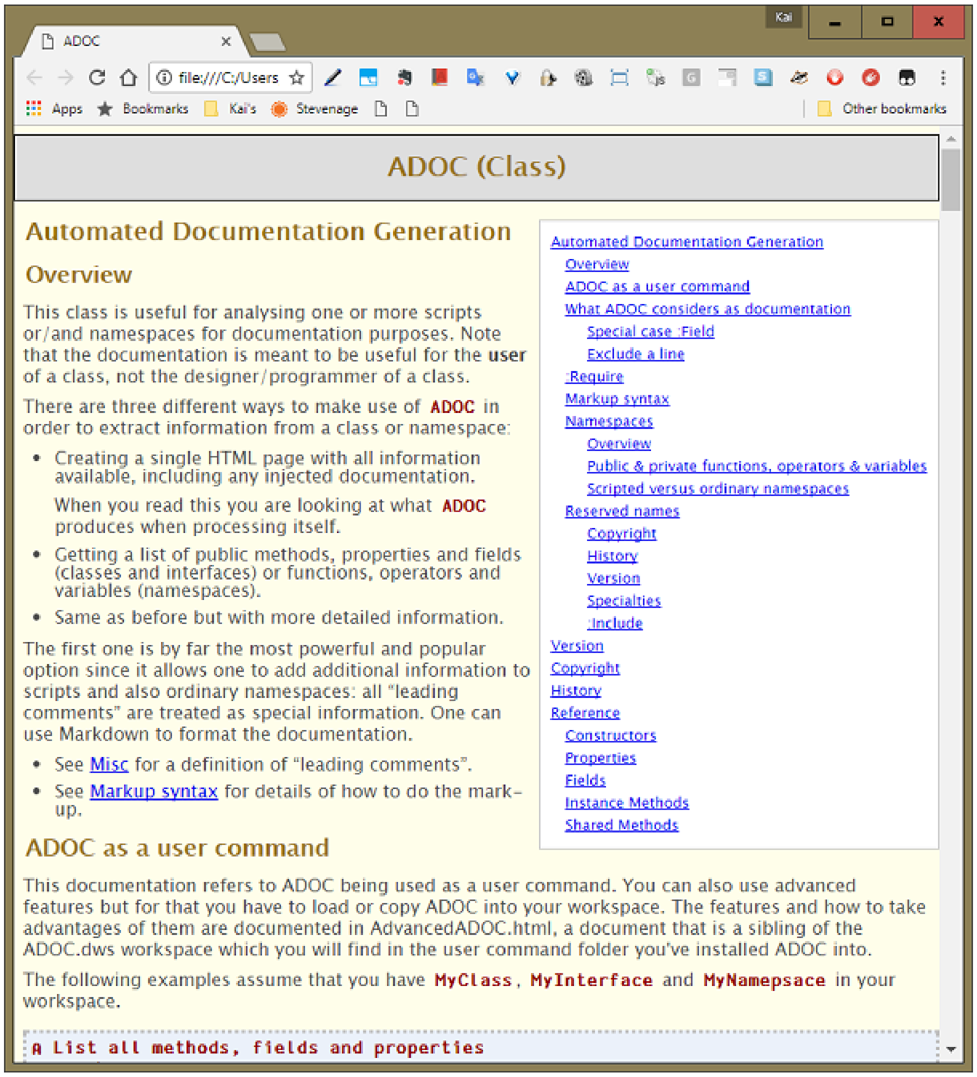
All parameters can be changed by setting variables in the namespace returned by CreateParms which can be passed as right argument to the main function Process. For defaults just pass an empty vector to Process

Using ADOC to browse the HandleError class

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

Getting help

To get basic information enter `]?adoc`. For some more details enter `]??adoc`. In order to get the full picture enter `]???adoc`. The underlying `ADOC` class then processes itself and creates an HTML page with detailed information.



ADOC (Class)

Automated Documentation Generation

Overview

This class is useful for analysing one or more scripts or/and 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 namespace:

- Creating a single HTML page with all information available, including any injected documentation.

  When you read this you are looking at what **ADOC** produces when processing itself.
- Getting a list of public methods, properties and fields (classes and interfaces) or functions, operators and variables (namespaces).
- Same as before but with more detailed information.

The first one is by far the most powerful and popular option since it allows one to add additional information to scripts and also ordinary namespaces: all "leading comments" are treated as special information. One can use Markdown to format the documentation.

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

ADOC as a user command

This documentation refers to ADOC being used as a user command. You can also use advanced features but for that you have to load or copy ADOC into your workspace. The features and how to take advantages of them are documented in AdvancedADOC.html, a document that is a sibling of the ADOC.dws workspace which you will find in the user command folder you've installed ADOC into.

The following examples assume that you have `MyClass`, `MyInterface` and `MyNamepsace` in your workspace.

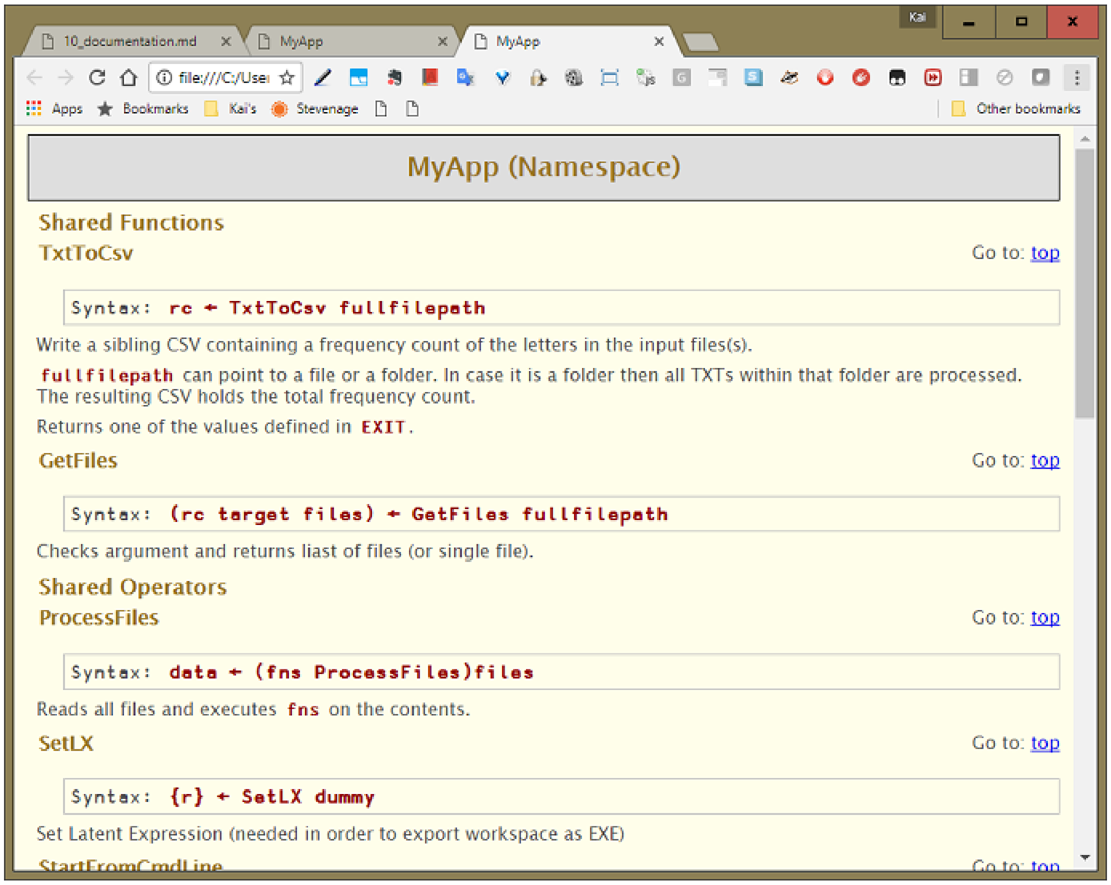
```
⍝ List all methods, fields and properties
```

ADOC's own documentation

9.6 ADOC for MyApp

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

```
]ADoc #.MyApp
```



MyApp (Namespace)

Shared Functions

TxtToCsv Go to: top

```
Syntax: rc ← TxtToCsv fullfilepath
```

Write a sibling CSV containing a frequency count of the letters in the input files(s).

`fullfilepath` can point to a file or a folder. In case it is a folder then all TXTs within that folder are processed. The resulting CSV holds the total frequency count.

Returns one of the values defined in `EXIT`.

GetFiles Go to: top

```
Syntax: (rc target files) ← GetFiles fullfilepath
```

Checks argument and returns liast of files (or single file).

Shared Operators

ProcessFiles Go to: top

```
Syntax: data ← (fns ProcessFiles)files
```

Reads all files and executes `fns` on the contents.

SetLX Go to: top

```
Syntax: {r} ← SetLX dummy
```

Set Latent Expression (needed in order to export workspace as EXE)

StartFromCmdLine Go to: top

Using ADOC to browse the MyApp namespace

ADOC has found and displayed all the functions within the `MyApp` namespace. If `MyApp` would contain any operators and/or variables you would find them in the document as well.

We can improve this in a number of ways. Time for a new version of MyApp. Make a copy of `Z:\code\v08` as `Z:\code\v09`.

Leading comments: basic information

First we edit the top of the script to follow ADOC's conventions:

```
:Namespace MyApp
    ⍝ Counting letter frequencies in text.\\
    ⍝ Can do one of:
    ⍝ * calculate the letters in a given document.
    ⍝ * calculate the letters in all documents in a given folder.
    ⍝
    ⍝ Sample application used by the Dyalog Cookbook.\\
    ⍝ Authors: Kai Jaeger & Stephen Taylor.
    ⍝ For more details see <http://cookbook.dyalog.com>
...
```

Public interface

Next we specify which functions we want to be included in the document: not all but just those that are designed to be called from the outside. In a class those are called “Public interface”, and it’s easy to see why.

For classes ADOC can work out what’s public and what isn’t due to the `Public Access` statements. For namespaces there is no such mechanism. We just learned that by default ADOC considers all functions and operators as well as all variables public, but it also offers a mechanism to reduce this list to what’s really public. For that ADOC looks for a function `Public`. It may return an empty vector (nothing is public at all) or a list of names. This list would define what is public.

Let’s define the public functions at the bottom of the script:

```
...
∇ r←Public
  r←'StartFromCmdLine' 'TxtToCsv' 'SetLX'
∇
:EndNamespace
```

Reserved names

ADOC honours five functions in a special way if they exist: `Copyright`, `History`, `Version`, `Public` and `ADOC_Doc`. If they exist when they (or rather their results) will be treated in a special way.

Version

If `Version` is niladic and returns a three-item vector then this vector is expected to be:

- Name
- Version number
- Version data

These pieces of information are then integrated into the document.

Copyright

If `Copyright` is niladic and returns either a simple text vector or a vector of text vectors then these pieces of information are then integrated accordingly into the document.

History

`History` is expected to be a niladic function that does not return a result. Instead it should carry comments with information about the history of the script.

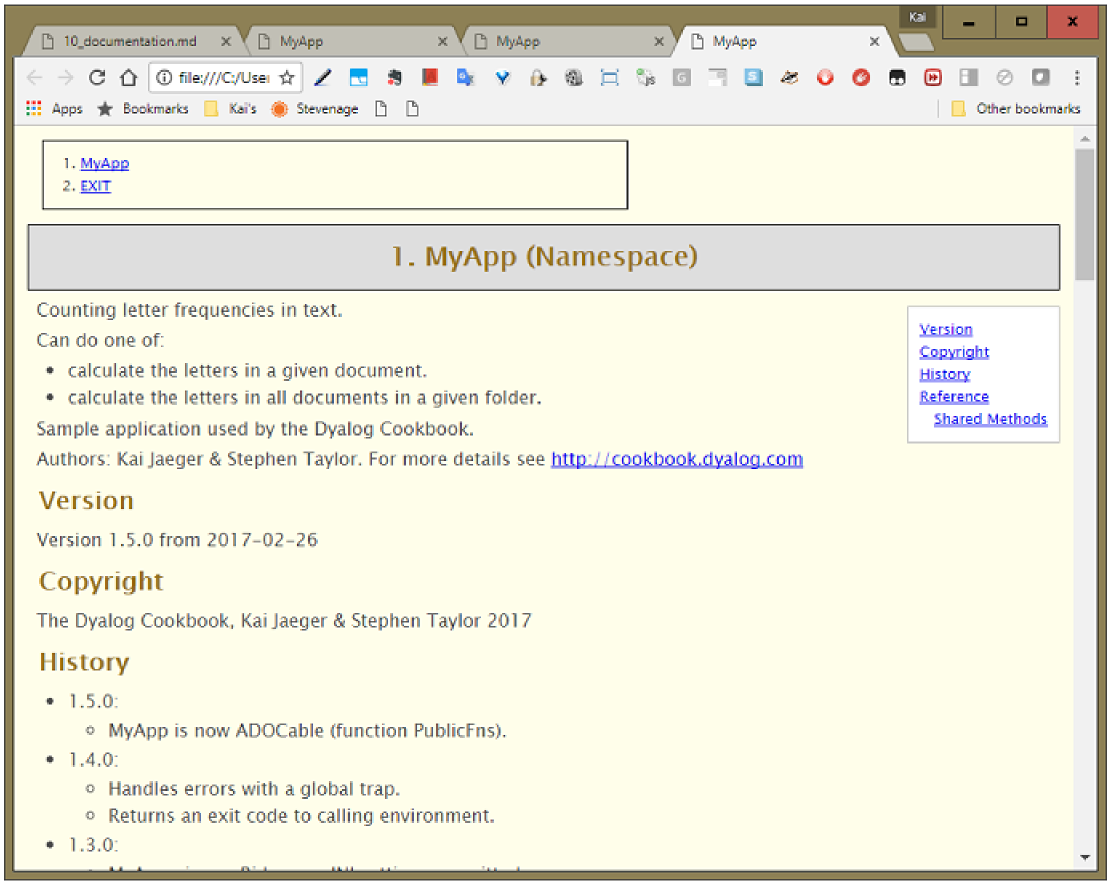
We had already a function `Version` in place, and so far we’ve added comments regarding the different versions to it. Those should go into `History` instead. Therefore we reaplace the existing `Version` function by these three functions:

```
∇ Z←Copyright
  Z←'The Dyalog Cookbook, Kai Jaeger & Stephen Taylor 2017'
∇

∇ r←Version
  r←(⍕⎕THIS)'1.5.0' 'YYYY-MM-DD'
∇

∇ History
  ⍝ * 1.5.0:
  ⍝   * MyApp is now ADOCable (function Public.
  ⍝ * 1.4.0:
  ⍝   * Handles errors with a global trap.
  ⍝   * Returns an exit code to calling environment.
  ⍝ * 1.3.0:
  ⍝   * MyApp gives a Ride now, INI settings permitted.
  ⍝ * 1.2.0:
  ⍝   * The application now honours INI files.
  ⍝ * 1.1.0:
  ⍝   * Can now deal with non-existent files.
  ⍝   * Logging implemented.
  ⍝ * 1.0.0
  ⍝   * Runs as a stand-alone EXE and takes parameters from the command line.
∇
```

This gives us more prominent copyright and version notices as well as information about the most recent changes. Note that `History` is not expected to carry a history of all changes but rather the most recent ones.

Finally we need to address the problem that the variables inside `EXIT` are essential for using `MyApp`; they should be part of the documentation. ADOC has ignored the namespace EXIT but we can change this by specifying it explicitly:

```
    ]ADoc #.MyApp #.MyApp.EXIT
```



1. MyApp
2. EXIT

1. MyApp (Namespace)

Counting letter frequencies in text.

Can do one of:

- calculate the letters in a given document.
- calculate the letters in all documents in a given folder.

Sample application used by the Dyalog Cookbook.

Authors: Kai Jaeger & Stephen Taylor. For more details see http://cookbook.dyalog.com

Version
Copyright
History
Reference
Shared Methods

Version

Version 1.5.0 from 2017-02-26

Copyright

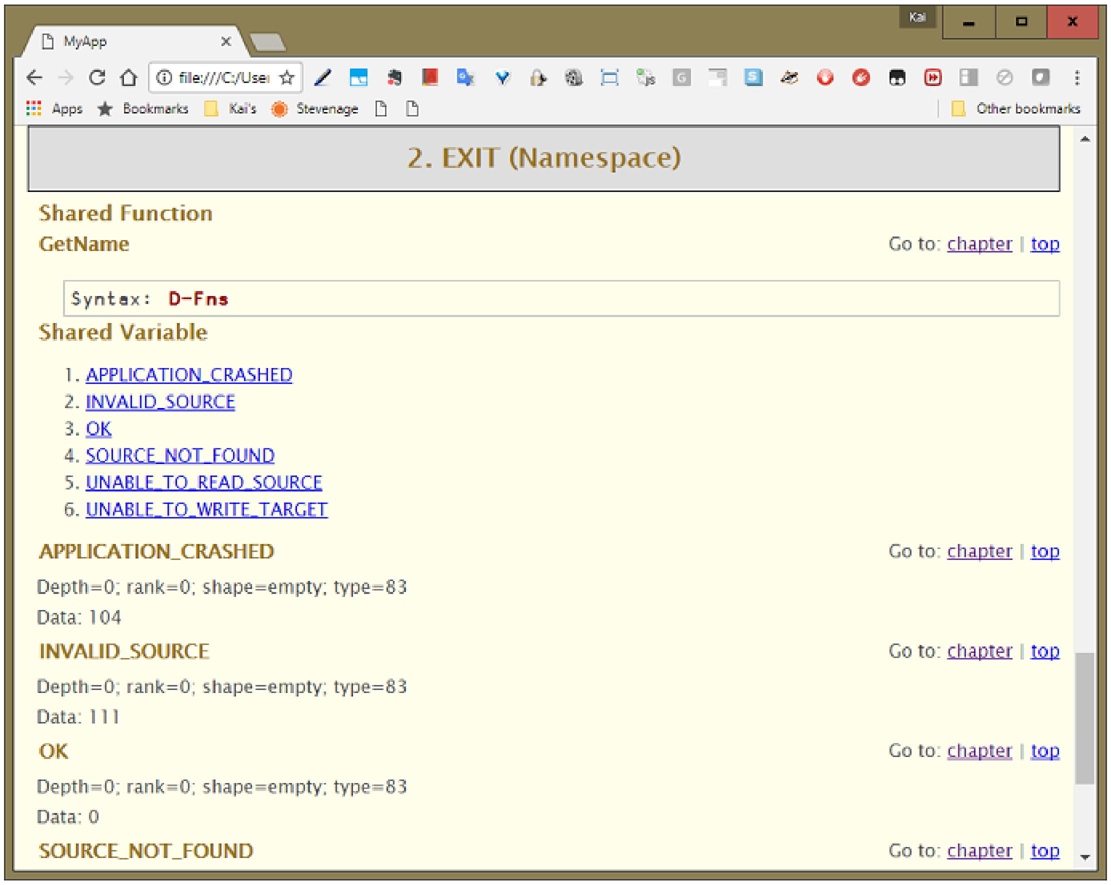
The Dyalog Cookbook, Kai Jaeger & Stephen Taylor 2017

History

- 1.5.0:
  - MyApp is now ADOCable (function PublicFns).
- 1.4.0:
  - Handles errors with a global trap.
  - Returns an exit code to calling environment.
- 1.3.0:

Browsing the revised MyApp namespace

When you scroll down (or click at “Exit” in the top-left corner) then you get to the part of the document where `EXIT` is documented:



2. EXIT (Namespace)

Shared Function

GetName — Go to: chapter | top

```
Syntax:  D-Fns
```

Shared Variable

1. APPLICATION_CRASHED
2. INVALID_SOURCE
3. OK
4. SOURCE_NOT_FOUND
5. UNABLE_TO_READ_SOURCE
6. UNABLE_TO_WRITE_TARGET

APPLICATION_CRASHED — Go to: chapter | top

Depth=0; rank=0; shape=empty; type=83

Data: 104

INVALID_SOURCE — Go to: chapter | top

Depth=0; rank=0; shape=empty; type=83

Data: 111

OK — Go to: chapter | top

Depth=0; rank=0; shape=empty; type=83

Data: 0

SOURCE_NOT_FOUND — Go to: chapter | top

Browsing the revised MyApp namespace

Public

This function was already discussed; see “Public interface”.

ADOC_Doc

There is one more reserved name: `ADOC_Doc`. This is useful when you want to document an unscripted (ordinary) namespace. Just add a niladic function carrying comments that returns no or a shy resul. Those comments are then processed in exactly the same way leading comments in scripts are processed.

That will do for now.

10. 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

- add the automated execution of test cases to the DYAPP.
- create a "Make" utility that allows us to create everything thats needed for what will finally be shipped to the customer.

At first glance you might think we can get away with splitting the DYAPP into two different versions, one for development and one for producing the final version of the EXE, but there will be tasks we cannot carry out with this approach. Examples are:

- Currently we depend on whatever version of Dyalog is associated with DYAPPs. We need an explicit way to define that version, even if for the time being there is just one version installed on our machine.
- We might want to convert any Markdown documents – like README.MD – into HTML documents. While the MD is the source, the HTML will become part of the final product.
- We need to make sure that the help system – which we will introduce soon – is properly compiled and configured by the "make" utility.
- Soon we need an installer that produces an EXE we can send to the customer for installing the software.

We resume, as usual, by saving a copy of `Z:\code\v09` as `Z:\code\v10`. Now delete `MyApp.exe` from `Z:\code\v10`: from now on we will create the EXE somewhere else.

10.1 The development environment

`MyApp.dyapp` does not need many changes, it comes with everything that's needed for development. The only thing we add is to execute the test cases automatically. Well, almost automatically. Ideally we should always make sure that all test cases pass when we call it a day, but sometimes that is just not possible due to the amount of work involved. In such cases it might or might not be sensible to execute the test cases before you start working: in case you *know* they will fail and there are *many* of them there is no point in wasting computer ressources and your time, so we better ask.

For that we are going to have a function `YesOrNo` which is very simple and straightforward: the right argument (`question`) is printed to the session and then the user might answer that question. If she does not enter one of: "YyNn" the question is repeated. If she enters one of "Yy" a `1` is returned, otherwise a `0`. Since we use this to ask ourself (or any other programmer) the function does not have to be bullet proof; that's why we allow `¯1↑⍞`.

But where exactly should this function go? Though it is helpful it has no part in our final application. Therefore we put it into a new script called `DevHelpers`. We also add a function `RunTests` to this new script:

```
:Namespace DevHelpers

∇ {r}←RunTests forceFlag
⍝ Runs the test cases in debug mode, either in case the user wants to
⍝ or if `forceFlag` is 1.
  r←''
  :If forceFlag
  :OrIf YesOrNo'Would you like to execute all test cases in debug mode?'
      r←#.Tests.RunDebug 0
  :EndIf
∇

∇ flag←YesOrNo question;isOkay;answer
  isOkay←0
  ⎕←(⎕PW-1)⍴'-'
  :Repeat
      ⍞←question,' (y/n) '
      answer←¯1↑⍞
      :If answer∊'YyNn'
          isOkay←1
          flag←answer∊'Yy'
      :EndIf
  :Until isOkay
∇

:EndNamespace
```

We add a line to the bottom of `MyApp.dyapp`:

```
...
Run #.Tester.EstablishHelpersIn #.Tests
Run #.DevHelpers.RunTests 0
```

Now a developer who double-clicks the DYAPP in order to assemble the workspace will always be reminded of running all test cases before she starts working on the application. Experience tells us that this is a good thing.

10.2 MyApp.dyalog

One minor thing needs our attention: because we create `MyApp.exe` now in a folder `MyApp` simply setting `⎕WSID` to `MyApp` does not do any more. Therefore we need to make a change to the `StartFromCmdLine` function in `MyApp.dyalog`:

```
...
∇ {r}←StartFromCmdLine arg;MyLogger;Config;rc;⎕TRAP
   ⍝ Needs command line parameters, runs the application.
      r←⍬
      ⎕TRAP←#.HandleError.SetTrap ⍬
      ⎕WSID←⊃{⍵/⍨~'='∊¨⍵}{⍵/⍨'-'≠⊃¨⍵}1↓2⎕nq # 'GetCommandLineArgs'
      #.FilesAndDirs.PolishCurrentDir
...
```

This change makes sure that the `⎕WSID` will be correct. Under the current circumstances it will be `MyApp\MyApp.dws`.

Note that we access `GetCommandLineArgs` as a function call with `⎕NQ` rather than referring to `#.GetCommandLineArgs`; over the years that has proven to be more reliable.

10.3 "Make" the application

In most programming languages the process of compiling the source code and putting together an application is done by a utility that's called "Make"; therefore we use the same term.

At first sight it might seem that we can get away with a reduced version of `MyApp.dyapp`, but that is not quite true. Soon we will discuss how to add a help system to our application. We must then make sure that the help system is compiled properly when the application is assembled. Later even more tasks will come up. Conclusion: we cannot do this with a DYAPP; we need more flexibility.

More complex scenarios

In a more complex application than ours you might prefer a different approach. Using an INI file for this is not a bad idea: it gives you way more freedom in defining all sorts of things while a DYAPP allows you to define just the modules to be loaded, and to execute some code.

Also, if you have not one but quite a number of applications to deal with it is certainly not a bad idea to implement your own generalized user command like `]runmake`.

`Execute`, `Tester` and `Tests` have no place in the finished application, and we don't need to establish the test helpers either.

We are going to create a DYAPP file `Make.dyapp` that performs the "Make". However, if you want to make sure that you can specify explicitly the version of Dyalog that should run this DYAPP rather than relying on what happens to be associated with the file extensions DWS, DYALOG and DYAPP at the time you double-click it then you need a batch file that starts the correct version of Dyalog. Create such a batch file as `Make.bat`. This is the contents:

```
"C:\Program Files\Dyalog\Dyalog APL{yourPreferredVersion}\Dyalog.exe" DYAPP="%~dp0Make.dyapp"
@echo off
if NOT ["%errorlevel%"]==["0"] (
    echo Error %errorlevel%
    pause
    exit /b %errorlevel%
)
```

Of course you need to make amendments so that it is using the version of Dyalog of your choice. If it is at the moment what happens to run a DYAPP on a double-click then this will give you the correct path:

```
'"',(⊃#.GetCommandLineArgs),'"'
```

You might want to add other parameters like `MAXWS=128M` (or `MAXWS=6G`) to the BAT file.

Notes:

- The expression `%∼dp0` in a batch file will give you the full path – with a trailing `\` – of the folder that hosts the batch file. In other words, `"%∼dp0Make.dyapp"` would result in a full path pointing to `MyApp.dyapp`, no matter where that is as long as it is a sibling of the BAT file. You *must* specify a full path because when the interpreter tries to find the DYAPP, the current directory is where the EXE lives, *not* where the BAT file lives.
- Checking `errorlevel` makes sure that in case of an error the batch file shows the return code and then pauses. That gets us around the nasty problem that when you double-click a BAT file, you see a black windows popping up for a split of a second and then it's gone, leaving you wondering whether it has worked alright or not. Now when an error occurs it will pause. In addition it will pass the value of `errorlevel` as return code of the batch script.

  However, this technique is suitable only for scripts that are supposed to be executed by a WCU [1]; you don't want to have a pause in scripts that are called by other scripts.

The current directory

For APLers, the current directory (sometimes called "working directory") is, when running under Windows, a strange animal. In general, the current directory is where "the application" lives. That means that if you start an application `C:\Program Files\Foo\Foo.exe` then for the application "Foo" the current directory will be `C:\Program Files\Foo`.

That's fine except that for APLers "the application" is *not* the DYALOG.EXE, it's the workspace, whether it was loaded from disk or assembled by a DYAPP. When you double-click `MyApp.dyapp` then the interpreter changes the current directory for you: it's where the DYAPP lives, and that's fine from an APL application programmer's point of view.

The same holds true when you double-click a workspace but it is *not* true when you *load* a workspace: the current directory remains what it was before, and that's where the Dyalog EXE lives. Therefore it's probably not a bad idea to change the current directory yourself *at the earliest possible stage* after loading a workspace: call `#.FilesAndDirs.PolishCurrentDir` and your are done, no matter what the circumstances are. One

[1]Worst Case User, also known as Dumbest Assumable User (DAU).

of the authors is doing this for roughly 20 years now, and it has solved several problems without introducing new ones.

Now we need to establish the `Make.dyapp` file:

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

Load Make
Run #.Make.Run 1
```

The upper part (until the blank line) is identical with `MyApp.dyapp` except that we don't load the stuff that's only needed during development. We then load a script `Make` and finally we call `Make.Run`. That's how `Make` looks at this point:

```
:Class Make
⍝ Puts the application `MyApp` together:
⍝ 1. Remove folder `DESTINATION\` in the current directory
⍝ 2. Create folder `DESTINATION\` in the current directory
⍝ 3. Copy icon to `DESTINATION\`
⍝ 4. Copy the INI file template over to `DESTINATION`
⍝ 5. Creates `MyApp.exe` within `DESTINATION\`
    ⎕IO←1 ⋄ ⎕ML←1

    DESTINATION←'MyApp'

    ∇ {filename}←Run offFlag;rc;en;more;successFlag;F;U;msg
      :Access Public Shared
      F←##.FilesAndDirs ⋄ U←##.Utilities
      (rc en more)←F.RmDir DESTINATION
      U.Assert 0≠rc
      successFlag←'Create!'F.CheckPath DESTINATION
      U.Assert 1≠successFlag
      (successFlag more)←2↑'images'F.CopyTree DESTINATION,'\images'
      U.Assert 1≠successFlag
      (rc more)←'MyApp.ini.template'F.CopyTo DESTINATION,'\MyApp.ini'
      U.Assert 0≠rc
      Export'MyApp.exe'
      filename←DESTINATION,'\MyApp.exe'
      :If offFlag
          ⎕OFF
      :EndIf
      ∇
:EndClass
```

The function `Assert` does not exist yet in `Utilities`:

```
:Namespace Utilities
      map←{
          (,2)≢⍴⍺:'Left argument is not a two-element vector'⎕SIGNAL 5
          (old new)←⍺
          nw←∪⍵
          (new,nw)[(old,nw)⍳⍵]
      }
      Assert←{
          ⍵:.
          1:r←⍬
      }
:EndNamespace
```

Note that `Assert` executes a full stop in case `⍵` is `1` but returns `⍬` as a shy (!) result in case ⍵ is `0`. This is an easy way to make the calling function stop when something goes wrong. There is no point in doing anything but stopping the code from continuing since it is called by a programmer, and when it fails she wants to investigate straight away. And things can go wrong quite easily; for example, the attempt to remove `DESTINATION` may fail simply because somebody is looking with the Windows Explorer into `DESTINATION` at the same time.

First we create the folder `DESTINATION` from scratch and then we copy everything that's needed to the folder `DESTINATION`: the application icon and the INI file. Whether the function executes `⎕OFF` or not depends on the right argument `offFlag`. Why that is needed will become apparent soon.

We don't copy `MyApp.ini` into `DESTINATION` but `MyApp.ini.template`; therefore we must create this file: copy `MyApp.ini` to `MyApp.ini.template` and then check its settings: in particular these settings are important:

```
...
[Config]
Debug       = ¯1   ; 0=enfore error trapping; 1=prevent error trapping;
Trap        = 1    ; 0 disables any :Trap statements (local traps)
ForceError  = 0    ; 1=let TxtToCsv crash (for testing global trap handling)
...
[Ride]
Active      = 0
...
```

Those might well get changed in `MyApp.ini` while working on the project, so we make sure that we get them set correctly in `MyApp.ini.template`.

However, that leaves us vulnerable to another problem: imagine we introduce a new section and/or a new key and forget to copy it over to the template. In order to avoid this we add a test case to `Tests`:

```
    ∇ R←Test_misc_01(stopFlag batchFlag);⎕TRAP;ini1;ini2
      ⍝ Check whether MyApp.ini and MyApp.ini.template have the same sections and keys
      ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
      R←∆Failed
      ini1←⎕NEW ##.IniFiles(,⊂'MyApp.ini')
      ini2←⎕NEW ##.IniFiles(,⊂'MyApp.ini.template')
      →PassesIf ini1.GetSections{(∧/⍺∊⍵)∧(∧/⍵∊⍺)}ini2.GetSections
      →PassesIf(ini1.Get ⍬ ⍬)[;2]{(∧/⍺∊⍵)∧(∧/⍵∊⍺)}(ini2.Get ⍬ ⍬)[;2]
      R←∆OK
    ∇
```

The test simply checks whether the two INI files have the same sections and the same keys; that's sufficient to notify us in case we forgot something.

`Run` then calls `Export`, a private function in the `Make` class that does not yet exist:

```
...
    ∇ {r}←{flags}Export exeName;type;flags;resource;icon;cmdline;try;max;success
    ⍝ Attempts to export the application
      r←⍬
      flags←##.Constants.BIND_FLAGS.RUNTIME{⍺←0 ⋄ 0<⎕NC ⍵:⍎⍵ ⋄ ⍺}'flags'
      max←50
      type←'StandaloneNativeExe'
      icon←F.NormalizePath DESTINATION,'\images\logo.ico'
      resource←cmdline←''
      success←try←0
      :Repeat
          :Trap 11
              2 ⎕NQ'.' 'Bind',(DESTINATION,'\',exeName)type flags resource icon cmdline
              success←1
          :Else
              ⎕DL 0.2
          :EndTrap
      :Until success∨max<try←try+1
      :If 0=success
          ⎕←'*** ERROR: Failed to export EXE to ',DESTINATION,'\',exeName,' after ',(⍕try),' tries.'
          . ⍝ Deliberate error; allows investigation
      :EndIf
    ∇
:EndClass
```

`Export` automates what we've done so far by calling the "Export" command from the "File" menu. In case the "Bind" method fails it tries up to 50 times before giving up. This is because from experience we know that depending on the OS and the machine and God knows what else sometimes the command fails several times before it finally succeeds.

The "Bind" method

Not that `Bind` is in version 16.0 not an official method. However, it's there, it works, and we expect it to become an official method in alter version of Dyalog.

We specified `##.Constants.BIND_FLAGS.RUNTIME` as a default for `flags`, but that does not exist yet, so we add it to the `Constants` namespace:

```
:Namespace Constants
...
    :EndNamespace
    :Namespace BIND_FLAGS
        BOUND_CONSOLE←2
        RUNTIME←8
    :EndNamespace
:EndNamespace
```

Double-click `Make.dyapp`: a folder `MyApp` should appear in `Z:\code\v10` with, among other files, `MyApp.exe`.

10.4 The tests

Now that we have a way to automatically assemble all the necessary files required by our application we need to amend our tests. Double-click `MyApp.dyapp`. You don't need to execute the test cases right now because we are going to change them.

We need to make a few changes:

```
:Namespace Tests
    ⎕IO←1 ⋄ ⎕ML←1
    ∇ Initial;list;rc
      U←##.Utilities ⋄ F←##.FilesAndDirs ⋄ A←##.APLTreeUtils
      ∆Path←F.GetTempPath,'\MyApp_Tests'
      F.RmDir ∆Path
      'Create!'F.CheckPath ∆Path
      list←⊃F.Dir'..\..\texts\en\*.txt'
      rc←list F.CopyTo ∆Path,'\'
      :If ~R←0∧.=⊃rc
          ⎕←'Could not create ',∆Path
      :EndIf
      ⎕SE.UCMD'Load ',F.PWD,'\Make.dyalog -target=#'
      #.Make.Run 0
    ∇
 ...
:EndNamespace
```

Notes: `Initial` …

- … loads the script `Make.dyalog` into `#`.
- … runs the function `Make.Run`. The `0` provided as right argument tells `Make.Run` to *not* execute `⎕OFF`, something we would not appreciate at this stage.

10.5 Workflow

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

1. Launch `MyApp.dyapp` and check the test results.
2. Fix any errors and rerun `#.Tests.Run` until it's fine. If you edit the test themselves, either rerun

   ```
   `#.Tester.EstablishHelpersIn #.Tests`
   ```

   or simply close the session and relaunch `MyApp.dyapp`.

11. Providing help

Users expect applications to provide help in one way or another. One option is to provide the help as a hyper text system. Under Windows, CHM files are the standard way to provide such help. There are powerful applications available that can assist you in providing help; HelpAndManual [1] is just an example.

However, we take a different approach here: rather than using any third-party software we use `Markdown2Help` from the APLTree library. That allows us to create a help system that…

- offers pretty much the same functionality as CHM.
- allows us to keep the help close to the code.

This is the simplest way to create a help system, and it allows you to run the help system from within your application in order to view either its start page or a particular page as well as viewing the help system without running your application at all.

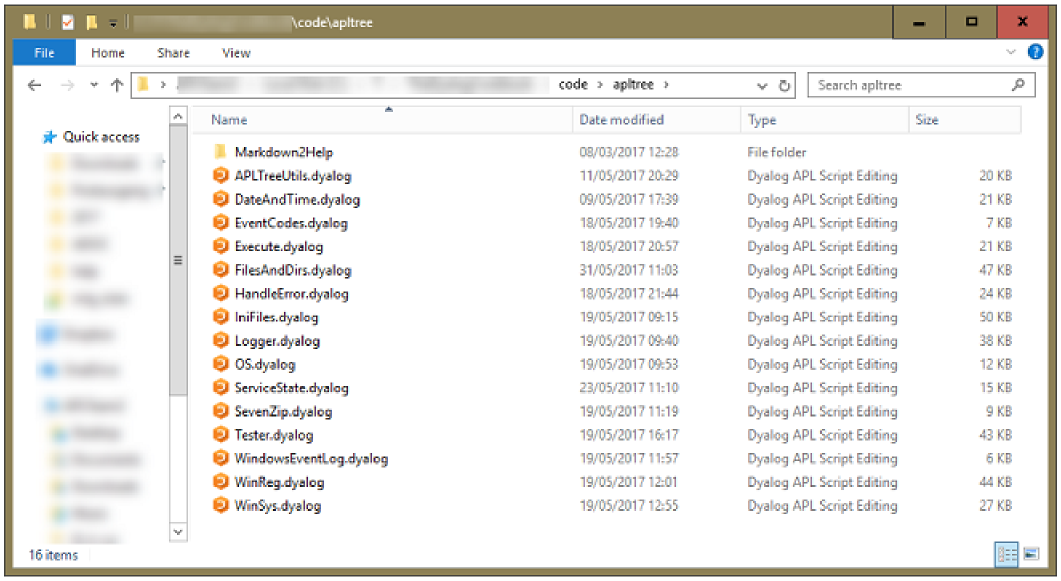
While CHM files are Windows specific, `Markdown2Help` allows you to export a help system as a web page that can be displayed with any modern browser. That makes it OS-independent. We'll discuss later how to do this.

11.1 Getting ready

It's time to save a copy of `Z:\code\v10` as `Z:\code\v11`.

In order to use `Markdown2Help` you need to download it from http://download.aplwiki.com/. We suggest creating a folder `Markdown2Help` within the folder `Z:\code\APLTree`. Copy the contents of the zip file you've just downloaded into `Z:\code\APLTree\Markdown2Help`:

[1] http://www.helpandmanual.com/

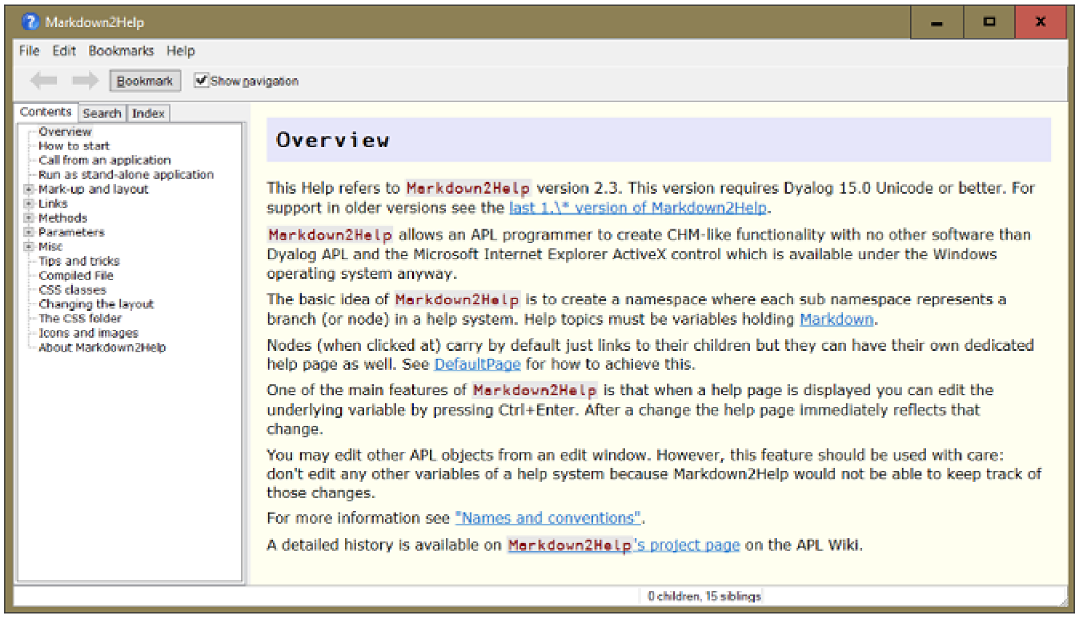


Name	Date modified	Type	Size
Markdown2Help	08/03/2017 12:28	File folder	
APLTreeUtils.dyalog	11/05/2017 20:29	Dyalog APL Script Editing	20 KB
DateAndTime.dyalog	09/05/2017 17:39	Dyalog APL Script Editing	21 KB
EventCodes.dyalog	18/05/2017 19:40	Dyalog APL Script Editing	7 KB
Execute.dyalog	18/05/2017 20:57	Dyalog APL Script Editing	21 KB
FilesAndDirs.dyalog	31/05/2017 11:03	Dyalog APL Script Editing	47 KB
HandleError.dyalog	18/05/2017 21:44	Dyalog APL Script Editing	24 KB
IniFiles.dyalog	19/05/2017 09:15	Dyalog APL Script Editing	50 KB
Logger.dyalog	19/05/2017 09:40	Dyalog APL Script Editing	38 KB
OS.dyalog	19/05/2017 09:53	Dyalog APL Script Editing	12 KB
ServiceState.dyalog	23/05/2017 11:10	Dyalog APL Script Editing	15 KB
SevenZip.dyalog	19/05/2017 11:19	Dyalog APL Script Editing	9 KB
Tester.dyalog	19/05/2017 16:17	Dyalog APL Script Editing	43 KB
WindowsEventLog.dyalog	19/05/2017 11:57	Dyalog APL Script Editing	6 KB
WinReg.dyalog	19/05/2017 12:01	Dyalog APL Script Editing	44 KB
WinSys.dyalog	19/05/2017 12:55	Dyalog APL Script Editing	27 KB

Download target

Within that folder you will find a workspace `Markdown2Help` (from which we are going to copy the module) and a folder "help". This folder contains in turn a sub folder "files" (which contains `Markdown2Help`'s own help system) and the file `ViewHelp.exe`; that is the external viewer you need in case you want to view your help system independently from your application.

Double-click `ViewHelp.exe` in order to see `Markdown2Help2`'s own help system:



Overview

This Help refers to `Markdown2Help` version 2.3. This version requires Dyalog 15.0 Unicode or better. For support in older versions see the last 1.* version of Markdown2Help.

`Markdown2Help` allows an APL programmer to create CHM-like functionality with no other software than Dyalog APL and the Microsoft Internet Explorer ActiveX control which is available under the Windows operating system anyway.

The basic idea of `Markdown2Help` is to create a namespace where each sub namespace represents a branch (or node) in a help system. Help topics must be variables holding Markdown.

Nodes (when clicked at) carry by default just links to their children but they can have their own dedicated help page as well. See DefaultPage for how to achieve this.

One of the main features of `Markdown2Help` is that when a help page is displayed you can edit the underlying variable by pressing Ctrl+Enter. After a change the help page immediately reflects that change.

You may edit other APL objects from an edit window. However, this feature should be used with care: don't edit any other variables of a help system because Markdown2Help would not be able to keep track of those changes.

For more information see "Names and conventions".

A detailed history is available on `Markdown2Help`'s project page on the APL Wiki.

Markdown2Help's Help

By default `ViewHelp.exe` expects to find a folder `files` as a sibling of itself, and it assumes that this folder contains a help system.

Specify help folder and help page

You can change the folder `ViewHelper.exe` expects to host the help system by specifying a command line parameter "helpFolder":

```
ViewHelp.exe -helpfolder=C:\Foo\Help
```

You can also tell `ViewHelper.exe` to put a particular help page on display rather than the default page:

```
ViewHelp.exe -page=Sub.Foo
```

However, all these details are discussed in `Markdown2Help`'s own help system.

`Markdown2Help` is an ordinary (non-scripted) namespace. We therefore need to copy it from its workspace. We also need the script `MarkAPL` which is used to convert the help pages (which are written in Markdown) to HTML. You know by now how to download scripts from the APLTree library. Modify `MyApp.dyapp` so that it loads the module `MarkAPL` and also copies `Markdown2Help`:

```
...
Load ..\AplTree\Execute
Load ..\AplTree\MarkAPL
Run 'Markdown2Help' #.⎕CY '..\apltree\Markdown2Help\Markdown2Help.dws'
Load Tests
...
```

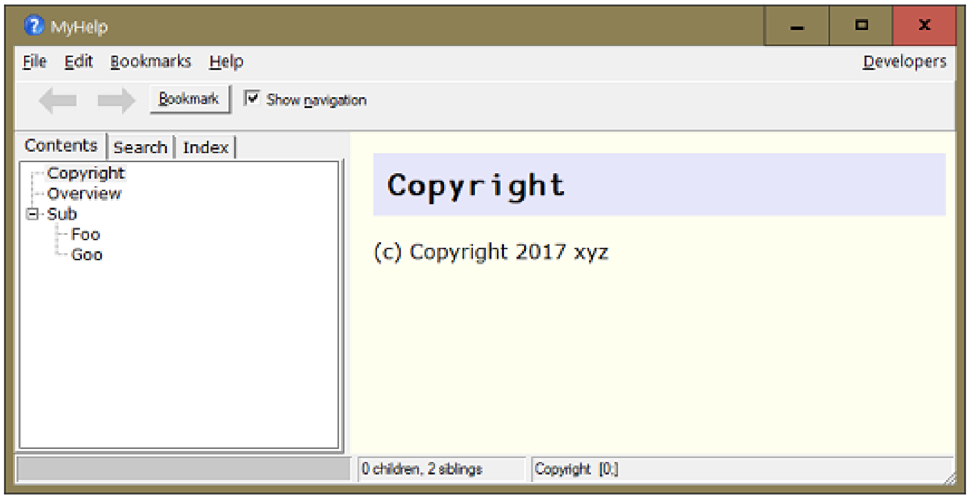
Double-click the DYAPP to get started.

11.2 Creating a new help system

`Markdown2Help` comes with a function `CreateStub` that will create a new help system for us. All we need to do is finding a good name which is not in use. In our case the obvious candidate is "MyHelp". We also want the help system to be managed by SALT, and for that we need to define a folder where all the help files are going to be saved. For that we call `CreateParms` and then specify that folder by setting the parameter `saltFolder`:

```
parms←#.Markdown2Help.CreateParms ⍬
parms.saltFolder←'Z:\code\v11\MyHelp'
parms.folderName←'Z:\code\v11\Help\Files'
parms #.Markdown2Help.CreateStub '#.MyHelp'
```

`CreateStub` will create some pages and a node (or folder) for us; that's what you should see:



Copyright

(c) Copyright 2017 xyz

0 children, 2 siblings

Copyright [0;]

Download target

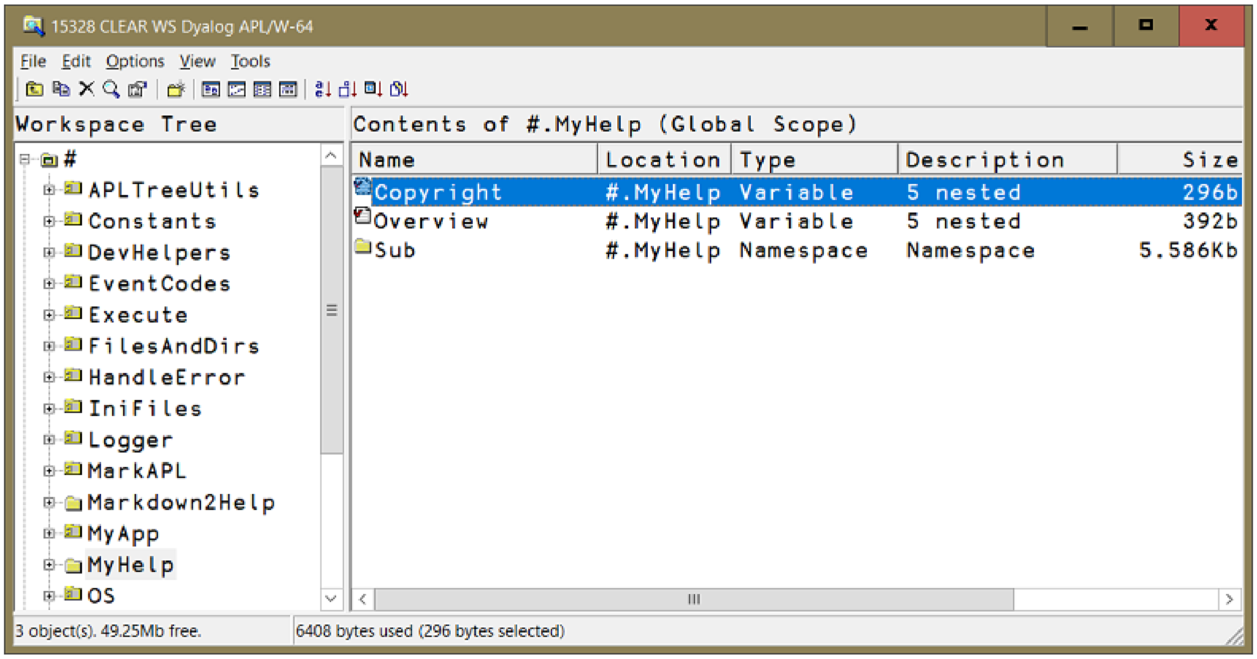
Notes:

- In our example the name of `saltFolder` and the help system are the same. That is not a requirement but certainly not a bad idea either.
- `folderName` is the folder the compiled help system will be saved into. That's the stuff that needs to be installed at your customers machine in order to be able to view the help system.
- The right argument must be a valid and unused APL name. `CreateStub` will create a namespace with that name for us.

  If a simple (not fully qualified) name is specified that namespace will be created in the namespace the function `CreateStub` was called from. Instead you can also specify a fully qualified name like `#.Foo.Goo.Help`. Note that `Foo` and `Goo` must both exist but `Help` must not.
- `CreateStub` will check the callback associated with the `Fix` event. If that happens to be a SALT function `CreateStub` will check the `saltFolder` parameter. If that's not empty the help system will use SALT for saving the nodes (namespaces), help pages (variables) and functions that resemble a help system.
- `CreateStub` will return a reference pointing to the help system but normally you don't need to assign that.
- When the help system is managed by SALT you will find a namespace `SALT_Var_Data` in the root. Ignore this; it is used by SALT to manage meta data.

11.3 Behind the scenes

In the workspace all nodes (in our case "MyHelp" and "Sub") are ordinary namespaces while the pages are variables. You can check with the Workspace Explorer:


15328 CLEAR WS Dyalog APL/W-64
File Edit Options View Tools
Workspace Tree
#
APLTreeUtils
Constants
DevHelpers
EventCodes
Execute
FilesAndDirs
HandleError
IniFiles
Logger
MarkAPL
Markdown2Help
MyApp
MyHelp
OS
Contents of #.MyHelp (Global Scope)
3 object(s). 49.25Mb free.
6408 bytes used (296 bytes selected)

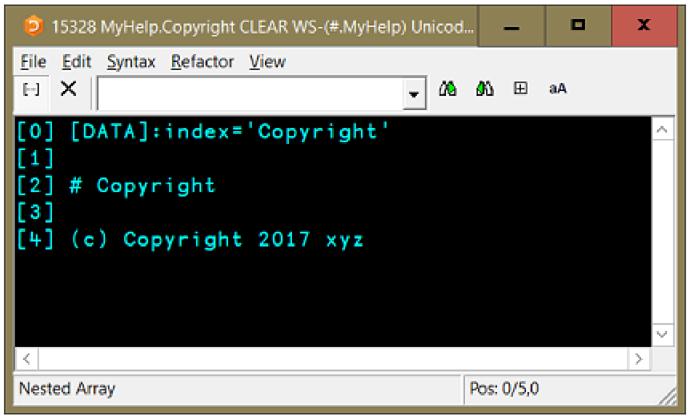
Name	Location	Type	Description	Size
Copyright	#.MyHelp	Variable	5 nested	296b
Overview	#.MyHelp	Variable	5 nested	392b
Sub	#.MyHelp	Namespace	Namespace	5.586Kb

The help system in the Workspace Explorer

This is the reason why the names of nodes and pages must be valid APL names. By default those names are shown in the help system as title in the tree, but if that is not good enough for you then there is of course a way to specify something different. We'll come back to this soon.

11.4 Editing a page

When you right-click on a page like "Copyright" and then select "Edit help page" from the context menu (pressing <Ctrl+Enter> will do the same) the APL editor opens and shows something similar to this:


15328 MyHelp.Copyright CLEAR WS-(#.MyHelp) Unicod...
File Edit Syntax Refactor View
Nested Array
Pos: 0/5,0

```
[0] [DATA]:index='Copyright'
[1]
[2] # Copyright
[3]
[4] (c) Copyright 2017 xyz
```

A help page in the editor

This is the definition of the help page in Markdown.

Notes:

- The first line specifies a key-value-pair (`[DATA]`). "index" is the key and "Copyright" is the value of that key. This is interpreted by `Markdown2Help` as an index entry.

  Note that this is not a Markdown feature but a `Markdown2Help` feature.
- `# Copyright` defines a header of level one. Every help page must have such a header.
- `(c) Copyright 2017 xyz` is a simple paragraph.

Make some changes, for example add another paragraph `Go to →[Overview]`, and then press <escape>. `Markdown2Help` takes your changes, converts the Markdown to HTML and shows the changed page straight away. This gives you an idea of how easy it actually is to change help pages. Adding, renaming and deleting help pages – and nodes – can be achieved via the context menu.

Note also that `→[Overview]` is a link. For the link to work "Overview" must be the name of an existing page. If the title of the page is different from the name, the title is going to be shown as link text in the help page.

Even if you are familiar with Markdown you should read `Markdown2Help`'s own help file before you start using `Markdown2Help` seriously. Some Markdown features are not supported by the help system, and internal links are implemented in a simplified way.

11.5 Changing title and sequence

Note that the "Copyright" page comes first. That's because by default the pages are ordered alphabetically. You can change this with a right-click on either the "Copyright" or the " Overview" page and then select "Manage ∆TopicProperties". After confirming that this is really what you want to do you will see something like this:

```
 ∆TopicProperties←{
⍝ This function is needed by the Markdown2Help system.
⍝ You can edit this function from the Markdown2Help GUI via the context menu.
⍝ *** NOTE:
⍝     Make only changes to this function that affect the explicit result.
⍝     Any other changes will eventually disappear because these functions are rebuild
⍝     under program control from their explicit result under certain circumstances.
⍝        This is also the reason why you should use the `active` flag to hide a topic
⍝     temporarily because although just putting a `⍝` symbol in front of its line
⍝     seems to have the same effect, in the long run that's not true because the
⍝     commented line will disappear in the event of a rebuild.
⍝ ----------------------------------
⍝ r gets a table with these columns:
⍝ [;0] namespace or function name.
⍝ [;1] caption in the tree view. If empty the namespace/fns name is taken.
⍝ [;2] active flag.
⍝ [;3] developmentOnly flag; 1=the corresponding node does not show in user mode.
     r←0 4⍴''
     r⍪←'Copyright' '' 1 0
     r⍪←'Overview' '' 1 0
     r⍪←'Sub' '' 1 0
     r
}
```

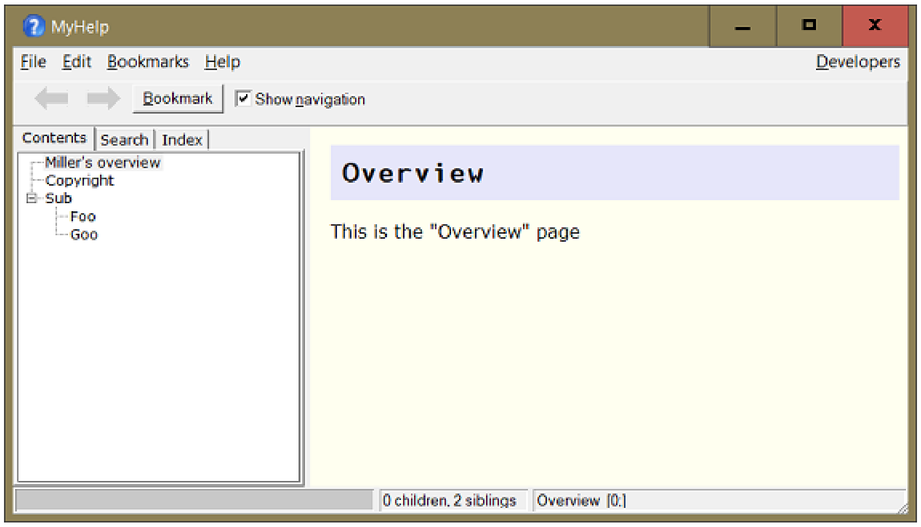
It's well worth reading the comments in this function.

You can specify a different sequence of the pages by simply changing the sequence in which the pages are added to `r`. Here we swap the position of "Copyright" and "Overview":

```
∆TopicProperties←{
    ...
    r←0 4⍴''
    r⍪←'Overview' 'Miller''s overview' 1 0
    r⍪←'Copyright' '' 1 0
    r⍪←'Sub' '' 1 0
    r
}
```

We have also changed the title of the "Overview" page to "Miller's overview". That's how you can specify a specific title to be shown instead of the name of the page.

After fixing the function the help system is re-compiled automatically; therefore our changes become visible immediately:



Overview

This is the "Overview" page

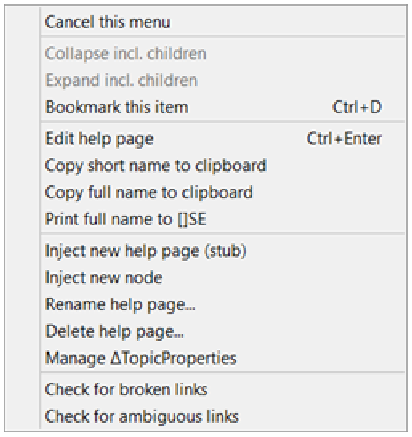
0 children, 2 siblings | Overview [0;]

The changed help system

What "compiling the help system" actually means is discussed soon.

11.6 More commands

The context menu offers plenty of commands. Note that the first three commands are always available. The other commands are useful for a developer (or shall we say help system author?) and are available only when the help system is running in a development version of Dyalog.



The context menu

As a developer you should have no problem mastering these commands.

11.7 Compiling the help system

"Compiling the help system" means to convert the pieces of information represented by the structure of the help system plus the variables holding Markdown plus the additional rules defined by any `∆TopicProperties` function into a single component file that contains the HTML generated from the Markdown plus some more pieces of information.

It's more than just converting Markdown to HTML. For example, the words of all pages are extracted, words like "and", "then", "it" etc. are removed (because searching for them does not make too much sense) and then the list is, together with the information to which page(s) they belong to, saved in a component. This allows `Markdown2Help` to provide a very fast search function. Actually the list is saved twice, once "as is" and once with all words lowercased: that speeds up any case insensitive search operations.

Without specifying a particular folder `Markdown2Help` would create a temporary folder and compile into that folder. It is better to define a permanent location because it means that the help system does not need to compile the Markdown into HTML over and over again whenever it is called. Such a permanent location is also the pre-condition for being able to put the help system on display with the external viewer, something you *must* do for obvious reasons when your help system is supposed to offer help on how to install your application.

Note that for converting the Markdown to HTML `Markdown2Help` needs the `MarkAPL` class, but once the help system has been compiled this class is not needed any more. Therefore the final version of your application would not need `MarkAPL`, and because `MarkAPL` comprises roughly 3,000 lines of code this is good news.

11.8 Manipulating the help system directly

What we actually mean by that is for example editing a variable with a double-click in the Workspace Explorer but also editing it with `)ED` from the session. Our advice: **don't!**

The reason is simple: when you change a help system via the context menu then all necessary steps are carried out for you. An example is when you have a `∆TopicProperties` function in a perticular node and you want to add a new help page to that node. You have to right-click on a page and select the "Inject new help page (stub)" command from the context menu. You will then be prompted for a valid name and finally the new help page is injected after the page you have clicked at. But there is more to it than just that: the page is also added for you to the `∆TopicProperties` function. That's one reason why you are advised to perform all changes via the context menu rather than manipulating the help system directly.

Maybe even more important: `Markdown2Help` also executes the necessary steps in order to keep the files and folders in `saltFolder` in sync with the help system *and* will automaticall re-compile the help system for you.

The only exception is when you change your mind about the structure of a help system. If that involves moving around namespaces or plenty of pages between namespaces then it is indeed better to do it in the Workspace Explorer and, when you are done, to check all the `∆TopicProperties` functions within your help system and finally recompile the help system; unless somebody implements drag-and-drop for the TreeView of the help system one day…

However, in that case you must make sure that the help system is saved properly. That means that you have to invoke the `SaveHelpSystemWithSalt` method yourself. You also nee to call the `Markdown2Help.CompileHelpFileInto` method in order to compile the help system from the source. Refer to `Markdown2Help`'s own help for details.

11.9 The "Developers" menu

In case the help system is running under a development version of Dyalog you have a "Developers" menu on the right side of the menubar. This offers a couple of commands that support you in keeping your help system healthy. We discuss just the most important ones:

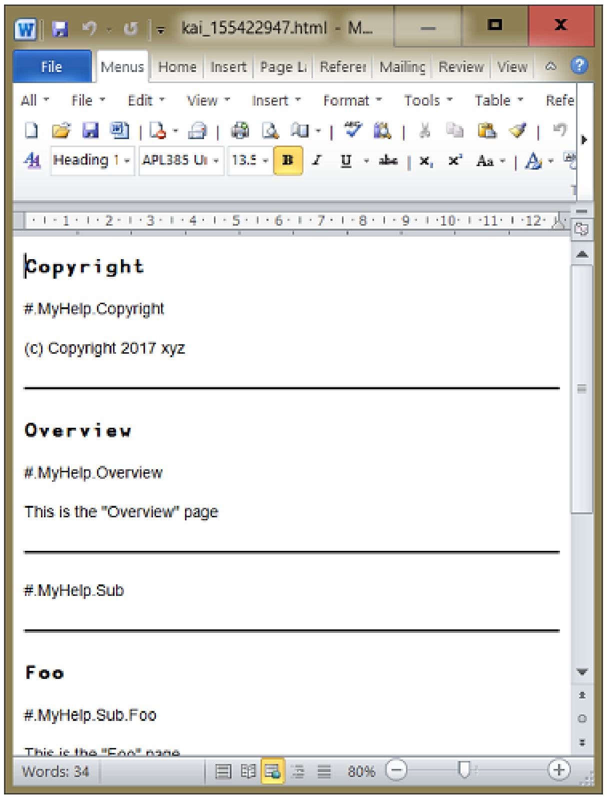
Show topic in browser

This is particularly useful when you use non-default CSS and there is a problem with it: all modern browsers offer excellent tools for investigating CSS, supporting you when hunting bugs or trying to understand unexpected behaviour.

"Create proofread document"

This command creates an HTML document from all the help pages and writes the HTML to a temporary file. The filename is printed to the session.

You can then open that document with your favourite word processor, say Microsoft Word. This will show something like this:



Copyright

#.MyHelp.Copyright

(c) Copyright 2017 xyz

Overview

#.MyHelp.Overview

This is the "Overview" page

#.MyHelp.Sub

Foo

#.MyHelp.Sub.Foo

The help system as a single HTML page

This is a great help when it comes to proofreading a document: one can use the "Review" features of the chosen word processor and also print the document. You are much more likely to spot any problems in a printed version of the document than on screen.

"Reports"

There are several reports available reporting broken and ambiguous links, `∆TopicProperties` functions and help pages that do not carry any index entries.

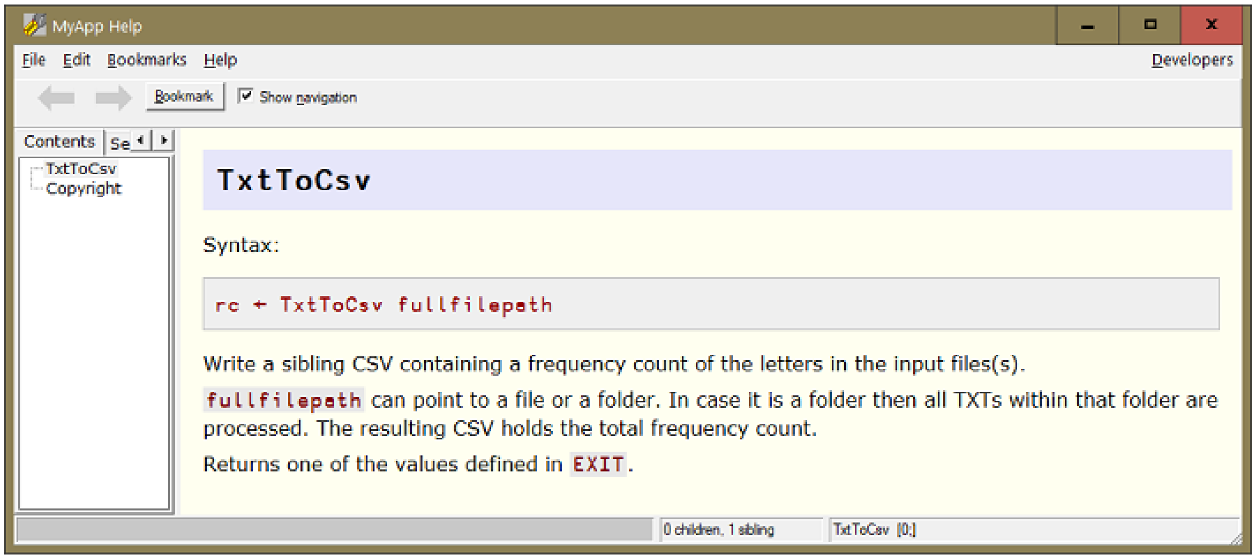
11.10 Export to HTML

You can export the help system so that it becomes a website. For that select "Export as HTML…" from the "File" menu.

The resulting website does not offer all the features the Windows version comes with but at least you can read and print all the pages, you have the tree structure representing the contents and all the links work. That must do under Linux and macOS for the time being.

11.11 Making adjustments

If you have not copied the contents of `code\v11\*` from the book'w website then you should make adjustments to the help system to keep it in sync with the book. We have just two help pages; a page regarding the main method `TxtToCsv`:



TxtToCsv

Syntax:

```
rc ← TxtToCsv fullfilepath
```

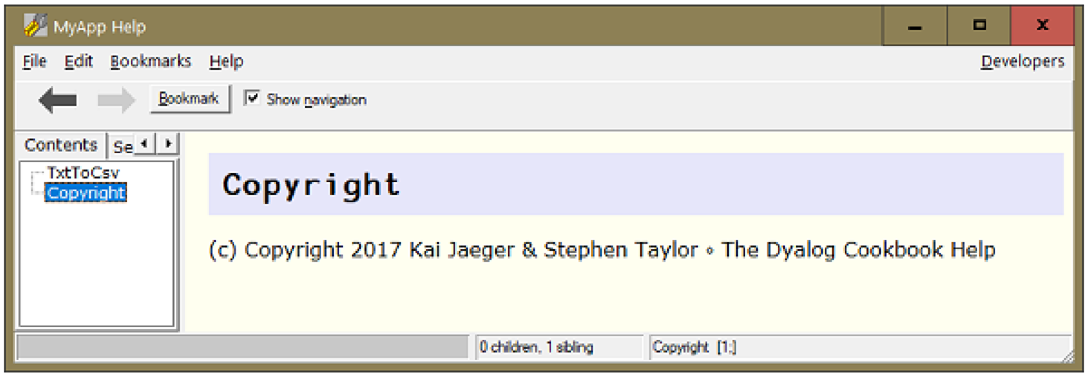
Write a sibling CSV containing a frequency count of the letters in the input files(s).

`fullfilepath` can point to a file or a folder. In case it is a folder then all TXTs within that folder are processed. The resulting CSV holds the total frequency count.

Returns one of the values defined in `EXIT`.

The changed help system

And a page regarding copyright:



Copyright

(c) Copyright 2017 Kai Jaeger & Stephen Taylor ◦ The Dyalog Cookbook Help

The changed help system

11.12 How to view the help system

We want to make sure that we can call the help system from within our application. For that we need a new function, and the obvious name for this function is `ShowHelp`. The function accepts a right argument which might be an empty vector but can be a page name instead. If a page name is provided then of course `Markdown2Help` does not show the first page of the help system but the page specified. It returns an instance of the help system. The function goes into the `MyApp.dyalog` script:

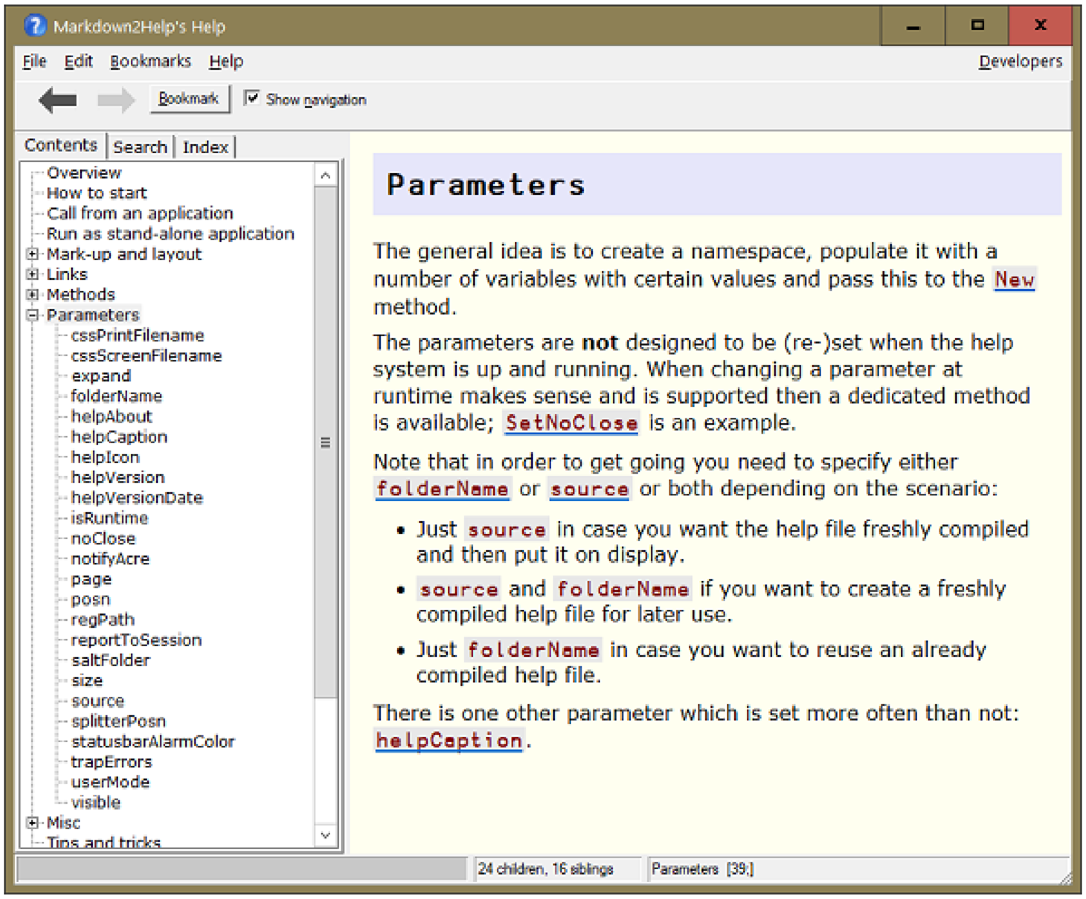
```
:Namespace MyApp
...
∇

∇{r}←ShowHelp pagename;ps
  ps←#.Markdown2Help.CreateParms ⍬
  ps.source←#.MyHelp
  ps.foldername←'Help'
  ps.helpAbout←'MyApp''s help system by John Doe'
  ps.helpCaption←'MyApp Help'
  ps.helpIcon←'file://',##.FilesAndDirs.PWD,'\images\logo.ico'
  ps.helpVersion←'1.0.0'
  ps.helpVersionDate←'YYYY-MM-DD'
  ps.page←pagename
  ps.regPath←'HKCU\Software\MyApp'
  ps.noClose←1
  r←#.Markdown2Help.New ps
∇

∇ r←Public
  r←'StartFromCmdLine' 'TxtToCsv' 'SetLX' 'ShowHelp'
∇

:EndNamespace
```

In case you wonder why a Window Registry key is specified: the user can mark any help page as a favourite, and this is saved in the Windows Registry. We will discuss the Windows Registry in a later chapter.

This function requires the help system to be available in the workspace.

Strictly speaking only the “source” parameter needs to be specified to get it to work, but you really want to specify other parameters as well before a client sets eye on your help system.

Most of the parameters should explain themselves, but if you are in doubt you can always start `Markdown2Help`’s own help system with `#.Markdown2Help.Selfie ⍬` and read the pages under the “Parameters” node. That’s what you would see:



Parameters

The general idea is to create a namespace, populate it with a number of variables with certain values and pass this to the New method.

The parameters are **not** designed to be (re-)set when the help system is up and running. When changing a parameter at runtime makes sense and is supported then a dedicated method is available; SetNoClose is an example.

Note that in order to get going you need to specify either folderName or source or both depending on the scenario:

- Just source in case you want the help file freshly compiled and then put it on display.
- source and folderName if you want to create a freshly compiled help file for later use.
- Just folderName in case you want to reuse an already compiled help file.

There is one other parameter which is set more often than not: helpCaption.

The context menu

You can request a list of all parameters with their default values with this statement:

```
⎕←(#.Markdown2Help.CreateParms'').∆List''
```

Note that `CreateParms` is one of the few functions in the APLTree library with that name that actually require a right argument. This right argument may be just an empty vector, but instead it could be a namespace with variables like "source" or "page". In that case `CreateParms` would inject any missing parameters into that namespace and return it as result.

Therefore we could re-write the function `ShowHelp`:

```
∇{r}←ShowHelp pagename;ps
  ps←⎕NS ''
  ps.source←#.MyHelp
  ps.foldername←'Help'
  ps.helpAbout←'MyApp''s help system by John Doe'
  ps.helpCaption←'MyApp Help'
  ps.helpIcon←'file://',##.FilesAndDirs.PWD,'\images\logo.ico'
  ps.helpVersion←'1.0.0'
  ps.helpVersionDate←'YYYY-MM-DD'
  ps.page←pagename
  ps.regPath←'HKCU\Software\MyApp'
  ps.noClose←1
  ps←#.Markdown2Help.CreateParms ps
  r←#.Markdown2Help.New ps
∇
```

This version of `ShowHelp` would produce exactly the same result.

11.13 Calling the help system from your application

- Start the help system by calling the `New` function as soon as the user presses F1 or selects "Help" from the menubar or requests a particular help page by other means. Catch the result and assign it to a meaningful name: this represents your help system. We use the name `MyHelpInstance`.
- Specify `noClose←1`. This means that when the user attempts to close the help system with a click into the close box or by selecting the "Quit" command from the "File" menu or by pressing Alt+F4 or Ctrl+W then the help system is not really closed down, it just makes itself invisible.
- When the user later requests again a help page use this:

  ```
  1 #.Markdown2Help.Display MyHelpInstance 'Misc'
  ```

  - The `1` provided as left argument forces the GUI to make itself visible, no matter whether it is visible right now or not: the user might have "closed" the help system since she requested a help page earlier on.
  - `MyHelpInstance` represents the help system.
  - "Misc" is the name of the page to be displayed. Can also be empty (`⍬`) in which case the first page is shown.

  Note that the overhead of bringing the help system back this way is pretty close to zero. If you *really* want to get rid of the help system call the `Close` method before deleting the reference:

  ```
  MyHelpInstance.Close
  )erase MyHelpInstance
  ```

11.14 Adding the help system to "MyApp.dyapp"

Now that we have a help system that is saved in the right place we have to make sure that it is loaded when we assemble a workspace with a DYAPP. In a first step we add a function `LoadHelp` to the `DevHelpers` class:

```
:Namespace DevHelpers
...

    ∇{r}←LoadHelp dummy;parms
    parms←#.Markdown2Help.CreateParms ⍬
    parms.saltFolder←#.FilesAndDirs.PWD,'\MyHelp'
    parms.source←'#.MyHelp'
    parms.folderName←'Z:\code\v11\Help\Files'
    {}#.Markdown2Help.LoadHelpWithSalt parms
    ∇

:EndNamespace
```

Calling this function will load the help system from `saltFolder` into the namespace `#.MyHelp` in the current workspace. Therefore we need to call this function within `MyApp.dyapp`:

```
...
Load DevHelpers
Run DevHelpers.LoadHelp ⍬
Run #.MyApp.SetLX ⍬
...
```

11.15 Enhancing "Make.dyapp" and "Make.dyalog"

Now we need to make sure that the "Make" process incorporates the help system. First we add the needed modules to `Make.dyapp`:

```
Target #
Load ..\AplTree\APLTreeUtils
Load ..\AplTree\FilesAndDirs
Load ..\AplTree\HandleError
Load ..\AplTree\IniFiles
Load ..\AplTree\OS
Load ..\AplTree\Logger
Load ..\AplTree\EventCodes
Load ..\APLTree\WinReg
Run 'Markdown2Help' #.⎕CY '..\apltree\Markdown2Help\Markdown2Help.dws'
Load Constants
Load Utilities
Load MyApp
Run #.MyApp.SetLX ⍬

Load Make
Run #.Make.Run 1
```

Finally we have to make sure that the compiled help system is copied over together with the stand-alone help viewer:

```
:Class Make
...
⍝ 5. Creates `MyApp.exe` within `DESTINATION\`
⍝ 6. Copy the Help system into `DESTINATION\Help\files`
⍝ 7. Copy the stand-alone Help viewer into `DESTINATION\Help`
⎕IO←1 ⋄ ⎕ML←1

    DESTINATION←'MyApp'

    ∇ {filename}←Run offFlag;rc;en;more;successFlag;F;U;msg
      :Access Public Shared
      (F U)←##.(FilesAndDirs Utilities)
      (rc en more)←F.RmDir DESTINATION
      U.Assert 0=rc
      successFlag←'Create!'F.CheckPath DESTINATION
      U.Assert successFlag
      (successFlag more)←2↑'images'F.CopyTree DESTINATION,'\images'
      U.Assert successFlag
      (rc more)←'MyApp.ini.template'F.CopyTo DESTINATION,'\MyApp.ini'
      U.Assert 0=rc
      (successFlag more)←2↑'Help\files'F.CopyTree DESTINATION,'\Help\files'
      U.Assert successFlag
      (rc more)←'..\apltree\Markdown2Help\help\ViewHelp.exe'F.CopyTo DESTINATION,'\Help\'
      U.Assert 0=rc
      Export'MyApp.exe'
      filename←DESTINATION,'\MyApp.exe'
      :If offFlag
          ⎕OFF
      :EndIf
    ∇
...
:EndClass
```

12. Scheduled Tasks

12.1 What is a Scheduled Task?

Windows offers a task scheduler in order to run applications at specific times. Like Services Scheduled Tasks are designed for background job, meaning that such applications have no GUI, and cannot have a GUI.

The Scheduler allows you to start the application on a specific date and time once, or every day, every week or every month. The user does not have to be logged on (that's different from old versions of Windows) and it allows to run an application in elevated mode (soon to be discussed).

12.2 What can and cannot be achieved by Scheduled Tasks

Scheduled Tasks – like Services – are perfect for background tasks. Examples are:

- Take a backup once a week
- Check the availability of your website once every hour
- Send a test email to all your email addresses once a week

Scheduled Tasks cannot interact with the user: when you try to put up a GUI and ask a question then nothing will appear on the screen: you just can't do this.

12.3 Scheduled Tasks versus Services

If your application needs to run all the time, even with delays between actions, then running as a Service would be more appropriate. Services are typically started automatically when the machine is booted, and they typically keep running until the next boot.

To make this point clear, imagine these two scenarios:

- You need an application to start once a week and take a backup of a specific folder.
- You need an application to constantly monitor a specific folder for certain file types (say Markdown) and convert them (say into HTML files).

The former is clearly a candidate for a Scheduled Task while the latter is a candidate for a Service.

12.4 Preconditions for a Scheduled Task

You need either a saved workspace with `⎕LX` set or an EXE created from a workspace. An EXE has two advantages compared with an ordinary workspace:

1. The user cannot investigate the code.
2. Dyalog is not required on the target system, not even the Runtime EXE.

If this does not bother you then an EXE has no advantages over a simple saved workspace; it just adds complexity and therefore should be avoided if there aren't any advantages. However, if you cannot be sure whether the required version of Dyalog is installed on the target machine then you have no choice: it has to be a stand-alone EXE.

We have already taken care of handling errors and writing to log files, which are the only sources of information in general, and in particular for analyzing any problems that pop up when a Scheduled Task runs, or crashes. In other words, we are ready to go.

Our application does not suggest itself as a Scheduled Task; it's obviously a candidate for running as a Service, but that does not mean it cannot run as a Scheduled Task, so let's start.

12.5 Precautions: ensure one instance only

When dealing with Scheduled Tasks then usually you don't want more than one instance of the application running at the same time. When there is a problem with a Scheduled Task then one of the most common reasons why getting to the bottom of the problem turns out to be difficult is that you fire up another instance when there is already one running. For example, you try to Ride into it but the port used by Ride is already occupied by an instance that was started earlier without you being aware of this. For that reason we are going to prevent this from happening.

> Even in the rare circumstances when you want an application managed by the Task Scheduler to run in parallel more than once you should establish a mechanism that allows you to enforce having just one instance running if only for debugging purposes. Make it an INI entry (like "AllowMultipleInstances") and document it appropriately.

We resume, as usual, by saving a copy of `Z:\code\v11` as `Z:\code\v12`.

In order to force the application to run only once at any given time we add a function `CheckForOtherInstances` to `MyApp.dyalog`:

```
∇ {tno}←CheckForOtherInstances dummy;filename;listOfTiedFiles;ind
 ⍝ Attempts to tie the file "MyAppCtrl.dcf" exclusively and returns the tie number.
 ⍝ If that is not possible then an error is thrown because we can assume that the
 ⍝ application is already running.\\
 ⍝ Notes:
 ⍝ * In case the file is already tied it is untied first.
 ⍝ * If the file does not exist it is created.
   filename←'MyAppCtrl.dcf'
   :If 0=F.IsFile filename
       tno←filename ⎕FCREATE 0
   :Else
       :If ~0∊⍴⎕FNUMS
           listOfTiedFiles←A.dtb↓⎕FNAMES
           ind←listOfTiedFiles⍳⊂filename
       :AndIf ind≤⍴⎕FNUMS
           ⎕FUNTIE ind⊃⎕FNUMS
       :EndIf
       :Trap 24
           tno←filename ⎕FTIE 0
       :Else
           'Application is already running'⎕SIGNAL C.APP_STATUS.ALREADY_RUNNING
       :EndTrap
   :EndIf
∇
```

Notes:

- First we check whether the file `MyAppCtrl.dcf` exists. If it doesn't we create it and the job is done: creating a file always implies an exclusive tie.
- If it does exist we check whether it is tied by itself, in case we are developing and have restarted the application without having closed it down properly. We then untie the file.
- Finally we attempt to tie the file exclusively but trap error 24 - that's "FILE TIED". If that's the case we throw an error `Constants.APP_STATUS.ALREADY_RUNNING`.
- The file is expected (or will be created) in the current directory.

Since this function will throw an error `Constants.APP_STATUS.ALREADY_RUNNING` we need to add this to the `EXIT` namespace in `MyApp`:

```
:Namespace EXIT
...
        UNABLE_TO_WRITE_TARGET←114
        ALREADY_RUNNING←115
          GetName←{
        ....
:EndNamespace
```

We change `Initial` so that it calls this new function:

```
∇ (Config MyLogger)←Initial dummy
...
   Config←CreateConfig ⍬
   Config.ControlFileTieNo←CheckForOtherInstances ⍬
   CheckForRide Config.(Ride WaitForRide)
...
∇
```

We want to untie the file as well. So far we have not paid any attention to how to close the application down properly, therefore we take the opportunity to introduce a function `Cleanup` which is doing that:

```
∇ {r}←Cleanup
   r←⍬
   ⎕FUNTIE Config.ControlFileTieNo
   Config.ControlFileTieNo←⍬
∇

:EndNamespace
```

Of course we have to call `Cleanup` from somewhere:

```
∇ {r}←StartFromCmdLine arg;MyLogger;Config;rc;⎕TRAP
 ...
   rc←TxtToCsv arg~''''
   Cleanup
   Off rc
∇
```

After all these changes it's time to execute our test cases. Execute `#.Tests.Run`.

Turns out that two of them fail! The reason: when we run `Test_exe_01` and `Test_exe_02` the control file is already tied. That's because `Test_TxtToCsv` runs first, and it calls `Initial` – which ties the control file – but not `Cleanup`, which would untie it. The fix is simple: we need to call `Cleanup` in the test. However, we can't just do this at the end of `Test_TxtToCsv_01`:

```
∇ R←Test_TxtToCsv_01(stopFlag batchFlag);⎕TRAP;rc
...
   →FailsIf rc≢##.MyApp.EXIT.SOURCE_NOT_FOUND
   #.MyApp.Cleanup ⍬
   R←∆OK
∇
```

If we do this then `Cleanup` would not be called in case the check fails. Let's do it properly instead:

```
∇ R←Test_TxtToCsv_01(stopFlag batchFlag);⎕TRAP;rc
...
   →GoToTidyUp rc≢##.MyApp.EXIT.SOURCE_NOT_FOUND
   R←∆OK
  ∆TidyUp:
   ##.MyApp.Cleanup θ
```

Note that we must call `MyApp.Cleanup` rather than just `Cleanup` because we are at that moment in `Tests`, and we don't want to execute `Tests.Cleanup`!

We can learn some lessons from the failure of those two test cases:

1. Obviously the sequence in which the test cases are executed can have an impact on whether tests fail or not. If `Test_TxtToCsv` would have been the last test case the problem would have slipped through undetected.
2. That a test suite runs through OK does not necessarily mean it will keep doing so when you execute it again. If `Test_TxtToCsv` would have been the very last test case the test suite would have passed without a problem but an attempt to run it again would fail because now the control file would have been tied, and `Test_exe_01` would therefore fail.

In our specific case it was actually a problem in the test cases, *not* in `MyApp`, but the conclusion holds true anyway.

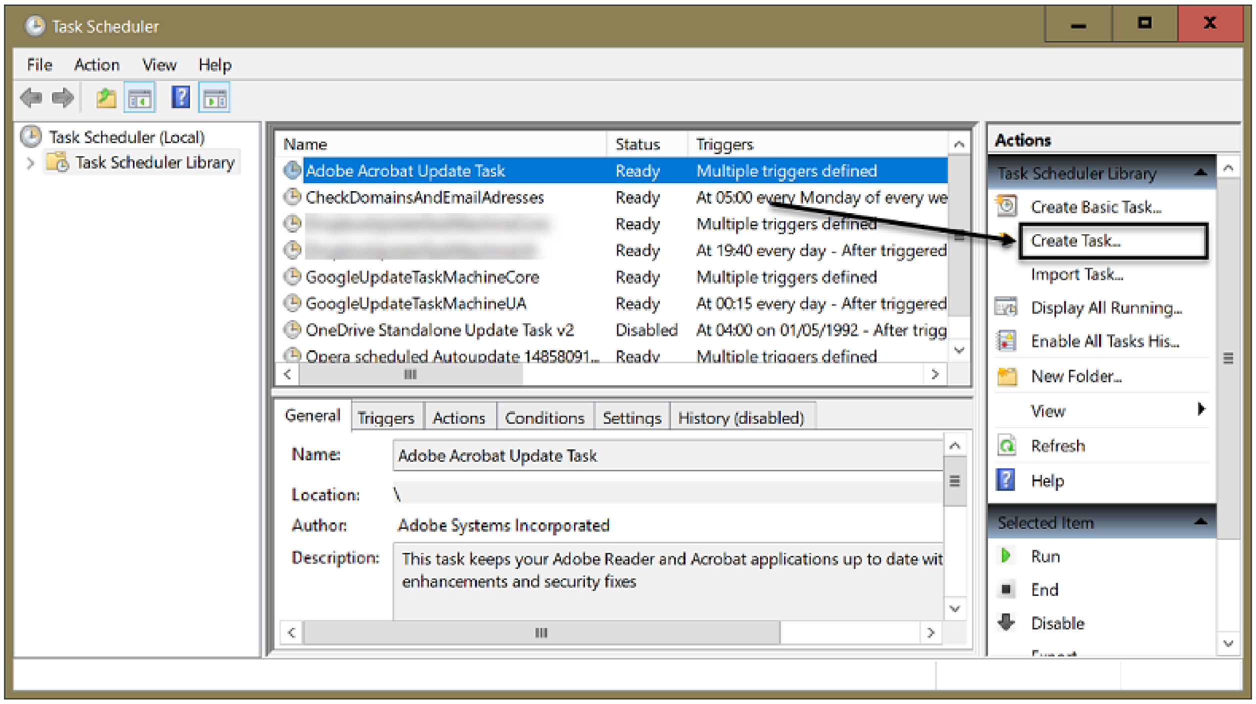
Shuffle test cases

At the time of writing (2017-07) the sequence of the test cases relied on alphabetical order and is therefore predictable. On the to-do list for the `Tester` class is a topic "Add option that makes the test framework shuffle the test cases so that the order of execution is not predictable anymore".

12.6 Create a Scheduled Task

Start the Scheduler

Press the <Win> key and type Scheduler. Select "Task Scheduler" form the list. This is what will come up:



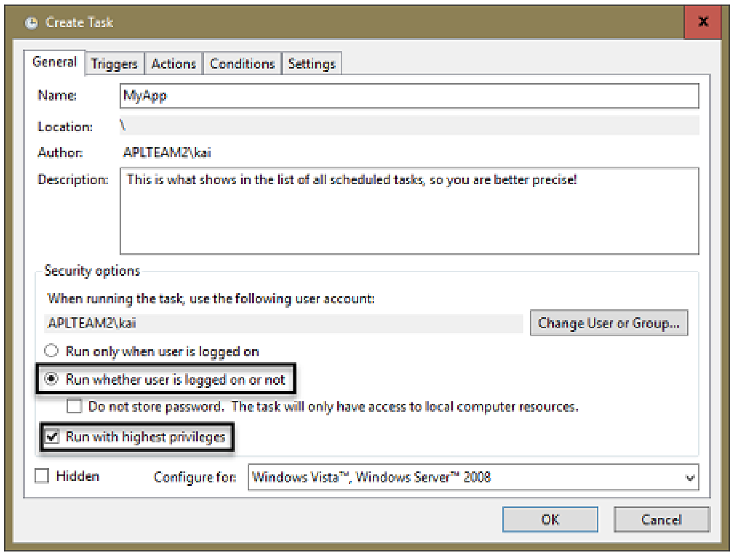
Name	Status	Triggers
Adobe Acrobat Update Task	Ready	Multiple triggers defined
CheckDomainsAndEmailAdresses	Ready	At 05:00 every Monday of every we
[redacted]	Ready	Multiple triggers defined
[redacted]	Ready	At 19:40 every day - After triggered
GoogleUpdateTaskMachineCore	Ready	Multiple triggers defined
GoogleUpdateTaskMachineUA	Ready	At 00:15 every day - After triggered
OneDrive Standalone Update Task v2	Disabled	At 04:00 on 01/05/1992 - After trigg
Opera scheduled Autoupdate 14858091...	Ready	Multiple triggers defined

Name: Adobe Acrobat Update Task
Location: \
Author: Adobe Systems Incorporated
Description: This task keeps your Adobe Reader and Acrobat applications up to date wit enhancements and security fixes

The Windows Task Scheduler

First thing to check is whether the fifth pointin the "Actions" pane on the right reads "Disable All Tasks History" - if it does not you won't be able to get any details regarding a Scheduled Task.

The arrow points to the "Create Task" command - click it.



Name: MyApp
Location: \
Author: APLTEAM2\kai
Description: This is what shows in the list of all scheduled tasks, so you are better precise!

Security options
When running the task, use the following user account:
APLTEAM2\kai
○ Run only when user is logged on
● Run whether user is logged on or not
☐ Do not store password. The task will only have access to local computer resources.
☑ Run with highest privileges
☐ Hidden Configure for: Windows Vista™, Windows Server™ 2008

Create Task

The “General” tab

Name
Used in the list presented by the Task Scheduler.

Description
Shown in the list presented by the Task Scheduler. Keep it concise.

Run only when user is logged on
You will almost certainly change this to “Run whether user is logged on or not”.

Do not store password
The password is stored safely, so there is not really a reason not to provide it.

Running with highest privileges
Unfortunately this check box is offered no matter whether your user account has admin rights or not. If it does not, then ticking the box won’t make a difference at all.

If your user account has no admin rights but your Scheduled Task needs to run with highest privileges then you need to specify a different user id / password after clicking the “Change user or group” button.

Whether your application needs to run with highest privileges or not is impossible to say. Experience shows that sometimes something that does not work when – and only when – the application is running as a Scheduled Task will work fine with highest privileges although it is by no means clear what those rights are required for.

Configure for
Generally you should select the OS the task is running on.

UAC, admin rights and all the rest

With the UAC, users of the admin group have 2 tokens. The filtered token represents standard user rights. This token is used by default, for example when you create a shell (console). Therefore you have just standard user rights by default even when using a user account with admin rights. However, when you have admin rights and you click an EXE and select “run as administrator”, the full token is used which contains admin rights.

Notes:

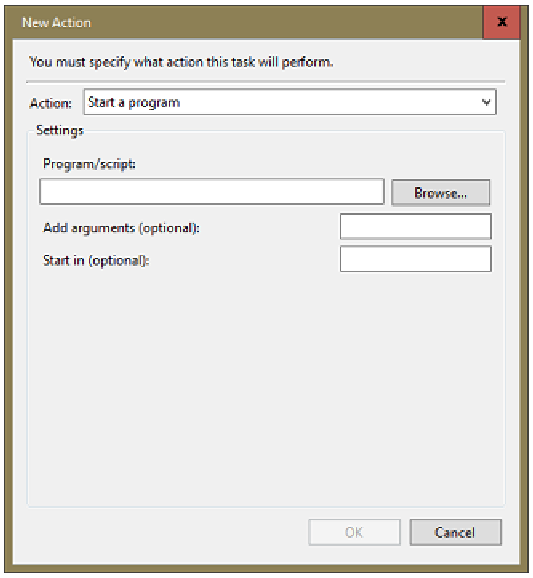
- Some applications ask for admin rights even when you do not right-click on the EXE and select “Run as administrator”; the Registry Editor and the Task Manager are examples.
- Even if you run an application with admin rights (sometimes called “in elevated mode”) it does not mean that the application can do whatever it likes, but as an admin you can always grab any missing rights.
- Remote filesystems are unlikely to be visible; you need to mount them again.

The "Trigger" tab

The tab does not carry any mysteries.

The "Action" tab

After clicking "New" this is what you get:



You must specify what action this task will perform.

Action: Start a program

Settings

Program/script:

Add arguments (optional):

Start in (optional):

New Action

Make sure that you use the "Browse" button to navigate to the EXE/BAT/whatever you want to run as a Scheduled Task. That avoids typos.

"Add arguments" allows you specify something like "MAXWS=345M" or the name of a workspace in case "Program" is not an EXE but a Dyalog interpreter. In particular you should add `DYALOG_NOPOPUPS=1`. This prevents any dialogs from popping up (aplcore, WS FULL etc.). You don't want them when Dyalog is running in the background because there's nobody around to click the "OK" button…

"Start in" is useful for specifying what will become the current (or working) directory for the running program. We recommend setting the current directory in your code as early as possible, so you don't really need to set this here except that when you don't you might well get an error code 2147942512. We will discuss later how such error codes can be analyzed, but for the time being you have to believe us that it actually means "Not enough space available on the disk". When you do specify the "Start in" parameter it runs just fine.

However, note that you *must not delimit* the path with double-quotes. It's understandable that Microsoft does not require them in this context because by definition any blanks are part of the path, but why they do not just ignore them when specified is less understandable.

The "Conditions" tab

The tab does not carry any mysteries.

The “Settings” tab

Unless you have a very good reason not to you should “Allow task to be run on demand” which means you have the “Run” command available on the context menu.

Note that you may specify restart parameters in case the task fails. Whether that makes any sense depends on the application.

The combo box at the bottom allows you to select “Stop the existing instance” which can be quite useful while debugging the application.

Running a Scheduled Task

To start the task right-click on it in the Task Scheduler and select “Run” from the context menu. Then check the log file. We have tested the application well, we know it works, so you should see a log file that contains something like this:

```
2017-03-31 10:03:35 *** Log File opened
2017-03-31 10:03:35 (0) Started MyApp in ...\code\v12\MyApp
2017-03-31 10:03:35 (0)  ...\code\v12\MyApp\MyApp.exe MAXWS=370M
2017-03-31 10:03:35 (0)  Accents           ÁÂÃÀÄÅÇĐÈÊËÉÌÍÎÏÑÒÓÔÕÖØÙÚÛÜÝ  AAAAAA...
2017-03-31 10:03:35 (0)  ControlFileTieNo   1
2017-03-31 10:03:35 (0)  Debug              0
2017-03-31 10:03:35 (0)  DumpFolder         C:\Users\kai\AppData\Local\MyApp\Errors
2017-03-31 10:03:35 (0)  ForceError         0
2017-03-31 10:03:35 (0)  LogFolder          C:\Users\kai\AppData\Local\MyApp\Log
2017-03-31 10:03:35 (0)  Ride               0
2017-03-31 10:03:35 (0)  Trap               1
2017-03-31 10:03:35 (0) Source: MAXWS=370M
2017-03-31 10:03:35 (0) *** ERROR RC=112; MyApp is unexpectedly shutting down: SOURCE_NOT_FOUND
```

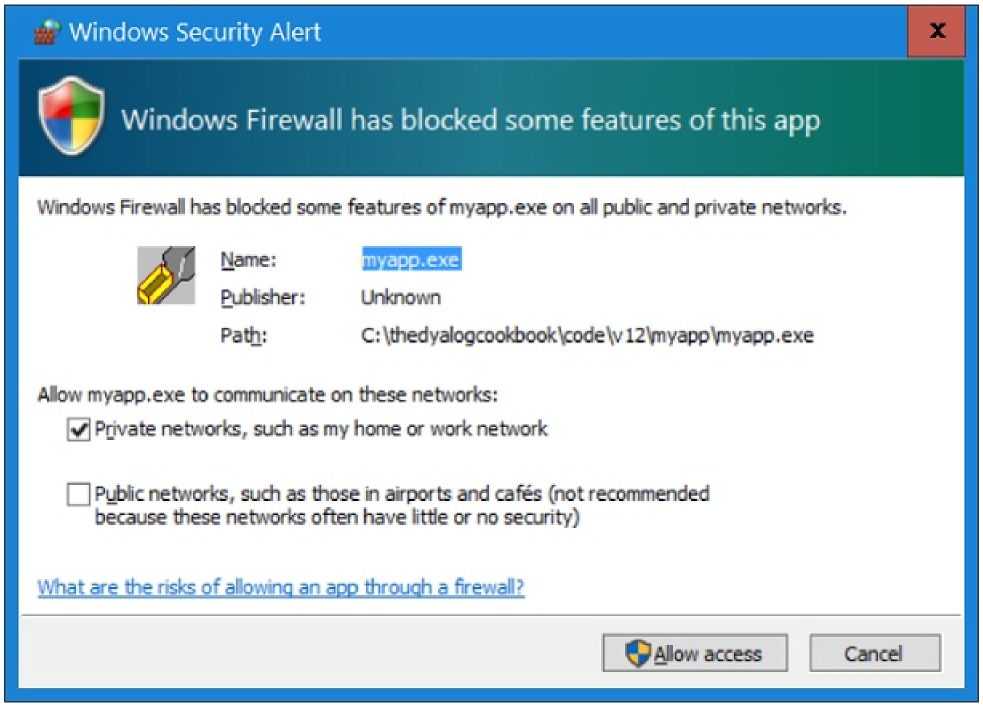
Since we have not provided a filename, `MyApp` assumed that “MAXWS=370M” would be the filename. Since that does not exist the application quits with a return code SOURCE_NOT_FOUND, which is exactly what we expected.

However, from experience we know that the likelihood of the task *not* running as intended is high. We have already discussed some of the issues that might pop up, and we will now discuss some more we have enjoyed over the years.

12.7 Tips, tricks, pitfalls.

Riding into a Scheduled Task

Imagine you want to Ride into a Scheduled Task. Therefore you set in the INI file the `[Ride]Active` flag to `1`. When the Windows Firewall has yet no rules for both this port and this application then you *won’t* see the usual message you expect to see when you run the application for the very first time, assuming that you use a user id with admin rights:


Windows Security Alert

Windows Firewall has blocked some features of this app

Windows Firewall has blocked some features of myapp.exe on all public and private networks.

Name: myapp.exe
Publisher: Unknown
Path: C:\thedyalogcookbook\code\v12\myapp\myapp.exe

Allow myapp.exe to communicate on these networks:

[✓] Private networks, such as my home or work network

[ ] Public networks, such as those in airports and cafés (not recommended because these networks often have little or no security)

What are the risks of allowing an app through a firewall?

Windows Firewall

The application would start, seemingly run for a short period of time and then stop again without leaving any traces: no error codes, no log files, no crash files, nothing.

It is different when you simply double-click the `MyApp.exe`: in that case the "Security Alert" dialog box would pop up, giving you an easy way to create a rule that allows the application to communicate via the given port.

BTW, when you click "Cancel" in the "Security Alert" dialog box then you might expect that the Windows Firewall does not allow access to the port but wouldn't create a rule either, but you would be mistaken. The two buttons "Allow access" and "Cancel" shouldn't be buttons at all! Instead there should be a group "Create rule" with two radio buttons: "Allow access" and "Deny access". In case the user clicks the "Cancel" button a message should pop up saying that although no rule will be created, access to the port in question is denied. That would imply that when the application is started again the "Security Alert" dialog box would pop up again, too. Instead when "Cancel" is clicked a blocking rule for that combination of application and port number is created, and you will not see that dialog box again for this combination.

The Task Scheduler GUI

Once you have executed the "Run" command from the context menu the GUI changes the status from "Ready" to "Running". That's fine. Unfortunately it won't change automatically back to "Ready" once the job has finished, at least not at the time of writing (2017-03) under Windows 10. For that you have to press F5.

Re-make "MyApp"

In case you've found a bug and execute `MyApp`'s `Make.bat` again keep in mind that this means the INI file will be overwritten. So in case you've changed, say, Ride's `Active` flag in the INI file from `0` to `1`, it will be

0 again after the call to `Make.bat`, so any attempt to Ride into the EXE will fail. That's something easy to forget.

MyApp crashes with rc=32

You most probably forgot to copy over the DLLs needed by Ride [1] itself. That's what triggers the return code 32 which stands for "File not find".

Windows return codes

In case you want to translate a Windows return code like 32 into a more meaningful piece of information you might consider downloading the user command `GetMsg` from the APL wiki. Once installed properly you can do this:

```
      ]GetMsgFrom 32
The process cannot access the file because it is being used by another process.
```

However, Microsoft being Microsoft, the error messages are not always that helpful. The above message is issued in case you try to switch on Ride in an application and the interpreter cannot find the DLLs needed by Ride.

Binding MyAPP with the Dyalog development EXE

If for some reason you've created `MyApp.exe` by binding the application with the development version of Dyalog rather than the Runtime (you can do this by providing a 0 as left argument to the `MakeExport` function) then you might run into a problem: Our code takes into account whether it is running under a development EXE or a runtime EXE: error trapping will be inactive (unless it is enforced via the INI file) and `⎕OFF` won't be executed; instead it would execute `→` and hang around but without you being able to see the session. Therefore you are advised not to do this: because you have Ride at your disposal the development version of Dyalog has no advantages over the runtime EXE anyway.

Your application doesn't do what it's supposed to do

… but only when running as a task. Start the Task Scheduler and go to the "History" tab; if this is empty then you have not clicked at "Enable all tasks history" as suggested earlier. Don't get fooled by "Action completed" and "Task completed" - whether a task failed or not does not become apparent this way. Click at "Action completed": at the bottom you get information regarding that run. You might read something like:

"Task Scheduler successfully completed task "MyApp" , instance "{c7cb733a-be97-4988-afca-a551a7907918}" , action "…\code\v12MyAppMyApp.exe" with return code 2147942512."

That tells you that the task did not run at all. Consequently you won't find either a log file or a crash file, and you cannot Ride into the application.

[1]This topic was discussed in the chapter "Debugging a stand-alone EXE"

Task Scheduler error codes

In case the Task Scheduler itself throws an error you will find them of little value at first sight. You can provoke such an error quite easily: edit the task we've created and change the contents of the "Program/script" field in the "Edit action" dialog to something that does not exist, meaning that the Task Scheduler won't find such a program. Then issue the "Run" command from the context menu.

Update the GUI by pressing F5 and you will see that errors are reported. The row that reads "Task Start Failed" in the "Task Category" columns and "Launch Failure" in the "Operational Code" columns is the one we are interested in. When you click at this row you will find that it reports an "Error Value 2147942402". What exactly does this mean?

One way to find out is to google for 2147942402. For this particular error this will certainly do, but sometimes you will have to go through plenty of pages when people managed to produce the same error code in very different circumstances, and it can be quite time consuming to find a page that carries useful information for *your* circumstances.

Instead we use the user command [2] `Int2Hex` which is based on code written and contributed by Phil Last [3]. With this user command we can convert the value 2147942402 into a hex value:

```
      ]Int2Hex 2147942402
80070002
```

Third-party user commands

Naturally there are quite a number of useful third-party user commands available. For details how to install them see Appendix 2.

Now the first four digits, 8007, mean that what follows is a win32 status code. The last 4 are the status code. This is a number that needs to be converted into decimal:

```
      ]Hex2Int 0002
```

but in our case that is of course not necessary because the number is so small that there is no difference between hex and integer anyway, so we can convert it into an error message straight away. Again we use a user command that is not part of a standard Dyalog installation but because it is so useful we strongly recommend to install this as well [4]; it translates any Windows error code into meaningful text.

[2]For details and download regarding the user commands `Hex2Int` and `Int2Hex` see http://aplwiki.com/UserCommands/Hex

[3]http://aplwiki.com/PhilLast

[4]For details and download regarding the user command `GetMsgFrom` see http://aplwiki.com/UserCommands/GetMsgFrom

```
      ]GetMsgFrom
The system cannot find the file specified.
```

That's the reason why it failed.

12.8 Creating tasks programmatically

It is possible to create Scheduled Tasks by a program, although this is beyond the scope of this book. See https://msdn.microsoft.com/en-us/library/windows/desktop/bb736357(v=vs.85).aspx
for details.

13. Windows Services

13.1 What is a Windows Service

While the Windows Task Manager just starts any ordinary application, any application that runs as a Windows Service must be specifically designed in order to meet a number of requirements. In particular services are expected to communicate by exchanging messages with the Windows Service Control Manager (SCM). Commands can be issued by the `SC.exe` (Service Controller) application or interactively via the "Services" application. This allows the user to not only start but also to pause, continue (also called resume) and stop a Windows Service.

13.2 Windows Services and the Window Event Log

Our application is already prepared to write log files and save information in case of a crash, but that's not sufficient: while applications started by the Windows Task Scheduler *might* write to the Windows Event Log, applications running as a Windows Service are *expected* to do that, and for good reasons: when running on a server one cannot expect anybody to be around for keeping an eye on log or crash files. In large organisations running server farms it is common to have a software in place that frequently checks the Windows Event Logs of all those servers, and raise an alarm in one way or another (TCP messages, text messages, emails, whatever) in case it finds any problems.

We won't add the ability to write to the Windows Event Log in this chapter but rather discuss how to do this in the next chapter.

13.3 Restrictions

With Dyalog version 16.0 we cannot install a stand-alone EXE as a Windows Service. All we can do is to install a given interpreter and then ask it to load a workspace which implies running `⎕LX`. In a future version of Dyalog this restriction will most likely be lifted.

That means that in case you don't want to expose your code you have a problem. There are some solutions:

- Lock all the functions and operators in the workspace.
- You can create .NET assemblies from your APL code and call them from the workspace that is running as a Windows Service.
- You can start the stand-alone EXE from the workspace that is running as a Windows Service and communicate with it via TCP/Conga.

All three solutions share the disadvantage that they add a level of complexity without any gain but hiding the code, but at least there are several escape routes available.

We resume, as usual, by saving a copy of `Z:\code\v12` as `Z:\code\v13`.

13.4 The ServiceState namespace

In order to simplify things we are going to make use of the `ServiceState` namespace, another member of the APLTree project. It requires you to do just two things:

1. Call `ServiceState.Init` as early as possible. This function will make sure that the Service is capable of communicating with the SCM.

   To do it as early as possible is necessary so that any request will be answered in time. Windows is not exactly patient when it waits for a Service to respond to a "Pause", "Resume" or "Stop" request: after 5 seconds you are already in danger to see an error message that is basically saying that the Service refused to cooperate. However, note that the interpreter confirms the "start" request for us; no further action is required.

   Normally you need to create a parameter space by calling `CreateParmSpace` and to set at least the name of the log function and possibly the namespace (or class instance) that log function is living in; this log function is used to log any incoming requests from the SCM. The parameter space is then passed as right argument to `Init`.
2. In its main loop the Service is expected to call `ServiceState.CheckServiceMessages`.

   This is an operator, so it needs a function as operand: that is a function that is doing the logging, allowing `CheckServiceMessages` to log its actions to the log file. (If you don't have the need to write to a log file then simply passing `{⍵}` will do.

   If no request of the SCM is pending when `CheckServiceMessages` is called then it will quit straight away and return a `0`. If a "Pause" is pending then it goes into a loop, and it will continue to loop (with a `⎕DL` in between) until either "Continue" (sometimes referred to as "Resume") or "Stop" is requested by the SCM. If a "Stop" is requested the operator will subsequently quit and return a `1`.

13.5 Installing and uninstalling a Service.

Note: for installing as well as un-installing a Service you need admin rights.

Let's assume you have loaded a WS `MyService` which you want to install as a Windows Service run by the same version of Dyalog you are currently in:

```
aplexe←'"',(2 ⎕NQ # 'GetEnvironment' 'dyalog'),'\dyalogrt.exe"'
wsid←'"whereEverTheWsLives\MyAppService.DWS"'
cmd←aplexe,' ',wsid,' APL_ServiceInstall=MyAppService DYALOG_NOPOPUPS=1'
```

`cmd` could now be execute with the `Execute` class which we introduced in chapter 8, "Handling Errors". That would do the trick.

Note `DYALOG_NOPOPUPS=1`: this prevents any dialogs from popping up (aplcore, WS FULL etc.). You don't want them when Dyalog is running in the background because there's nobody around to click the "OK" button. This also prevents the "Service MyAppService successfully installed" message from popping up which you don't want to see when executing tests that install, start, pause, resume, stop and uninstall a Service.

In order to uninstall the Service simply open a console window with "Run as administrator" and enter:

```
sc delete MyAppService
```

and you are done.

Pitfalls when installing / uninstalling Windows Services

Be warned that when you have opened the "Services" GUI while installing or uninstalling a Windows Service then you must press F5 on the GUI in order to update it. The problem is not that the GUI does not update itself, though this can be quite annoying; it can get much worse: you might end up with a Service marked in the GUI as "disabled", and the only thing you can do by then is rebooting the machine. This will happen when you try to perform an action on the GUI when it is not in sync with the Service's current state.

SC: Service Control

`SC` is a command line program that allows a user with admin rights to issue particular commands regarding Services. The general format is:

```
SC.exe {command} {serviceName}
```

Commonly used commands are:

- create
- start
- pause
- continue
- stop
- query
- delete

13.6 Obstacles

From experience we can tell that there are quite a number of traps. In general there are three different types of problems you might encounter:

1. The Service pretends to start (read: show "running" in the Services GUI) but nothing happens at all.
2. The Service starts, the log file reads fine, but once you request the Service to "Pause" or "Stop" you get nothing but a Windows error message.
3. It all works well but the application does not do anything, or something unexpected.

The Service seems to be doing nothing at all

If a Service does not seem to do anything when started:

- Check the name and path of the workspace the Service is expected to load: if that's wrong you won't see anything at all - the message "Workspace not found" goes straight into the ether.
- Make sure the workspace size is sufficent. Again too little memory would not produce any error message.
- The Service might create an aplcore when started. Look out for a file `aplcore` in the Service's current directory to exclude this possibility.
- The Service might have created a CONTINUE workspace for all sorts of reasons.

  Keep in mind that starting with version 16.0 by default Dyalog does *not* drop a CONTINUE workspace by default. You must configure Dyalog accordingly. Also, a CONTINUE cannot be saved in case there is more than one thread running, and Services are by definition multi-threade. However, in case it fails very early there might still be a CONTINUE.
- The Service might have created an aplcore. See the Appendix regarding aplcores for details.

  Keep in mind that once a second thread is started, Dyalog is not able any more to save a CONTINUE workspace. On the other hand you should have established error trapping before a second thread is started; that would avoid this problem.
- Start the "Event Viewer" and check whether any useful piece of information is provided. Although our application does not write to the application log yet, the SCM might!

The Service starts but ignores "Pause" and "Stop" requests.

This requires the log file to contain all the information we expect: calling parameters etc. In such a case we *know* that the Service has started and is running.

- Check whether you have really called `ServiceState.Init`.
- Make sure that you have called `CheckServiceMessages` in the main loop of the application.

If these two conditions are met then it's hard to imagine what could prevent the application from reacting to any requests of the SCM, except when you have an endless loop somewhere in your application.

The application does not do what's supposed to do.

First and foremost it is worth mentioning that any application that is supposed to run as a Service should be developed as an ordinary application, including test cases. When it passes such test cases you have reasons to be confident that the application should run fine as a Service as well.

Having said this, there can be surprising differences between running as an ordinary application and a Service. For example, when a Service runs not with a user's account but with the system account (which is quite normal to do) any call to `#.FilesAndDirs.GetTempPath` results in

```
"C:\Windows\System32\config\systemprofile\AppData\Local\Apps"
```

while for a user's account it would be something like

`'C:\Users\{username}\AppData\Local\Temp'`.

When the application behaves in an unexpected way you need to debug it, and for that Ride is invaluable.

13.7 Potions and wands

Ride

First of all we have to make sure that the application provides us a Ride if we need one. Since passing any arguments for a Ride via the command line requires the Service to be uninstalled and installed at least twice we recommend preparing the Service from within the application instead.

If you have trouble to Ride into any kind of application: make sure that there is not an old instance of the Service running which might occupy the port you need for the Ride.

There are two very different scenarios when you might want to use Ride:

- The Service does not seem to start or does not react to "Pause" or "Stop" requests.
- Although the Service starts fine and reacts properly to any "Pause" or "Stop" requests, the application is behaving unexpectedly.

In the former case make sure that you allow the programmer to Ride into the Service as soon as possible - literally. That means that the second line of the function noted on `⎕LX` should provide a Ride, assuming that the first line sets `⎕IO`, `⎕ML` etc.

At such an early stage we don't have an INI file instantiated, so we cannot switch Ride on and off via the INI file, we have to modify the code for that. You might feel tempted to overcome this by doing it a bit later (read: after having processed the INI file etc.) but we warn you: if a Service does not cooperate then "a bit later" might well be too late to get to the bottom of the problem, so don't.

In the latter case you should add the call to `CheckForRide` to the main loop of the application.

Make sure that you have *never* more than one of the two calls to `CheckForRide` active: if both are active you would be able to make use of the first one but the second one would throw you out!

Logging

Local logging

We want to log as soon as possible any command-line parameters as well as any message exchange between the Service and the SCM. Again we advise you to not wait until the folder holding the log files is defined by instantiating the INI file. Instead we suggest making the assumption that a certain folder ("Logs") will (or might) exist in the current directory which will become where the workspace was loaded from.

If that's not suitable then consider passing the directory that will host the "Logs" folder as a command line parameter.

Windows event log

In the next chapter we will discuss why and how to use the Windows Event Log, in particular when it comes to Services.

13.8 How to implement it

Setting the latent expression

First of all we need to point out that `MyApp` as it stands is hardly a candidate for a Service. Therefore we have to make something up: the idea is to specify one to many folders to be watched by the `MyApp` Service. If any files are found then those are processed. Finally the app will store hashes for all files it has processed. That allows it to recognize any added, changed or removed files efficiently.

For the Service we need to create a workspace that can be loaded by that Service. Therefore we need to set `⎕LX`, and for that we create a new function:

```
∇ {r}←SetLXForService(earlyRide ridePort)
  ⍝ Set Latent Expression (needed in order to export workspace as EXE)
  ⍝ `earlyRide` is a flag. 1 allows a Ride.
  ⍝ `ridePort`  is the port number to be used for a Ride.
    r←⍬
    ⎕LX←'#.MyApp.RunAsService ',(⍕earlyRide),' ',(⍕ridePort)
∇
```

The function takes a flag `earlyRide` and an integer `ridePort` as arguments. How and when this function is called will be discussed in a moment.

Because we have now two functions that set `⎕LX` we shall rename the original one (`SetLX`) to `SetLXForApplication` to tell them apart.

Initialising the Service

Next we need the main function for the service:

```
∇ {r}←RunAsService(earlyRide ridePort);⎕TRAP;MyLogger;Config;∆FileHashes
   ⍝ Main function when app is running as a Windows Service.
   ⍝ `earlyRide`: flag that allows a very early Ride.
   ⍝ `ridePort`: Port number used by Ride.
    r←⍬
    #.⎕IO←1 ⋄ #.⎕ML←1 ⋄ #.⎕WX←3 ⋄ #.⎕PP←15 ⋄ #.⎕DIV←1
    CheckForRide earlyRide ridePort
    #.FilesAndDirs.PolishCurrentDir
    ⎕TRAP←#.HandleError.SetTrap ⍬
    (Config MyLogger)←Initial #.ServiceState.IsRunningAsService
    ⎕TRAP←(Config.Debug=0)SetTrap Config
    Config.ControlFileTieNo←CheckForOtherInstances ⍬
```

```
      ∆FileHashes←0 2⍴''
      :If #.ServiceState.IsRunningAsService
          {MainLoop ⍵}&ridePort
          ⎕DQ'.'
      :Else
          MainLoop ridePort
      :EndIf
      Cleanup ⍬
      Off EXIT.OK
    ∇
```

Notes:

- This function allows a Ride very early indeed.
- It calls the function `Initial` and passes the result of the function `#.ServiceState.IsRunningAsService` as right argument. We will discuss `Initial` next.
- We create a global variable `∆FileHashes` which we use to collect the hashes of all files that we have processed. This gives us an easy and fast way to check whether any of the files we've already processed got changed.
- We call `MainLoop` (a function that has not been established yet) in different ways depending on whether the function is running as a Windows Service or not for the simple reason that it is much easier to debug an application that runs in a single thread.

The "business logic"

Time to change the `MyApp.Initial` function:

```
 ∇ (Config MyLogger)←Initial isService;parms
    ⍝ Prepares the application.
      #.⎕IO←1 ⋄ #.⎕ML←1 ⋄ #.⎕WX←3 ⋄ #.⎕PP←15 ⋄ #.⎕DIV←1
      Config←CreateConfig isService
      Config.ControlFileTieNo←CheckForOtherInstances ⍬
      CheckForRide (0≠Config.Ride) Config.Ride
      MyLogger←OpenLogFile Config.LogFolder
      MyLogger.Log'Started MyApp in ',F.PWD
      MyLogger.Log 2 ⎕NQ'#' 'GetCommandLine'
      MyLogger.Log↓⎕FMT Config.∆List
      :If isService
          parms←#.ServiceState.CreateParmSpace
          parms.logFunction←'Log'
          parms.logFunctionParent←MyLogger
          #.ServiceState.Init parms
      :EndIf
 ∇
```

Note that we pass `isService` as right argument to `CreateConfig`, so we must amend `CreateConfig` accordingly:

```
 ∇ Config←CreateConfig isService;myIni;iniFilename
   Config←⎕NS''
   ...
   Config.IsService←isService
   ...
       Config.Accents←⊃Config.Accents myIni.Get'Config:Accents'
        :If isService
            Config.WatchFolders←⊃myIni.Get'Folders:Watch'
        :Else
            Config.LogFolder←'expand'F.NormalizePath⊃Config.LogFolder myIni.Get'Folders:Logs'
        :EndIf
        Config.DumpFolder←'expand'F.NormalizePath⊃Config.DumpFolder myIni.Get'Folders:Errors'
   ...
 ∇
```

Note that `WatchFolder` is introduced only when the application is running as a Service.

Time to introduce the function `MainLoop`:

```
∇ {r}←MainLoop port;S
  r←⍬
  MyLogger.Log'"MyApp" server started'
  S←#.ServiceState
  :Repeat
      CheckForRide 0 port
      LoopOverFolder ⍬
      :If (MyLogger.Log S.CheckServiceMessages)S.IsRunningAsService
          MyLogger.Log'"MyApp" is about to shut down...'
          :Leave
      :EndIf
      ⎕DL 2
  :Until 0
 ⍝Done
∇
```

Notes:

- We put a call to the function `CheckForRide` into the code but pass a 0 as first item of the right argument, making it inactive for the time being.
- The call to `ServiceState.CheckServiceMessages` makes sure that the function reacts to any status change requests from the SCM.
- `LoopOverFolder` is doing the real work.

The function `LoopOverFolder`:

```
∇ {r}←LoopOverFolder dummy;folder;files;hashes;noOf;rc
  r←θ
  :For folder :In Config.WatchFolders
      files←#.FilesAndDirs.ListFiles folder,'\*.txt'
      hashes←GetHash¨files
      (files hashes)←(~hashes∊∆FileHashes[;2])∘/¨files hashes
      :If 0<noOf←LoopOverFiles files hashes
          :If EXIT.OK=rc←TxtToCsv folder
              MyLogger.Log'Totals.csv updated'
          :Else
              LogError rc('Could not update Totals.csv, RC=',EXIT.GetName rc)
          :EndIf
      :EndIf
  :EndFor
∇
```

This function calls `GetHash` so we better introduce this:

```
 GetHash←{
 ⍝ Get hash for file ⍵
     ⊣2 ⎕NQ'#' 'GetBuildID'⍵
 }
```

The function `LoopOverFiles`:

```
 ∇ noOf←LoopOverFiles(files hashes);file;hash;rc
   noOf←0
   :For file hash :InEach files hashes
       :If EXIT.OK=rc←TxtToCsv file
           ∆FileHashes⍪←file hash
           noOf+←1
       :EndIf
   :EndFor
 ∇
```

This function finally calls `TxtToCsv`.

Because of the change we've made to the right argument of `Initial` we need to change `StartFromCmdLine`; Here the function `Initial` needs to be told that it is *not* running as a Service:

```
∇ {r}←StartFromCmdLine arg;MyLogger;Config;rc;⎕TRAP
...
   (Config MyLogger)←Initial #.ServiceState.IsRunningAsService
...
```

Two more changes:

```
 ∇ {r}←Cleanup dummy
   r←⍬
   ⎕FUNTIE Config.ControlFileTieNo
   Config.ControlFileTieNo←⍬
   '#'⎕WS'Event' 'ServiceNotification' 0
 ∇
```

This disconnects the handler from the "ServiceNotification" event.

Finally we redefine what's a public function:

```
 ∇ r←PublicFns
   r←'StartFromCmdLine' 'TxtToCsv' 'SetLXForApplication' 'SetLXForService' 'RunAsService'
 ∇
```

Running the test cases

Now it's time to make sure that we did not break anything: double-click `MyApp.dyapp` and answer the question whether you would like to run all test cases with "y". If something does not work execute `#.Tests.RunDebug 0` and fix the problem(s).

Installing and un-installing the Service

In order to install as well as un-install the Service we should have two BAT files: `InstallService.bat` and `Uninstall_Service.bat`. We will create these BAT files from Dyalog. For that we create a class `ServiceHelpers`:

```
:Class ServiceHelpers

    ∇ {r}←CreateBatFiles dummy;path;cmd;aplexe;wsid
      :Access Public Shared
    ⍝ Write two BAT files to the current directory:
    ⍝ Install_Service.bat and Uninstall_Service.bat
      r←⍬
      path←#.FilesAndDirs.PWD

      aplexe←'"',(2 ⎕NQ'#' 'GetEnvironment' 'dyalog'),'\dyalogrt.exe"'
      wsid←'"%~dp0\MyAppService.DWS"'
      cmd←aplexe,' ',wsid,' APL_ServiceInstall=MyAppService'
      cmd,←' DYALOG_NOPOPUPS=1 MAXWS=64MB'
      #.APLTreeUtils.WriteUtf8File(path,'\Install_Service.bat')cmd

      cmd←⊂'sc delete MyAppService'
      cmd,←⊂'@echo off'
      cmd,←⊂'    echo Error %errorlevel%'
      cmd,←⊂'    if NOT ["%errorlevel%"]==["0"] ('
      cmd,←⊂'    pause'
      cmd,←⊂'exit /b %errorlevel%'
```

```
       cmd,←⊂')'
       #.APLTreeUtils.WriteUtf8File(path,'\Uninstall_Service.bat')cmd
      ⍝Done
    ∇

:EndClass
```

Notes:

- The install BAT will use the version of Dyalog used to create the BAT file, and it will call the runtime EXE.
- In case you are not familiar with `%~dp0`: this stand for "the directory this BAT file was loaded from". In other words: as long as the workspace `MyAppService.DWS` (which we have not created yet) is a sibling of the BAT file it will work.
- The un-install BAT file will check the `errorlevel` variable. If it detects an error it will pause so that one can actually see the error message even when we have just double-clicked the BAT file. We've discussed this in the chapter "Handling errors".

"Make" for the Service

Now it's time to create a DYAPP for the service. For that copy `Make.dyapp` as `MakeService.dyapp` and then edit it:

```
Target #
Load ..\AplTree\APLTreeUtils
Load ..\AplTree\FilesAndDirs
Load ..\AplTree\HandleError
Load ..\AplTree\IniFiles
Load ..\AplTree\OS
Load ..\AplTree\Logger
Load ..\AplTree\EventCodes
Load Constants
Load Utilities
Load MyApp

Load ..\AplTree\ServiceState
Load ..\AplTree\Tester
Load ..\AplTree\Execute
Load ..\AplTree\WinSys
Load TestsForServices
Load ServiceHelpers

Run #.ServiceHelpers.CreateBatFiles ⍬
Run '#.⎕EX''ServiceHelpers'''
Run #.MyApp.SetLXForService 0 4599   ⍝ [1|0]: Ride/no Ride, [n] Ride port number

Load MakeService
Run #.MakeService.Run 0
```

Notes:

- We need some more APLTree modules: `Tester`, `Execute` and `WinSys`.
- We make sure that the two BAT files for installing and un-installing the service are written to the disk.
- We delete the class `ServiceHelpers`: it is not needed for running the Service any more once we've created the BAT files.
- We set `⎕LX` by calling `SetLXForService`.
- We load the class `MakeService` and run `MakeService.Run`.

That obviously requires the class `MakeService` to be introduced:

```
:Class MakeService
⍝ Creates a workspace "MyAppService" which can then run as a service.
⍝ 1. Re-create folder DESTINATION in the current directory
⍝ 2. Copy the INI file template over to DESTINATION\ as MyApp.ini
⍝ 3. Save the workspace within DESTINATION
    ⎕IO←1 ⋄ ⎕ML←1
    DESTINATION←'MyAppService'

    ∇ {r}←Run offFlag;en;F;U
      :Access Public Shared
      r←⍬
      (F U)←##.(FilesAndDirs Utilities)
      (rc en more)←F.RmDir DESTINATION
      U.Assert 0=rc
      U.Assert 'Create!'##.FilesAndDirs.CheckPath DESTINATION
      'MyApp.ini.template' CopyTo DESTINATION,'\MyApp.ini'
      'Install_Service.bat' CopyTo DESTINATION,'\'
      'Uninstall_Service.bat' CopyTo DESTINATION,'\'
      ⎕WSID←DESTINATION,'\',DESTINATION
      #.⎕EX⍕⎕THIS
      0 ⎕SAVE ⎕WSID
      {⎕OFF}⍣(⊃offFlag)⊣⍬
    ∇

    ∇ {r}←from CopyTo to;rc;more;msg
      r←⍬
      (rc more)←from F.CopyTo to
      msg←'Copy failed RC=' ,(⍕rc),'; ',more
      msg ⎕signal 11/⍨0≠rc
    ∇
:EndClass
```

Notes:

- Assigns the name of the destination folder to the global `DESTINATION`.
- (Re-)creates a folder with the name `DESTINATION`.

- Copies over the INI file as well as the two BAT files.
- Finally it sets `⎕WSID` and saves the workspace without the status indicator and without `MakeService` by deleting itself.

Self-deleting code

In case you wonder how it is possible that the function `MakeService.Run` deletes itself and keeps running anyway:

APL code (functions, operators and scripts) that is about to be executed is copied onto the stack. You can investigate the stack at any given moment with `)si` and `)sinl`; for details type the command in question into the session and then press F1.

Even if the code of a class executes `⎕EX ⍕⎕THIS` or a function or operator `⎕EX ⊃⎕SI` the code keeps running because the copy on the stack will exist until the function or operator quits. Scripts might even live longer: only when the last reference pointing to a script is deleted does the script cease to exist.

13.9 Testing the Service

We have test cases that ensure that the "business logic" of `MyApp` works just fine. What we also need are tests that make sure that it runs fine as a Service as well.

Since the two test scenarios are only loosely related we want to keep those tests separate. It is easy to see way: testing the Service means assembling all the needed stuff, installing the Service, carrying out the tests and finally un-installing the tests and cleaning up. We don't want to execute this unless we really have to.

We start be creating a new script `TestsForServices` which we save alongside the other scrips in `v13/`:

```
:Namespace TestsForServices
⍝ Installs a service "MyAppService" in a folder within the Windows Temp directory with
⍝ a randomly chosen name. The tests then start, pause, continue and stop the service.\\
⍝ They also check whether the application produces the expected results.

    ⎕IO←1 ⋄ ⎕ML←1

:EndNamespace
```

We now discuss the functions we are going to add one after the other. Note that the `Initial` function is particularly important in this scenario: we need to copy over all the stuff we need, code as well as input files, make adjustments, and install the Service. This could all be done in a single function but it would be lengthy and difficult to read. To avoid this we split the function into obvious units. By naming those functions carefully we should get away without adding any comments because the code explains itself. Here we go:

```
∇ r←Initial;rc;ini;row;bat;more
   ∆Path←##.FilesAndDirs.GetTempFilename''
   #.FilesAndDirs.DeleteFile ∆Path
   ∆Path←¯4↓∆Path
   ∆ServiceName←'MyAppService'
   r←0
   :If 0=#.WinSys.IsRunningAsAdmin
       ⎕←'Sorry, but you need admin rights to run this test suite!'
       :Return
   :EndIf
   ∆CreateFolderStructure ⍬
   ∆CopyFiles ⍬
   ∆CreateBATs ⍬
   ∆CreateIniFile ⍬
   ∆InstallService ⍬
   ⎕←'*** Service ',∆ServiceName,' successfully installed'
   r←1
```

Note that all the sub-function and global variables start their names with `∆`. An example is the function `∆Execute_SC_Cmd`:

```
∇ {(rc msg)}←∆Execute_SC_Cmd command;cmd;buff
 ⍝ Executes a SC (Service Control) command
   rc←1 ⋄ msg←'Could not execute the command'
   cmd←'SC ',command,' ',∆ServiceName
   buff←#.Execute.Process cmd
   →FailsIf 0≠1⊃buff
   msg←⊃,/2⊃buff
   rc←3⊃buff
∇
```

It executes `SC` commands like “start”, “pause”, “continue”, “stop” and “query” by preparing a string and then passing it to `Execute.Process`. It analyzes the result and returns the text part of it as well as a return code. While the first 4 commands aim to change the current status of a Service, “query” is designed to establish what the current status of a Service actually is.

After having executed the test suite we want to clean up, so we create a function `Cleanup`. Just a reminder: in case the test framework finds a function `Initial` it executes it *before* executing the actual test cases, while any function `Cleanup` will be executed *after* the test cases have been executed.

```
∇ {r}←Cleanup
   r←⍬
   :If 0<⎕NC'∆ServiceName'
       ∆Execute_SC_Cmd'stop'
       ∆Execute_SC_Cmd'delete'
       ##.FilesAndDirs.RmDir ∆Path
       ⎕EX¨'∆Path' '∆ServiceName'
   :EndIf
∇
```

We also need `∆Pause`:

```
∇ {r}←∆Pause seconds
   r←⍬
   ⎕←'   Pausing for ',(⍕seconds),' seconds...'
   ⎕DL seconds
∇
```

We could discuss all the sub functions called by these two functions but it would tell us little. Therefore we suggest that you copy the code from the web site. We just discuss the two test functions:

```
∇ R←Test_01(stopFlag batchFlag);⎕TRAP;rc;more
  ⍝ Start, pause, continue and stop the service.
  ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
  R←∆Failed

  (rc more)←∆Execute_SC_Cmd'start'
  →FailsIf 0≠rc
  ∆Pause 2
  (rc more)←∆Execute_SC_Cmd'query'
  →FailsIf 0≠rc
  →FailsIf 0=∨/'STATE : 4 RUNNING'⍷#.APLTreeUtils.dmb more

  (rc more)←∆Execute_SC_Cmd'pause'
  →FailsIf 0≠rc
  ∆Pause 2
  →FailsIf 1≠⍴#.FilesAndDirs.ListFiles ∆Path,'\service\Logs\'
  (rc more)←∆Execute_SC_Cmd'query'
  →FailsIf 0=∨/'STATE : 7 PAUSED'⍷#.APLTreeUtils.dmb more

  (rc more)←∆Execute_SC_Cmd'continue'
  →FailsIf 0≠rc
  ∆Pause 2
  (rc more)←∆Execute_SC_Cmd'query'
  →FailsIf 0=∨/'STATE : 4 RUNNING'⍷#.APLTreeUtils.dmb more

  (rc more)←∆Execute_SC_Cmd'stop'
  →FailsIf 0≠rc
  ∆Pause 2
  (rc more)←∆Execute_SC_Cmd'query'○
  →FailsIf 0=∨/'STATE : 1 STOPPED'⍷#.APLTreeUtils.dmb more

  R←∆OK
∇
```

In order to understand the `→FailsIf` statements it is essential to have a look at a typical result returned by the `∆Execute_SC_Cmd` function, in this case a "query":

```
      ⍴more
328
      ≡more
1
      #.APLTreeUtils.dmb more
SERVICE_NAME: MyAppService TYPE : 10 WIN32_OWN_PROCESS STATE : 4 RUNNING (STOPPABLE, PAUSABLE, ACCEP\
TS_SHUTDOWN) WIN32_EXIT_CODE : 0 (0x0) SERVICE_EXIT_CODE
       : 0 (0x0) CHECKPOINT
```

Note that we have removed multiple blanks here in order to increase readability. The reason is that the result carries plenty of them.

This test starts, pauses, continues and finally stops the Service after having processed some files:

```
∇ R←Test_02(stopFlag batchFlag);⎕TRAP;rc;more;noOfCSVs;success;oldTotal;newTotal;A;F
  ⍝ Start service, check results, give it some more work to do, check and stop it.
   ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
   R←∆Failed
   (A F)←#.(APLTreeUtils FilesAndDirs)

   (rc more)←∆Execute_SC_Cmd'start'
   →FailsIf 0≠rc
   ∆Pause 1
   (rc more)←∆Execute_SC_Cmd'query'
   →FailsIf 0=∨/'STATE : 4 RUNNING'⍷A.dmb more

   ⍝ At this point the service will have processed all the text files, so there
   ⍝ must now be some CSV files, including the Total.csv file.
   ⍝ We then copy 6 more text files, so we should see 6 more CSVs & a changed Total.
   oldTotal←↑{','A.Split ⍵}¨A.ReadUtf8File ∆Path,'\input\en\total.csv'
   noOfCSVs←⍴F.ListFiles ∆Path,'\input\en\*.csv'
   (success more list)←(∆Path,'\texts')F.CopyTree ∆Path,'\input\'  ⍝ All of them
   {1≠⍵:.}success
   ∆Pause 2
   newTotal←↑{','A.Split ⍵}¨A.ReadUtf8File ∆Path,'\input\en\total.csv'
   →PassesIf(noOfCSVs+6)=⍴F.ListFiles ∆Path,'\input\en\*.csv'
   →PassesIf oldTotal≢newTotal
   oldTotal[;2]←⍎¨oldTotal[;2]
   newTotal[;2]←⍎¨newTotal[;2]
   →PassesIf oldTotal[;2]∧.≤newTotal[;2]
   (rc more)←∆Execute_SC_Cmd'stop'
   →FailsIf 0≠rc
   ∆Pause 2
   (rc more)←∆Execute_SC_Cmd'query'
   →FailsIf 0=∨/'STATE : 1 STOPPED'⍷A.dmb more

   R←∆OK
∇
```

Though this test starts and stops the Service, it's real purpose is to make sure that the Service processes input files as expected.

Running the tests

First we need to make sure that everything is assembled freshly, and with admin rights. The best way to do that is to run the script `MakeService.dyapp` from a console that was started with admin rights. This is because unfortunately you cannot right-click on a DYAPP and select "Run as administrator" from the context menu.

Console with admin rights.

The best way to start a console window with admin rights:

1. Press the Windows key.
2. Type "cmd"; if you are tempted to ask "where shall I type this into" then don't - just type.
3. Right-click on "Command prompt" and select "Run as administrator".

A Dyalog instance is started. In the session you should see something similar to this:

```
Booting C:\...\v13\MakeService.dyapp
Loaded: #.APLTreeUtils
Loaded: #.FilesAndDirs
Loaded: #.HandleError
Loaded: #.IniFiles
Loaded: #.OS
Loaded: #.Logger
Loaded: #.EventCodes
Loaded: #.Constants
Loaded: #.Utilities
Loaded: #.MyApp
Loaded: #.ServiceState
Loaded: #.Tester
Loaded: #.Execute
Loaded: #.WinSys
Loaded: #.TestsForServices
Loaded: #.ServiceHelpers
```

```
#.⎕EX'ServiceHelpers'
Loaded: #.MakeService
```

In the next step establish the test helpers by calling `#.TestsForServices.GetHelpers`.

Finally run `#.TestsForServices.RunDebug 0`. You should see something like this:

```
#.TestsForServices.RunDebug 0
--- Test framework "Tester" version 3.3.0 from YYYY-MM-DD -----------------------------
Searching for INI file testcases_APLTEAM2.ini
  ...not found
Searching for INI file Testcases.ini
  ...not found
Looking for a function "Initial"...
*** Service MyAppService successfully installed
  "Initial" found and sucessfully executed
--- Tests started at YYYY-MM-DD hh:mm:dd on #.TestsForServices ------------------------
   Pausing for 2 seconds...
   Pausing for 2 seconds...
   Pausing for 2 seconds...
   Pausing for 2 seconds...
  Test_01 (1 of 2) : Start, pause and continue the service.
   Pausing for 2 seconds...
   Pausing for 2 seconds...
   Pausing for 2 seconds...
  Test_02 (2 of 2) : Start service, check results, give it some more work to do, check and stop it.
 --------------------------------------------------------------------------------------------------
   2 test cases executed
   0 test cases failed
   0 test cases broken
Time of execution recorded on variable #.TestsForServices.TestCasesExecutedAt in: YYYY-MM-DD hh:mm:ss
Looking for a function "Cleanup"...
  Function "Cleanup" found and sucessfully executed.
*** Tests done
```

14. The Windows Event Log

Now that we have managed to establish `MyApp` as a Windows Service we have to make sure that it behaves. That means we need to make it report to the Windows Event Log.

14.1 What exactly is the Windows Event Log?

In case you've never heard of it, or you are not sure what exactly the purpose of it is, this is for you; otherwise jump to "Why is the Windows Event Log important?".

The Windows Event Log is by no means an alternative to application specific log files. Most ordinary applications do not write to the Windows Event Log at all, some only when things go wrong and very few always. In other words, for ordinary applications you may or may not find useful information in the Windows Event Log.

That's very different for any application that runs as a Windows Service: those are *expected* to write to the Windows Event Log when it starts, when it quits and when it encounters problems, and it might add even more information. You will find it hard to find an exception.

Similarly Scheduled Tasks are expected to do the same, although some don't, or report just errors.

14.2 Is the Windows Event Log important?

On a server all applications run either as Windows Services (most likely all of them) or as Windows Scheduled Tasks. Since no human is sitting in front of a server we need a way to detect problems on servers automatically. That can be achieved by using software that constantly scans the Windows Event Log. It can email or text admins when an application that's supposed to run doesn't, or when an application goes astray, drawing attention to that server.

In large companies, which usually run some to many servers, it is common to use a software which checks on the Windows Event Logs of *all* those servers.

So yes, the Windows Event Log is indeed really important.

14.3 How to investigate the Windows Event Log

In modern versions of Windows you just press the Win key and then type "Event". That brings up a list which contains at least "Event Viewer".

By default the Event Viewer displays all Event Logs on the current (local) machine. However, you can connect to another computer and investigate its Event Log, rights permitted. We keep it simple and focus just on the local Windows Event Log.

14.4 Terms used

From the Microsoft documentation: "Each log in the Eventlog key contains subkeys called event sources. The event source is the name of the software that logs the event. It is often the name of the application or the name of a subcomponent of the application if the application is large. You can add a maximum of 16,384 event sources to the registry. The Security log is for system use only. Device drivers should add their names to the System log. Applications and services should add their names to the Application log or create a custom log." [1]

14.5 Application log versus custom log

The vast majority of applications that write to the Windows Event Log write into "Windows Logs\Application", but if you wish you can create your own log under "Applications and services logs". However, be aware that for creating a custom log you need admin rights. Therefore creating a custom log is something that is usually done by the installer installing your software since that needs admin rights by definition anyway.

We keep it simple here and write to the "Application" log.

14.6 Let's do it

Copy `Z:\code\v13` to `Z:\code\v14`.

Loading WindowsEventLog

We are going to make `MyApp` writing to the Windows Event Log only when it runs as a Service. Therefore we need to load the module `WindowsEventLog` from within `MakeService.dyapp` (but not `MyApp.dyapp`):

```
...
Load ..\AplTree\OS
Load ..\AplTree\WindowsEventLog
Load ..\AplTree\Logger
...
```

Modify the INI file

We need to add a flag to the INI file that allows us to toggle writing to the Window Event Log:

[1]Microsoft on the Windows Event Log: https://msdn.microsoft.com/en-us/library/windows/desktop/aa363648(v=vs.85).aspx

```
...
[Ride]
Active      = 0
Port        = 4599
wait        = 1

[WindowsEventLog]
write       = 1 ; Has an affect only when it's running as a Service
```

Why can this be useful? During development, when you potentially run the Service in order to check what it's doing, you might not want the application to write to your Windows Event Log, for example.

Get the INI entry into the "Config" namespace

We modify the `MyApp.CreateConfig` function so that it creates `Config.WriteToWindowsEventLog` from that INI file entry:

```
∇ Config←CreateConfig isService;myIni;iniFilename
...
      :If isService
          Config.WatchFolders←⊃myIni.Get'Folders:Watch'
          Config.WriteToWindowsEventLog←myIni.Get'WINDOWSEVENTLOG:write'
      :Else
          Config.LogFolder←'expand'F.NormalizePath⊃Config.LogFolder myIni.Get'Folders:Logs'
          Config.WriteToWindowsEventLog←0
      :EndIf
...
∇
```

The functions "Log" and "LogError"

For logging purposes we introduce two new functions, `Log` and `LogError`. First `Log`:

```
∇ {r}←{both}Log msg
 ⍝ Writes to the application's log file only by default.
 ⍝ By specifying 'both' as left argument one can force the fns to write
 ⍝ `msg` also to the Windows Event Log if Config.WriteToWindowsEventLog.
   r←⍬
   both←(⊂{0<⎕NC ⍵:⍎⍵ ⋄ ''}'both')∊'both' 1
   :If 0<⎕NC'MyLogger'
       MyLogger.Log msg
   :EndIf
   :If both
   :AndIf Config.WriteToWindowsEventLog
       :Trap 0    ⍝ Don't allow logging to break!
           MyWinEventLog.WriteInfo msg
       :Else
           MyLogger.LogError'Writing to the Windows Event Log failed for:'
           MyLogger.LogError msg
       :EndTrap
   :EndIf
∇
```

Note that this function always writes to the application's log file. By specifying "both" as left argument one can enforce the function to also write to the Windows Event Log, given that `Config.WriteToWindowsEventLog` is true. That allows us to use `Log` for logging all events but errors, and to specify "both" as left argument when we want the function to record the Service starting, pausing and stopping. In other words, all calls to `MyLogger.Log` will be replaced by `Log`, although some calls require "both" to be passed as left argument.

We also introduce a function `LogError`:

```
∇ {r}←LogError(rc msg)
 ⍝ Write to **both** the application's log file and the Windows Event Log.
   MyLogger.LogError msg
   :If Config.WriteToWindowsEventLog
       :Trap 0
           MyWinEventLog.WriteError msg
       :Else
           MyLogger.LogError'Could not write to the Windows Event Log:'
           MyLogger.LogError msg
       :EndTrap
   :EndIf
∇
```

Note that the `Logger` class traps any errors that might occur. The `WindowsEventClass` does not do this, and the calls to `WriteInfo` and `WriteError` might fail for all sorts of reasons: invalid data type, invalid depth, lack of rights, you name it. Therefore both `Log` and `LogError` trap any errors and write to the log file in case something goes wrong. Note also that in this particular case it's okay to trap all possible errors (0) because we cannot possibly foresee what might go wrong. In a real-world application you still want to be able to switch this kind of error trapping of via an INI entry etc.

In case of an error we now want the function `LogError` to be called, so we change `SetTrap` accordingly:

```
∇ trap←{force}SetTrap Config
...
  #.ErrorParms.returnCode←EXIT.APPLICATION_CRASHED
  #.ErrorParms.(logFunctionParent logFunction)←⎕THIS'LogError'
  #.ErrorParms.windowsEventSource←'MyApp'
...
∇
```

Now it's time to replace the call to `MyLogger.Log` by a call to `Log` in the `MyApp` class; use the "Replace" feature of the editor in order to achieve that.

There are however three functions where we need to add `'both'` as left argument:

```
∇ {r}←MainLoop port;S
  r←θ
  'both'Log'"MyApp" server started'
  S←#.ServiceState
  :Repeat
      CheckForRide 0 port
      LoopOverFolder θ
      :If ('both'∘Log S.CheckServiceMessages)S.IsRunningAsService
          'both'Log'"MyApp" is about to shut down...'
          :Leave
      :EndIf
      ⎕DL 2
  :Until 0
 ⍝Done
∇
```

Note that we have to make use of the "compose" (∘) operator here: only by "gluing" the left argument (`'both'`) to the function name with the compose operator can we make sure that everything that's passed on to the `Log` function will be written not only to the log file but also to the Windows Event Log when `ServiceState` is managing the communication between the SCM and the application.

The second function to be changed is `Off`:

```
    ∇ Off exitCode
      :If exitCode=EXIT.OK
          'both'Log'Shutting down MyApp'
      :Else
```

Now we change `Initial`: in case the application is running as a service we let `Initial` create an instance of `WindowsEventLog` and return it as part of the result.

```
∇ r←Initial isService;parms;Config;MyLogger;MyWinEventLog
⍝ Prepares the application.
  #.⎕IO←1 ⋄ #.⎕ML←1 ⋄ #.⎕WX←3 ⋄ #.⎕PP←15 ⋄ #.⎕DIV←1
  Config←CreateConfig isService
  Config.ControlFileTieNo←CheckForOtherInstances θ
  CheckForRide(0≠Config.Ride)Config.Ride
  MyLogger←OpenLogFile Config.LogFolder
  Log'Started MyApp in ',F.PWD
  Log 2 ⎕NQ'#' 'GetCommandLine'
  Log↓⎕FMT Config.∆List
  r←Config MyLogger
  :If isService
      MyWinEventLog←⎕NEW #.WindowsEventLog(,⊂'MyAppService')
      parms←#.ServiceState.CreateParmSpace
      parms.logFunction←'Log'
      parms.logFunctionParent←⎕THIS
      #.ServiceState.Init parms
      r,←MyWinEventLog
  :EndIf
∇
```

`Initial` is called by `RunAsService` and `StartFromCmdLine` but because the result of `Initial` remains unchanged if the application is not running as a Service we need to amend just `RunAsService`. We localize `MyWinEventLog` (the name of the instance) and change the call to `Initial` since it now returns a three-item vector:

```
∇ {r}←RunAsService(earlyRide ridePort);⎕TRAP;MyLogger;Config;∆FileHashes;MyWinEventLog
 ⍝ Main function when app is running as a Windows Service.
...
  ⎕TRAP←#.HandleError.SetTrap ⍬
  (Config MyLogger MyWinEventLog)←Initial 1
  ⎕TRAP←(Config.Debug=0)SetTrap Config
...
∇
```

Does it still work?

Having made all these changes we should check whether the basics still work:

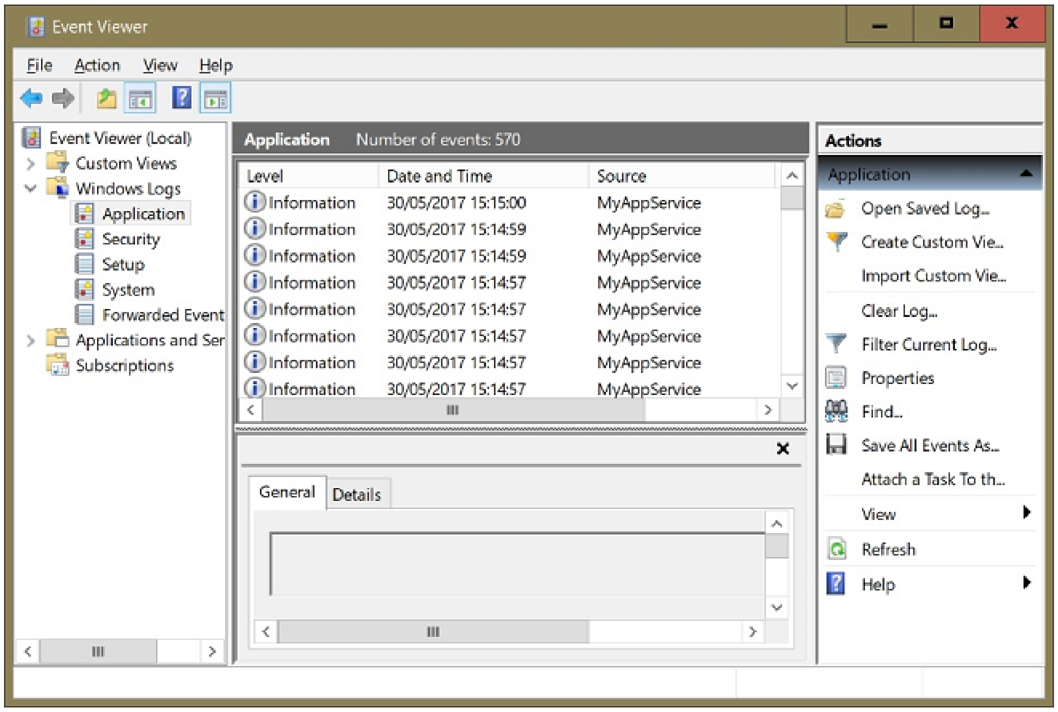
1. Double-click `Make.bat` in order to re-compile the EXE.
2. Double-click `MyApp.dyapp`. This assembles the workspace, including the test cases.
3. Answer the question whether all test cases shall be executed with "y".

Ideally the test cases should pass.

Now it's time to run the test cases for the Service:

1. Open a console window with admin rights.
2. Navigate to the `v13\` folder.
3. Call `MakeService.dyapp`.
4. Execute `TestsForServices.GetHelpers`.
5. Call `TestsForServices.RunDebug 0`.

Now start the Event Viewer; you should see something like this:



Level	Date and Time	Source
Information	30/05/2017 15:15:00	MyAppService
Information	30/05/2017 15:14:59	MyAppService
Information	30/05/2017 15:14:59	MyAppService
Information	30/05/2017 15:14:57	MyAppService
Information	30/05/2017 15:14:57	MyAppService
Information	30/05/2017 15:14:57	MyAppService
Information	30/05/2017 15:14:57	MyAppService
Information	30/05/2017 15:14:57	MyAppService

The Windows Event Log

You might need to scroll down a bit.

Adding a test case

We shall add a test case that checks whether the new logging feature works. For that we introduce `Test_03`:

```
∇ R←Test_03(stopFlag batchFlag);⎕TRAP;MyWinLog;noOfRecords;more;rc;records;buff
  ⍝ Start & stop the service, then check the Windows Event Log.
  ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
  R←∆Failed

  MyWinLog←⎕NEW #.WindowsEventLog(,⊂'MyAppService')
  noOfRecords←MyWinLog.NumberOfLogEntries

  (rc more)←∆Execute_SC_Cmd'start'
  →FailsIf 0≠rc
  ∆Pause 1
  (rc more)←∆Execute_SC_Cmd'query'
  →FailsIf 0=∨/'STATE : 4 RUNNING'⍷#.APLTreeUtils.dmb more
  ∆Pause 2

  (rc more)←∆Execute_SC_Cmd'stop'
  →FailsIf 0≠rc
  ∆Pause 2

  records←(noOfRecords-10)+⍳(MyWinLog.NumberOfLogEntries+10)-noOfRecords
  buff←↑MyWinLog.ReadThese records
  →PassesIf∨/,'"MyApp" server started '⍷buff
  →PassesIf∨/,'Shutting down MyApp'⍷buff

  R←∆OK
∇
```

Notes:

1. First we save the number of records currently saved in the Windows Event Log "Application".
2. We then start and stop the server in order to make sure that we get some fresh records written.
3. We then read the number of records plus 10 (others write to the Windows Event Log as well) and investigate them.

14.7 Tips, tricks and traps

No doubt you feel now confident with the Windows Event Log, right? Well, keep reading:

- When you create a new source in a (new) custom log then in the Registry the new log is listed as expected but it has *two* keys, one carrying the name of the source you intended to create and a second one with the same name as the log itself. In the Event Viewer however only the intended source is listed.
- The names of sources must be *unique* across *all* logs.
- Only the first 8 characters of the name of a source are really taken into account; everything else is ignored. That means that when you have a source `S1234567_1` and you want to register `S1234567_2` you will get an error "Source already exists".
- When the Event Viewer is up and running and you either create or delete a log or a source and then press F5 then the Event Viewer GUI flickers, and you might expect that to be an indicator for the GUI having updated itself; however, that's not the case, at least not at the time of writing (2017-03). You have to close the Event Viewer and re-open it to actually see your changes.
- Even when your user ID has admin rights and you've started Dyalog in elevated mode ("Run as administrator" in the context menu) you *cannot* delete a custom log with calls to `WinReg` or `WinRegSimple` (the APLTree classes that deal with the Windows Registry). The only way to delete custom logs is with the Registry Editor: go to the key

  `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog\`

  and delete the key(s) (=children) you want to get rid of. It's not a bad idea to create a system restore point [2] before you do that. By the way, if you never payed attention to System Restore Points you really need to follow the link because under Windows 10 System Restore Points are not generated automaticelly by default any more; they need to be switched on explicitly, and they really should.

[2]Details about System Restore Point: https://en.wikipedia.org/wiki/System_Restore

- Once you have written events to a source and then deleted the log the source pretends to belong to, the events remain saved anyway. They are just not visible anymore. That can be proven by re-creating the log: all the events make a come-back and show up again as they did before.

  If you really want to get rid of the logs then you have to select the "Clear log" command from the context menu in the Event Viewer (tree only!) before you delete the log.
- If you want to analyze the contents of a log in APL you will find the instance methods `Read` (which reads the whole log) and `ReadThese` (which takes line numbers and reads just those specified) useful.

15. The Windows Registry

15.1 What is it, actually?

We cannot say it any better than the Wikipedia [1]:

> The Registry is a hierarchical database that stores low-level settings for the Microsoft Windows operating system and for applications that opt to use the Registry. The kernel, device drivers, services, Security Accounts Manager (SAM), and user interface can all use the Registry. The Registry also allows access to counters for profiling system performance.
>
> In simple terms, The Registry or Windows Registry contains information, settings, options, and other values for programs and hardware installed on all versions of Microsoft Windows operating systems. For example, when a program is installed, a new subkey containing settings like a program's location, its version, and how to start the program, are all added to the Windows Registry.

The Windows Registry is still subject of heated discussions among programmers. Most hate it, some like it, but whatever your opinion is: you cannot ignore it.

Originally Microsoft designed the database as *the* source for any configuration parameters, be it for the operating system, users or applications. The Windows Registry will certainly remain to be the source for any OS-related pieces of information, but for applications we have seen a comeback of the old-fashioned configuration file, be at as an INI, an XML or a JSON file.

Even if you go for configuration files in order to configure your own application, you must be able to read and occasionally also to write to the Windows Registry, if only to configure Dyalog APL in order to make it suit your needs.

The Windows Registry can be useful for any application to store user specific data. For example, if you want to save the current position and size of the main form of your application for every user in order to be able to restore both position and size next time the application is started then the Windows Registry is the prefect place to store theses pieces of information. The key suggests itself:

```
HKCU\Software\MyApplication\MainForm\Posn
HKCU\Software\MyApplication\MainForm\Size
```

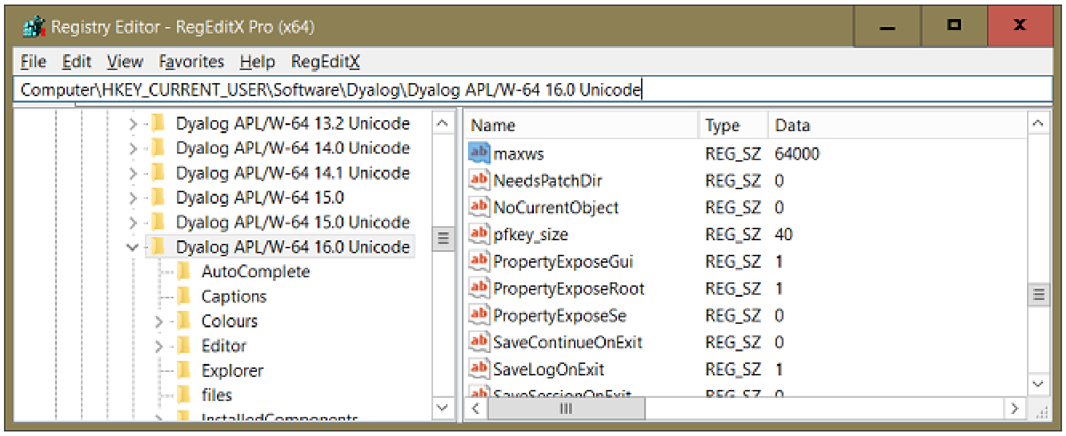
Note that `HKCU` is a short cut for "HKey Current User". There are others, and we will discuss them.

[1] The Wikipedia on the Windows Registry:

15.2 Terminology

If you find the terminology strange: so do we, but it was invented by Microsoft and therefore defines the standard. That is the reason why we go for it: it makes it easier to understand the Microsoft documentation but also to understand others while talking about the Windows Registry. It also helps when you google for Registry tweaks because the guys posting the solution to your problem are most likely using Microsoft speech as well.

Why is the terminology strange? Because Microsoft uses common words but gives them unusual meaning when it comes to the Windows Registry. Let's look at an example. This is the definition of the MAXWS parameter for Dyalog 64 bit Unicode version 16:


Registry Editor - RegEditX Pro (x64)
File Edit View Favorites Help RegEditX

Computer\HKEY_CURRENT_USER\Software\Dyalog\Dyalog APL/W-64 16.0 Unicode

- Dyalog APL/W-64 13.2 Unicode
- Dyalog APL/W-64 14.0 Unicode
- Dyalog APL/W-64 14.1 Unicode
- Dyalog APL/W-64 15.0
- Dyalog APL/W-64 15.0 Unicode
- Dyalog APL/W-64 16.0 Unicode
  - AutoComplete
  - Captions
  - Colours
  - Editor
  - Explorer
  - files

Name	Type	Data
maxws	REG_SZ	64000
NeedsPatchDir	REG_SZ	0
NoCurrentObject	REG_SZ	0
pfkey_size	REG_SZ	40
PropertyExposeGui	REG_SZ	1
PropertyExposeRoot	REG_SZ	1
PropertyExposeSe	REG_SZ	0
SaveContinueOnExit	REG_SZ	0
SaveLogOnExit	REG_SZ	1

Definition of maxws in the Windows Registry

The full path is:

```
Computer\HKEY_CURRENT_USER\Software\Dyalog\Dyalog APL/W-64 16.0 Unicode\maxws
```

We can get rid of "Computer" if it is the local machine, and we can shorten "HKEY_CURRENT_USER" as "HKCU". That leaves us with:

```
HKCU\Software\Dyalog\Dyalog APL/W-64 16.0 Unicode\maxws
```

That looks pretty much like a file path, doesn't it? So what about calling the different parts to the left of `maxws` folders? Well, that would be logical, therefore Microsoft did not do that. Instead they call `HKCU` a *key*, although the top level ones are sometimes called *root keys*. The other bits and pieces but `maxws` are called subkeys but sometimes just keys.

Okay, what's `maxws` then? Well it holds a value, so why not call it *key*? Ups, that's been taken already, but maybe "name" or "ID"? Well, Microsoft calls it a *value*. That's a strange name because is *has* as value, in our example the string `'64000'`.

To repeat: any given path to a particular piece of data stored in the Windows Registry consists of a key, one or more subkeys and a value that is associated with data.

There are a couple of things you should know:

- Keys and subkeys must not contain a backslash character (\) but values (!) and data may.
- A subkey may or may not have a *default value*. This is a piece of data that is associated with the subkey, not with a particular value.
- The Microsoft documentation clearly defines the word *key* for the top level only but later uses *key* and *subkey* interchangeably.
- According to the Microsoft documentation both keys and subkeys are case insensitive. That seems to imply that values are case sensitive but they are case insensitive, too.
- Key names are not localized into other languages, although values may be.

15.3 Data types

These days the Windows Registry offers quite a range of data types, but most of the time you can get away with these:

REG_SZ
The "string" data type. APLers call this a text vector. Both `WinReg` as well as `WinRegSimple` write text vectors as Unicode strings.

REG_DWORD
A 32-bit number.

REG_BINARY
Binary data in any form.

REG_MULTI_SZ
For an APLer this is a vector of text vectors. This data type was not available in the early days of the Windows Registry which is probably why it is not as widely used as you would expect.

REG_QWORD
A 64-bit number

There are more data types available, but they are not exactly popular.

15.4 Root keys

Any Windows Registry has just 5 root keys:

Root key	Shortcut
HKEY_CLASSES_ROOT	HKCR
HKEY_CURRENT_USER	HKCU
HKEY_LOCAL_MACHINE	HKLM
HKEY_USERS	HKU
HKEY_CURRENT_CONFIG	HKCC

From an application programmers point of view the HKCU and the HKLM are the most important ones, and usually the only ones they might actually write to.

With the knowledge you have accumulated by now you can probably get away for the rest of your life as an application programmer. If you want to know all the details we recommend the Microsoft documentation [2].

15.5 The class "WinRegSimple"

The APLTree class `WinRegSimple` is a very simple class that offers just three methods:

- `Read`
- `Write`
- `Delete`

It is also limited to the two data types `REG_SZ` and `REG_DWORD`.

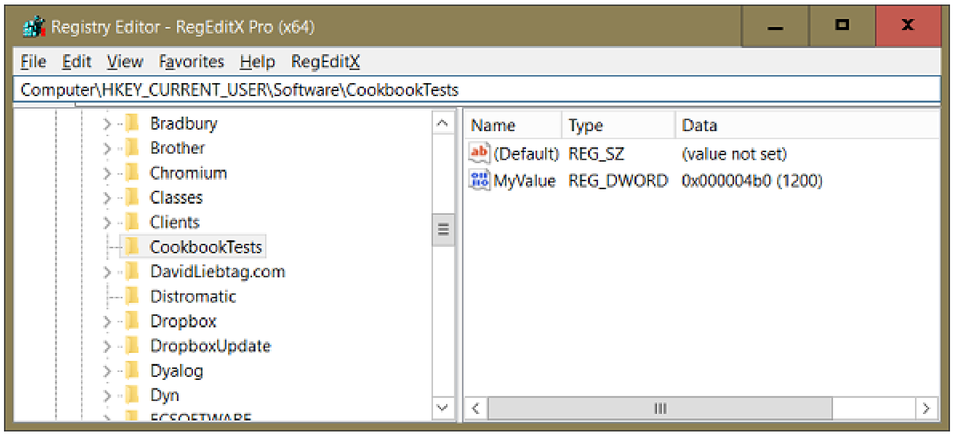
The class uses the Windows Scripting Host (WSH) [3]. It is available on all Windows systems although it can be switched off by group policies, something we have never seen in the wild.

If you just want to read a certain value then this – very small – class might be sufficient. For examples, in order to read the aforementioned `maxws` value:

```
      #.WinRegSimple.Read 'HKCU\Software\Dyalog\Dyalog APL/W-64 16.0 Unicode\maxws'
64000
```

You can create a new value as well as a new key with `Write`:

```
      #.WinRegSimple.Write 'HKCU\Software\Cookbooktests\MyValue' 1200
```



Computer\HKEY_CURRENT_USER\Software\CookbookTests

Name	Type	Data
(Default)	REG_SZ	(value not set)
MyValue	REG_DWORD	0x000004b0 (1200)

MyValue

You can also delete a subkey or a value, but a subkey must be empty:

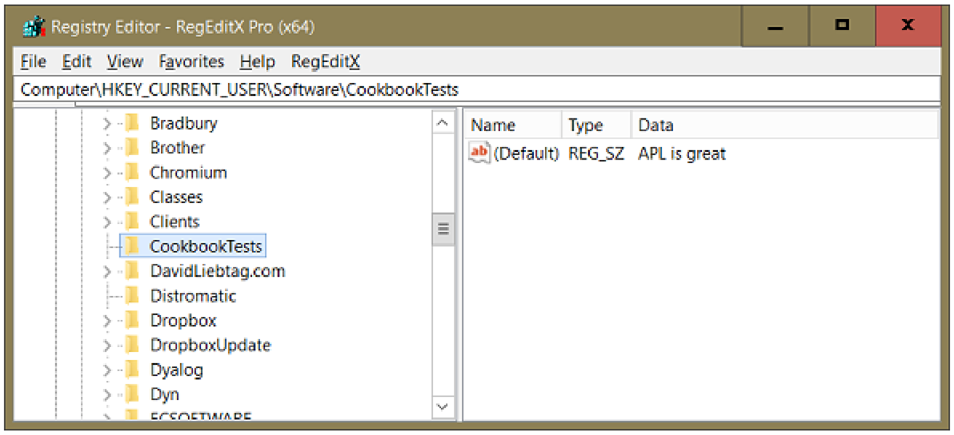
[2]Microsoft on the Windows Registry:
[3]The Wikipedia on the Windows Scripting Host:

```
      #.WinRegSimple.Delete 'HKCU\Software\Cookbooktests'
      #.WinRegSimple.Read 'HKCU\Software\Cookbooktests'
      #.WinRegSimple.Read'HKCU\Software\Cookbooktests\MyValue'
1200
      #.WinRegSimple.Delete 'HKCU\Software\Cookbooktests\MyValue'
Unable to open registry key "HKCU\Software\Cookbooktests\MyValue" for reading.
      #.WinRegSimple.Read'HKCU\Software\Cookbooktests\MyValue'
     ^
      #.WinRegSimple.Delete 'HKCU\Software\Cookbooktests\'
```

Note that in order to delete a subkey you must specify a trailing backslash.

You can also write the default value for a key. For that you must specify a trailing backslash as well. The same holds true for reading a default value:

```
      #.WinRegSimple.Write 'HKCU\Software\Cookbooktests\' 'APL is great'
      #.WinRegSimple.Read 'HKCU\Software\Cookbooktests\'
APL is great
```


Registry Editor - RegEditX Pro (x64)
File Edit View Favorites Help RegEditX
Computer\HKEY_CURRENT_USER\Software\CookbookTests
Bradbury
Brother
Chromium
Classes
Clients
CookbookTests
DavidLiebtag.com
Distromatic
Dropbox
DropboxUpdate
Dyalog
Dyn

Name	Type	Data
(Default)	REG_SZ	APL is great

Default values

Note that whether **`Write`** writes REG_SZ or a REG_DWORD depends on the data: a text vector becomes “REG_SZ” while a 32-bit integer becomes “REG_DWORD” though Booleans as well as smaller integers are converted to a 32-bit integer. Other data types are rejected.

If the **`WinRegSimple`** class does not suit your needs then have a look at the **`WinReg`** class. This class is much larger but has virtually no limitations at all.

To give you idea here the list of methods:

```
]adoc WinReg -summary
*** WinReg (Class) ***

Shared Fields:
  ERROR_ACCESS_DENIED
  ...
  REG_SZ
Shared Methods:
  Close
  CopyTree
  DeleteSubKeyTree
  DeleteSubKey
  DeleteValue
  DoesKeyExist
  DoesValueExist
  GetAllNamesAndValues
  GetAllSubKeyNames
  GetAllValueNames
  GetAllValues
  GetDyalogRegPath
  GetErrorAsStringFrom
  GetString
  GetTreeWithValues
  GetTree
  GetTypeAsStringFrom
  GetValue
  History
  KeyInfo
  ListError
  ListReg
  OpenAndCreateKey
  OpenKey
  PutBinary
  PutString
  PutValue
  Version
```

15.6 Examples

We will use both the `WinReg` class and the `WinRegSimple` class for two tasks:

- Add a specific folder holding user commands to all versions of Dyalog APL installed on the current machine.
- Add pieces of information to the caption definitions for all dialog boxes of all versions of Dyalog APL installed on the current machine.

The functions we develop along the way as well as the variables we need can be found in the workspace `WinReg` in the folder `Z:\code\Workspaces\`.

Add user command folder

Let's assume we have a folder `C:\MyUserCommands`. We want to add this folder to the list of folders holding user commands. For that we must find out the subkeys of all versions of Dyalog installed on your machine:

```
∇ list←GetAllVersionsOfDyalog dummy
  ⍝ Returns a vector of text vectors with Registry subkeys for all
  ⍝ versions of Dyalog APL installed on the current machine.
    list←#.WinReg.GetAllSubKeyNames'HKCU\Software\Dyalog'
    ⍝ Get rid of "Installed components" etc:
    list←'Dyalog'{ω/⍨((ρα)↑[2]↑ω)∧.=α}list
∇
      ↑GetAllVersionsOfDyalog ⍬
Dyalog APL/W 14.1 Unicode
Dyalog APL/W 15.0 Unicode
Dyalog APL/W 16.0 Unicode
Dyalog APL/W-64 13.2 Unicode
Dyalog APL/W-64 14.0 Unicode
Dyalog APL/W-64 14.1 Unicode
Dyalog APL/W-64 15.0
Dyalog APL/W-64 15.0 Unicode
Dyalog APL/W-64 16.0 Unicode
```

That's step one. In the next step we need to write a function that adds a folder to the list of user command folders:

```
∇ {r}←path Add version;subkey;folders
   r←⍬
   subkey←'HKCU\Software\Dyalog\',version,'\SALT\CommandFolder'
   'Subkey does not exist'⎕SIGNAL 11/⍨1≠#.WinReg.DoesValueExist subkey
   folders←#.WinReg.GetString subkey
   folders←';'{¯1↓¨ω⊂⍨';'=¯1↓';',ω}folders,';'
   folders←(({(819⌶)ω}¨folders)≢¨⊂(819⌶)path)/folders ⍝ drop doubles
   folders←⊃{α,';',ω}/folders,⊂path
   #.WinReg.PutString subkey folders
∇
```

Let's check the current status:

```
      dyalogVersions←AllVersionsOfDyalog ''
      ⍪{#.WinReg.GetValue 'HKCU\Software\Dyalog\',⍵,'\SALT\CommandFolder'}¨dyalogVersions
 C:\...\Dyalog APL 14.1 Unicode\SALT\Spice;C:\T\UserCommands\APLTeam\
 C:\...\Dyalog APL 15.0 Unicode\SALT\Spice;C:\T\UserCommands\APLTeam\
 C:\...\Dyalog APL 16.0 Unicode\SALT\Spice;C:\T\UserCommands\APLTeam\
...
      'C:\MyUserCommands'∘Add¨dyalogVersions
      ⍪{#.WinReg.GetValue 'HKCU\Software\Dyalog\',⍵,'\SALT\CommandFolder'}¨dyalogVersions
      C:\..\Dyalog APL 14.1 Unicode\SALT\Spice;C:\T\UserCommands\APLTeam\;C:\MyUserCommands
C:\...\Dyalog APL 15.0 Unicode\SALT\Spice;C:\T\UserCommands\APLTeam\;C:\MyUserCommands
C:\...\Dyalog APL 16.0 Unicode\SALT\Spice;C:\T\UserCommands\APLTeam\;C:\MyUserCommands
...
      'C:\MyUserCommands'∘Add¨dyalogVersions
      ⍪{#.WinReg.GetValue 'HKCU\Software\Dyalog\',⍵,'\SALT\CommandFolder'}¨dyalogVersions
C:\...\Dyalog APL 14.1 Unicode\SALT\Spice;C:\T\UserCommands\APLTeam\;C:\MyUserCommands
C:\...\Dyalog APL 15.0 Unicode\SALT\Spice;C:\T\UserCommands\APLTeam\;C:\MyUserCommands
C:\...\Dyalog APL 16.0 Unicode\SALT\Spice;C:\T\UserCommands\APLTeam\;C:\MyUserCommands
```

Note that although we called `Add` twice the folder `C:\MyUserCommands` makes an appearance only once. This is because we carefully removed it before adding it.

Configure Dyalog's window captions

In Appendix 4 "Your development environment" we mention that if you run more than once instance of Dyalog in parallel then you want to be able to allocate any dialog box to the instance it was issued from. This can be achieved by adding certain pieces of information to certain entries in the Windows Registry. We talk about this subkey of, say, Dyalog APL/W-64 16.0 Unicode:

```
HKCU\Software\Dyalog\Dyalog APL/W-64 16.0 Unicode\Captions
```

If that subkey exists (after an installation it doesn't) then it is supposed to contain particular values defining the captions for all dialog boxes that might make an appearance when running an instance of Dyalog. So in order to configure all these window captions you have to add the subkey `Chapter` and the required values in one way or another. This is a list of values honoured by version 16.0:

Editor
Event_Viewer
ExitDialog
Explorer
FindReplace
MessageBox
Rebuild_Errors
Refactor
Session
Status
SysTray
WSSearch

Although it is not a big deal to add these values with the Registry Editor we do not recommend this, if only because when the next version of Dyalog comes along then you have to do it again.

Let's assume that you have a variable `captionValues` which is a matrix with two columns:

- `[;1]` is the name of a value
- `[;2]` is the definition of the caption

That's what `captionValues` may look like:

```
      ρ⎕←values
 Editor          {PID} {TITLE} {WSID}-{NSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
 Event_Viewer    {PID} {WSID} {PRODUCT}
 ExitDialog      {PID} {WSID} {PRODUCT}
 Explorer        {PID} {WSID} {PRODUCT}
 FindReplace     {PID} {WSID}-{SNSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
 MessageBox      {PID} {WSID} {PRODUCT}
 Rebuild_Errors  {PID} {WSID} {PRODUCT}
 Refactor        {PID} {WSID}-{SNSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
 Session         {PID} {WSID}-{NSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
 Status          {PID} {WSID} {PRODUCT}
 SysTray         {PID} {WSID}
 WSSearch        {PID} {WSID} {PRODUCT}
13 2
```

Again, this variable can be copied from the workspace `Z:\code\Workspaces\`. We are going to write this data to the Windows Registry for all versions of Dyalog installed on the current machine. For that we need a list with all versions of Dyalog installed on the current machine. For this we can use the function `GetAllVersionsOfDyalog` we've developed earlier in this chapter:

```
   dyalogVersions←GetAllVersionsOfDyalog ''
```

Now we write a function that takes a version and the variable `captionValues` as argument and creates a subkey `Captions` with all the values. This time we use `#.WinRegSimple.Write` for this:

```
∇ {r}←values WriteCaptionValues version;rk
[1]  r←⍬
[2]  rk←'HKCU\Software\Dyalog\',version,'\Captions\'
[3]  rk∘{#.WinRegSimple.Write(⍺,(1⊃⍵))(2⊃⍵)}¨↓values
∇
```

We can now write `captionValues` to all versions:

```
       captionValues∘WriteCaptionValues¨dyalogVersions
      ⍝ Let's check:
      rk←'HKCU\Software\Dyalog\Dyalog APL/W-64 16.0 Unicode\Captions'
      #.WinReg.GetTreeWithValues rk
0  HKCU\...\Captions\
1  HKCU\...\Editor          {PID} {TITLE} {WSID}-{NSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
1  HKCU\...\Event_Viewer    {PID} {WSID} {PRODUCT}
1  HKCU\...\ExitDialog      {PID} {WSID} {PRODUCT}
1  HKCU\...\Explorer        {PID} {WSID} {PRODUCT}
1  HKCU\...\FindReplace     {PID} {WSID}-{SNSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
1  HKCU\...\MessageBox      {PID} {WSID} {PRODUCT}
1  HKCU\...\Rebuild_Errors  {PID} {WSID} {PRODUCT}
1  HKCU\...\Refactor        {PID} {WSID}-{SNSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
1  HKCU\...\Session         {PID} {WSID}-{NSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
1  HKCU\...\Status          {PID} {WSID} {PRODUCT}
1  HKCU\...\SysTray         {PID} {WSID}
1  HKCU\...\WSSearch        {PID} {WSID} {PRODUCT}
```

https://en.wikipedia.org/wiki/Windows_Registry

https://msdn.microsoft.com/en-us/library/windows/desktop/ms724946(v=vs.85).aspx

https://en.wikipedia.org/wiki/Windows_Script_Host

Appendix 01: Windows environment variables

Overview

Windows comes with quite a number of environment variables. Those variables are helpful in addressing, say, a particular path without actually using a physical path. For example, on most PCs Windows is installed in C:\Windows, but this is by no means guaranteed. It is therefore much better to address this particular folder as `¯2 ⎕NQ # 'GetEnvironment' 'WINDIR'`.

Underneath you find some of the environment variables found on a Windows 10 system listed and explained.

> Keep in mind that the Dyalog Cookbook is usually referring to Windows 10 which in most cases is identicall with Windows 8 and 7. The Cookbook does not care about unsupported versions of Windows like Vista and earlier.

Notes:

- When something like "{this}" is part of a path then it means that this string has to be exchanged against something reasonable. For example, in "C:/Users/{yourName}/" the string "{yourName}" needs to be replaced by the value of `⎕AN`.
- The selection of environment variables discussed in this appendix is by no means complete.
- Under Windows, the names of environment variables is case insensitive.

Outdated?!

Some consider environment variables an outdated technology. We don't want to get involved into a religious argument here but we insist that enviroment variables will be round for a very long time, and that Windows relies on them. Also, they are standard in Linux and macOS.

The variables

AllUserProfile

Defaults to "C:\ProgramData". See ProgramData.

AppData

Defaults to “C:\Users\{yourName}\AppData\Roaming”.

Use this to store application specific data that is supposed to roam [4] with the user. An INI file might be an example.

See also **LocalAppData**.

CommonProgramFiles

Defaults to “C:\Program Files\Common Files”.

CommonProgramFiles(x86)

Defaults to “C:\Program Files (x86)\Common Files”.

CommonProgramW6432

Defaults to “C:\Program Files\Common Files”.

ComputerName

Carries the name of the computer.

ComSpec

Defaults to “C:\WINDOWS\system32\cmd.exe”.

ErrorLevel

This variable does not necessarily exist. If you execute `⎕OFF 123` in an APL application then this will set `ErrorLevel` to 123.

HomePath

Defaults to “\Users\{yourName}”.

LocalAppData

Defaults to “C:\Users\{yourName}\AppData\Local”.

Use this to store application specific data the is **not** supposed to roam [5] with the user. A log file might be an example. The reason is that when a user logs in all the data stored in %APPDATA% is copied over. A large log file might take significant time to be copied over with very little benefit.

See also **AppData**.

[4] https://en.wikipedia.org/wiki/Roaming_user_profile
[5] https://en.wikipedia.org/wiki/Roaming_user_profile

LogonServer:

Defaults to “ComputerName”. This carries the name of the computer your are logged on to. In case of your own desktop PC the values of `LogonServer` and `ComputerName` will be the same. In a Windows Server Domain however they will differ.

OS

Specifies the Operating System. Under Windows 10 you get “Windows_NT”.

Path

Specifies all the folders (separated by semicola) that the operating system should check in case the user enters something like `my.exe` into a console window and `my.exe` does not live in the current directory.

PathExt

The `PathExt` environment variable returns a list of the file extensions that the operating system considers to be executable, for example: “.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC”.

ProgramData

Defaults to “C:\ProgramData”. Use this for storing information that is application specific but needs write rights beyond installation. For Dyalog, this would actually be the right place to store the session file, workspaces and user commands.

ProgramFiles

Defaults to “C:\Program Files”. On a 64-bit version of Windows this is where 64-bit programs are installed. Note however that on a 32-bit version of Windows this points to ProgramFiles(x86).

ProgramFiles(x86)

Defaults to “C:\Program Files (x86)”. This is where 32-bit programs are installed.

ProgramW6432

Defaults to “C:\Program Files”. On a 64-bit version of Windows this path points to “ProgramFiles”. On a 32-bit version of Windows it also points to “ProgramFiles” which in turn points to ProgramFiles(x86).

For details see “WOW64 Implementation Detail” [6].

[6]https://msdn.microsoft.com/en-us/library/windows/desktop/aa384274(v=vs.85).aspx

PSModulePath

Defaults to “C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\”. This path is used by Windows Power Shell [7] to locate modules when the user does not specify the full path to a module.

Public

This defaults to “C:\Users\Public”. It contains folders like “Public Documents”, “Public Music”, “Public Pictures”, “Public Videos”, … well, you get the picture,

SystemRoot

Specifies the folder Windows is installed in. Defaults to “C:\WINDOWS”.

Temp

Points to the folder that holds temporary files and folders. Defaults to “C:\Users{username}\AppData\Local\Temp”. See also “TMP”.

TMP

Points to the folder that holds temporary files and folders. Defaults to “C:\Users\{username}\AppData\Local\Temp”. Note that the `GetTempFileName` API function (which is available as `FilesAndDirs.GetTempFilename`) will first look for the “TMP” environment variable and only if that does not exist for the “TEMP” variable.

Username

The username of the user currently logged on. Same as `⎕AN` in APL.

UserProfile

Defaults to “C:\Users\{username}”. That’s where all the stuff is saved that belongs to the user currently logged on. Note that this is kept apart from other user’s spying eyes by the operating system.

WinDir

Defaults to %SystemRoot%. Deprecated.

[7] https://en.wikipedia.org/wiki/PowerShell

Appendix 02: User commands

Overview

User commands are a great way to make utilities available to the developer without cluttering the workspace. They also allow you to have one code base for several Dyalog installations. Since its introduction they have proven to be be indispensable.

Whether you want to write your own user commands or to make use of any third-party user commands like those available from the APL wiki for download [8], you need to consider your options how to integrate non-Dyalog user commands into your development environment.

The default folder depends on your version of Dyalog of course, but you can always find out from a running instance of Dyalog APL:

```
      '"',(2⎕NQ # 'GetEnvironment' 'Dyalog'),'\SALT\spice\"'
"C:\Program Files\Dyalog\Dyalog APL-64 16.0 Unicode\SALT\spice\"
```

The above is the default folder for the 64-bit Unicode version of Dyalog 16.0 for all Dyalog user commands available within that version.

Using the default folder

If you want to keep life simple the obvious choice seems to be this folder: you just copy your user command into this folder and it becomes available straight away.

Simple may it be, but this is *not* recommended:

- When a new version of Dyalog comes along you need to copy your own stuff over.
- When you use more than one version of Dyalog in parallel you have to maintain several copies of your user commands.
- Because of Microsoft's security measures, writing to that folder requires admin rights.

For these reasons you are advised to use a different folder.

[8]Dyalog user commands from the APL wiki

Use your own dedicated folder

Let's assume that you have a folder `C:\MyUserCommands` that's supposed to hold all non-Dyalog user commands. Via the "Options > Configure" command you can select the "User Commands" tab and add that folder to the search path; don't forget to press the "Add" button once you have browsed to the right directory.

If you use several versions of Dyalog in parallel then you are advised *not* to add that folder via the configuration dialog box in each of those versions. Instead we recommend to write an APL function that adds the folder to all versions of Dyalog currently installed. See the chapter "The Windows Registry" where this scenario is used as an example.

Name clashes

If two user commands share the same name the last definition wins. You can achieve this only by having a user command "Foo" in two different scripts in different folders *with the same group name, or no group name at all!*

In other words, the full name of a user command is compiled by the group name (say "foo") and the user comand name (say "goo"): `]foo.goo`. However, as long as there is only one user command `goo` this will do nicely:

```
      ]goo
```

Group name

Group names are not only useful in order to avoid name clashes, they also allow user commands to be, well, grouped in a sensible way. Whether you should add your own user commands to any of the groups Dyalog comes with is a diffult question to answer. There are pros and cons:

Pros

- Keeping the number of groups small is a good idea.
- When a user command clearly belongs to such a group, where else should it go anyway?

Cons

- You might miss on a new Dyalog user command because of a *real* name clash.
- Your own stuff and the Dyalog stuff are mixed up.

Updates

Note that once you've copied a new user command into that folder it is available straight away, even in instances of Dyalog that have already been running. However, auto-complete does not know about the new user command until it was called for the first time in an already running instance. You can at any time execute `]ureset` to make sure that even auto-complete knows about it.

In case you change an existing user command, for example by modifying the parsing rules, you must execute `]ureset` in order to get access to the changes from any instance of Dyalog that has already been running by then.

Writing your own user commands

It's not difficult to write your own user commands, and there is an example script available that makes that easy and straightforward. However, if your user command is not too simple consider developing it as an independent application, living in a particular namespace (let's assume `Foo`) in a particular workspace (let's assume `Goo`). Then write a user command that creates a namespace local to the function in the user command script, copy the namespace `Foo` from the workspace `Goo` into that local namespace and finally run the required function. Make sure the workspace is a sibling of the user command script.

This approach has the advantage that you can develop and test your user commands independently from the user command framework. This is particularly important because changing a user command script from the Tracer is a bit dangerous; you will see more aplcores than under normal circumstances. On the other hand it is difficult to execute the user command without the user command framework calling it: you need those arguments and sometimes even variables that live in the parent (`##`).

You are therefore advised to make sure that no function in `Foo` relies on anything provided by the user command framework. Instead the calling function (`Run` in your user command) must pass such values as arguments to any functions in `Foo` called by `Run`. That makes it easy to test all the public functions in `Foo`. Of course you should have proper test cases for them.

The following code is a simple example that assumes the following conditions:

- It requires a single argument.
- It offers an optional switch `-verbose`.
- It copies `Foo` from `Goo` and then runs `Foo.Run`.

```
:Namespace  Foo
      ⎕IO←1 ⋄ ⎕ML←1

    ∇ r←List
      r←⎕NS''
      r.Name←'Foo'
      r.Desc←'Does this and that'
      r.Group←'Cookbook'
      r.Parse←'1 -verbose'
    ∇

    ∇ r←Run(Cmd Args);verbose;ref;path;arg
      ref←⎕NS''
      verbose←Args.Switch'verbose'
      path←⊃1 ⎕NPARTS ##.SourceFile
      arg←⊃Args.Arguments
      :Trap 11
          'Foo'ref.⎕CY path,'\Goo'
      :Else
          'Copy operation for "Foo" in "Goo" failed' ⎕Signal 11
      :EndTrap
      r←ref.Foo.Run arg verbose
    ∇

    ∇ r←Help Cmd
      r←⊂'Help for "Foo".'
      r,←⊂'This user command ...'
      r←,[0.5]r
    ∇

:EndNamespace
```

Notes:

- The user command refers to `##.SourceFile`; this is a variable created by SALT that carries the full path of the currently executed user command. By taking just the directory part we know where to find the workspace.
- We create an unnamed namespace and assign it to a variable `ref`. We make sure that `Foo` is copied *into* `ref` by executing `ref.⎕CY`.
- The user command's `Run` function extracts the argument and the optional flag and passes both of them as arguments to the function `Foo.Run`. That way `Foo` does not rely on anything but the arguments passed to its functions.
- We have implemented the user command as a namespace. It could have been a class instead but in this case that does not offer any benefits since all functions are public anyway.

The workspace `Goo` can be tested independently from the user command framework, and the workspace `Goo` might well hold test cases for the functions in ‘ Foo‘.

http://aplwiki.com//CategoryDyalogUserCommands

Appendix 03: The source code management system “acre”

Overview

Appendix 04: The development environment.

Configure your session

Most developers insist of twisting the development environment in one way or another:

- Make your favourite utilities available from within `⎕SE`.
- Add a menu to the session with often used commands.
- Define some function keys carrying out often used commands (not applicable with Ride).
- …

There are several ways to achieve that:

1. Modify and save a copy of the default session file (by default `def_{countryCode}.dse` in the installation directory) and twist the configuration so that this new DSE is loaded.
2. Modify and save a copy of the build workspace; that is typically something like `"C:\Program Files\Dyalog\...\ws\buildse.dws"`. Then use it to create your own tailored version of a DSE.

Both approaches have their own problems, the most obvious being that with a new version of Dyalog you start from scratch. However, there is a better way: save a function `Setup` in either `C:\Users\{UserName}\Documents\MyUCMDs\s`
or one of the SALT work directories and it will be executed when…

- a new instance of Dyalog is fired up as part of the SALT boot process.

  Note that the SALT boot process will be carried out even when the "Enable SALT callbacks" checkbox on the "SALT" tab of the "Configuration" dialog box is not ticked.
- the user command `]usetup` is issued.

  This means that you can execute the function at will at any time in order to re-initialise your environment.

The function may be saved in that file either on its own or as part of a namespace.

You might expect that saving a class script "Setup.dyalog" with a public shared function `Setup` would work as well but that's not the case.

SALT work directories

You can check which folders are currently considered SALT work directories by issuing `]settings workdir`.

You can add a folder `C:\Foo` with `]settings workdir ,C:\Foo`.

When called as part of the SALT boot process a right argument `'init'` will be passed. When called via `]usetup` then whatever is specified as argument to the user command will become the right argument of the `Setup` function.

The Dyalog manuals mention this feature only when discussing the user command `]usetup` but not anywhere near of how you can configure your environment; that's why we mention it here.

Note that if you want to debug any `Setup` function then the best way to do this is to make `⎕TRAP` a local variable of `Setup` and then add these lines at the top of the function:

```
[1] ⎕TRAP←0 'S'
[2] .
```

This will cause an error that stops execution because error trapping is switched off. This way you get around the trap that the SALT boot process uses to avoid `Setup` causing a hiccup. However, in case you change the function from the Tracer don't expect those changes to be saved automatically: you have to take care of that yourself.

The following code is an example for how you can put this mechanism to good use:

```
:Namespace Setup
⍝ Up to - and including - version 15.0 this script needs to go into:
⍝ "C:\Users\[username]\Documents\MyUCMDs"
⍝ Under 16.0 that still works but the SALT workdir folders are scanned as well.
  ⎕IO←1 ⋄ ⎕ML←1

∇ {r}←Setup arg;myStuff
  r←⍬
  'MyStuff'⎕SE.⎕CY 'C:\MyStuff'
  ⎕SE.MyStuff.DefineMyFunctionKeys ⍬
  EstablishOnDropHandler ⍬
∇

∇ {r}←EstablishOnDropHandler dummy;events
  r←⍬
  events←''
  events,←⊂'Event' 'DropObjects' '⎕se.MyStuff.OnDrop'
  events,←⊂'Event' 'DropFiles' '⎕se.MyStuff.OnDrop'
  events,←⊂'AcceptFiles' 1
  events∘{⍵ ⎕WS ¨⊂⍺}'⎕se.cbbot.bandsb2.sb' '⎕se.cbbot.bandsb1.sb'
∇
```

```
:EndNamespace
```

We assume that in the workspace `MyStuff` there is a namespace `MyStuff` that contains at least two functions:

1. `DefineMyFunctionKeys`; this defines the function keys.
2. `OnDrop`; a handler that handles “DropObject” and “DropFiles” events on the session’s status bar.

This is how the `OnDrop` function might look like:

```
OnDrop msg;⎕IO;⎕ML;files;file;extension;i;target
⍝ Handles files dropped onto the status bar.
 ⎕IO←1 ⋄ ⎕ML←1
 files←3⊃msg
 :For file :In files
     extension←1(819⌶)3⊃1 ⎕NPARTS file
     :Select extension
     :Case '.DWS'
         ⎕←'      )XLOAD ',{b←' '∊⍵ ⋄ (b/'"'),⍵,(b/'"')}file
     :Case '.DYALOG'
         :If 9=⎕NC'⎕SE.SALT'
             target←((,'#')≢,1⊃⎕NSI)/' -Target=',(1⊃⎕NSI),''''
             ⎕←'      ⎕SE.SALT.Load ''',file,'',target
         :EndIf
     :Else
         :If 'APLCORE'{α≡1(819⌶)(⍴α)↑⍵}2⊃⎕NPARTS file
             ⎕←'      )COPY ',{b←' '∊⍵ ⋄ (b/'"'),⍵,(b/'"')}file,'.'
         :Else
             :If ⎕NEXISTS file
                 ⎕←{b←' '∊⍵ ⋄ (b/'"'),⍵,(b/'"')}file
             :Else
                 ⎕←file
             :EndIf
         :EndIf
     :EndSelect
 :EndFor
```

What this handler does depends on what extension the file has:

- For `.dyalog` it writes a SALT load statement to the session.

  If the current namespace is not # but, say, `Foo` then `-target=Foo` is added.
- For `.dws` it writes an)XLOAD statement to the session.
- If the filename contains the string `aplcore` then it writes a)COPY statement for that aplcore with a trailing dot to the session.
- For any other files the fully qualified filename is written to the session.

When you start Dyalog with admin rights then it's not possible to drop files onto the status bar. That's because Microsoft considers drag'n drop too dangerous for admins. Funny; one would think it's a better strategyy to leave the dangerous stuff to the admins.

How you configure your development environment is of course very much a matter of personal preferences. However, you might consider to load a couple of scripts into `⎕SE` from within `Setup.dyalog`; the obvious candidates for this are `APLTreeUtils`, `FilesAndDirs`, `OS`, `WinSys`, `WinRegSimple` and `Events`. That would allow you to write user commands that can reference them with, say, `⎕SE.APLTreeUtils.Split`.

Define your function keys

Defining function keys is of course not exactly a challenge. Implementing it in a way that is actually easy to read and maintain *is* a challenge.

```
:Namespace FunctionKeyDefinition

    ∇ {r}←DefineFunctionKeys dummy;⎕IO;⎕ML
      ⎕IO←1 ⋄ ⎕ML←3
      r←⍬
      ⎕SHADOW⊃list←'LL' 'DB' 'DI' 'ER' 'LC' 'DC' 'UC' 'RD' 'RL' 'RC' 'Rl' 'Ll' 'CP' 'PT' 'BH'
      ⍎¨{⍵,'←⊂''',⍵,''''}¨list
      r⍪←'F01'('')('(Reserved for help)')
      r⍪←'F02'(')WSID',ER)(')wsid')
      r⍪←'F03'('')('Show next hit')                ⍝ Reserved for NX
      r⍪←'F04'('⎕SE.Display ')('Call "Display"')
      r⍪←'F05'(LL,'→⎕LC+1 ⍝ ',ER)('→⎕LC+1')
      r⍪←'F06'(LL,'→⎕LC ⍝',ER)'→⎕LC'
      ...
:EndNamespace
```

This approach first defines all special shortcuts – like `ER` for <enter> etc. – as local variables; using `⎕SHADOW` avoids the need for maintaining a long list of local variables. The statement `⍎¨{⍵,'←⊂''',⍵,''''}¨list` assigns every name as an enclosed text string to itself like `ER←⊂'ER'`. Now we can use `ER` rather than `(⊂'ER)` which improves readability.

A definition like `LL,'→⎕LC ⍝',ER` reads as follows:

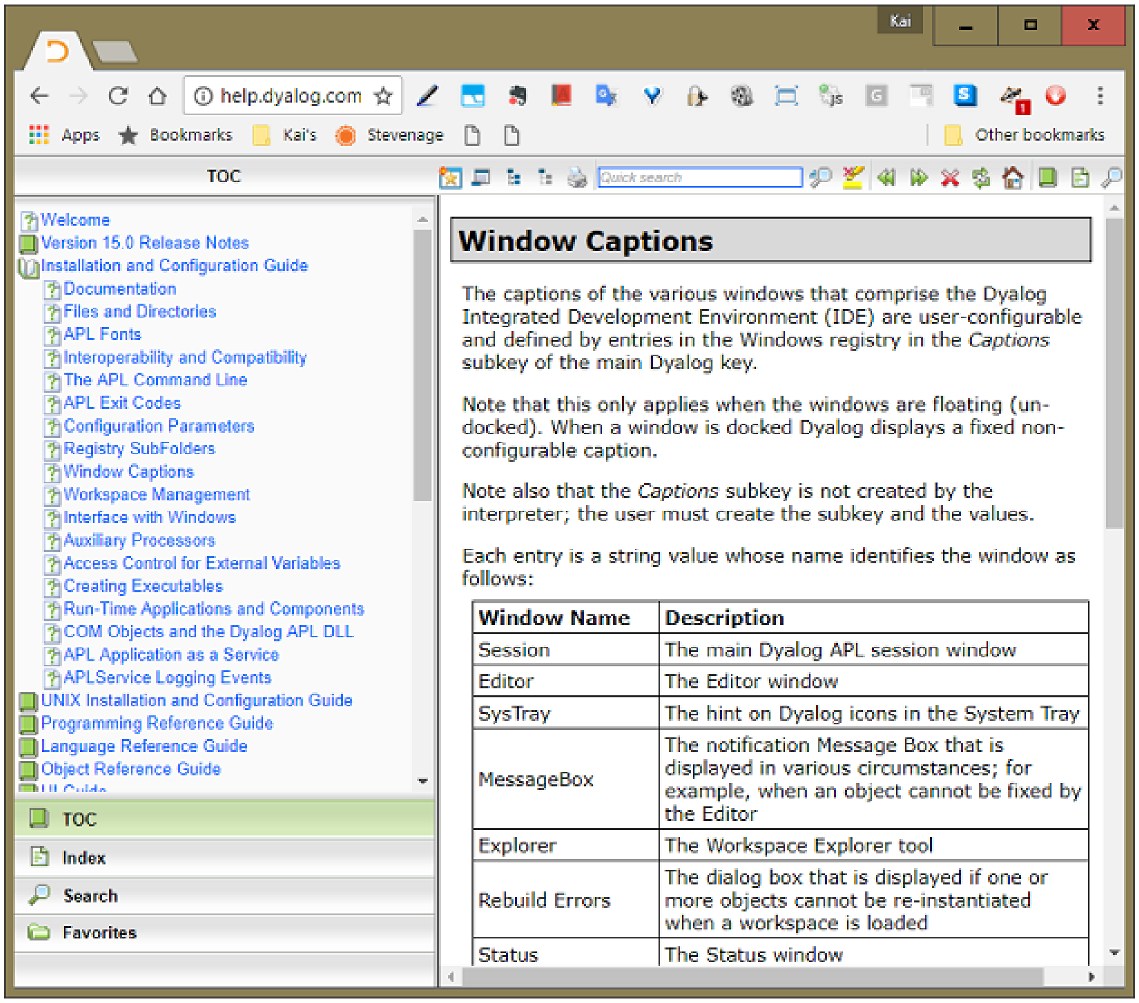
- `LL` positions the cursor to the very left of the current line.
- `→⎕LC ⍝` is then written to the session, meaning that everything that was already on that line is now on the right of the `⍝` and therefore has no effect.
- `ER` then executes <enter>, meaning that the statement is actually executed.

If you don't know what `LL` and `ER` actually are read the page "Keyboard shortcuts" in the "UI Guide".

Windows captions

If you always run just one instance of the interpreter you can safely ignore this.

If on the other hand you run occasionally (let alone often) more than one instance of Dyalog in parallel then you are familiar with how it feels when all of a sudden an unexpected dialog box pops up, be it an aplcore or a message box asking "Are you sure?" when you have no idea what you are expected to be sure about, or which instance has just crashed. There is a way to get around this. With version 14.0 windows captions became configurable. This is a screenshot from the online help:



Window Captions

The captions of the various windows that comprise the Dyalog Integrated Development Environment (IDE) are user-configurable and defined by entries in the Windows registry in the *Captions* subkey of the main Dyalog key.

Note that this only applies when the windows are floating (un-docked). When a window is docked Dyalog displays a fixed non-configurable caption.

Note also that the *Captions* subkey is not created by the interpreter; the user must create the subkey and the values.

Each entry is a string value whose name identifies the window as follows:

Window Name	Description
Session	The main Dyalog APL session window
Editor	The Editor window
SysTray	The hint on Dyalog icons in the System Tray
MessageBox	The notification Message Box that is displayed in various circumstances; for example, when an object cannot be fixed by the Editor
Explorer	The Workspace Explorer tool
Rebuild Errors	The dialog box that is displayed if one or more objects cannot be re-instantiated when a workspace is loaded
Status	The Status window

Dyalog's help on window captions

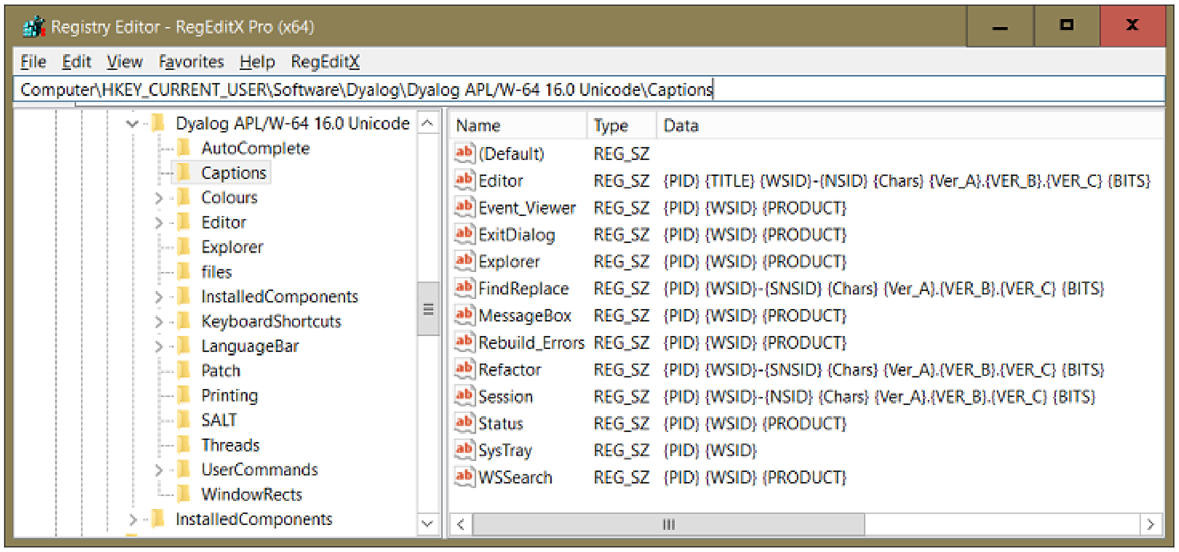
Help - online versus offline

There are pros and cons:

- Pressing F1 on something you need help with opens the offline help at the time of writing (2017-07).

- The online help is frequently updated by Dyalog.

We suggest you configure Windows captions in a particular way in order to overcome this problem. The following screen shot shows the definitions for all windows captions in the Windows Registry for version 16 in case you follow our suggestions:



Computer\HKEY_CURRENT_USER\Software\Dyalog\Dyalog APL/W-64 16.0 Unicode\Captions

Name	Type	Data
(Default)	REG_SZ	
Editor	REG_SZ	{PID} {TITLE} {WSID}-{NSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
Event_Viewer	REG_SZ	{PID} {WSID} {PRODUCT}
ExitDialog	REG_SZ	{PID} {WSID} {PRODUCT}
Explorer	REG_SZ	{PID} {WSID} {PRODUCT}
FindReplace	REG_SZ	{PID} {WSID}-{SNSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
MessageBox	REG_SZ	{PID} {WSID} {PRODUCT}
Rebuild_Errors	REG_SZ	{PID} {WSID} {PRODUCT}
Refactor	REG_SZ	{PID} {WSID}-{SNSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
Session	REG_SZ	{PID} {WSID}-{NSID} {Chars} {Ver_A}.{VER_B}.{VER_C} {BITS}
Status	REG_SZ	{PID} {WSID} {PRODUCT}
SysTray	REG_SZ	{PID} {WSID}
WSSearch	REG_SZ	{PID} {WSID} {PRODUCT}

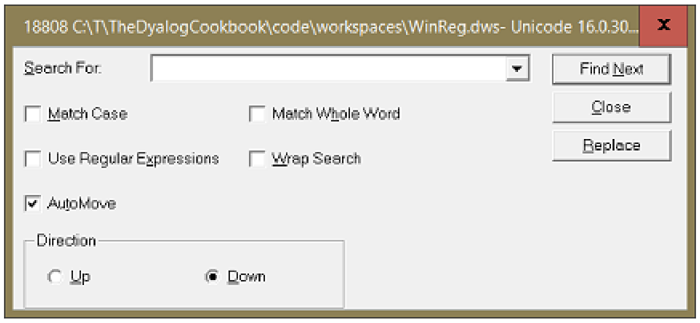
Windows Registry entries for "Window captions"

Notes:

- All definitions start with `{PID}` which stands for process ID. That allows you to identify which process a particular window belongs to, and even to kill that process if needs must.
- All definitions contain `{WSID}` which stands for the workspace ID.
- `{PRODUCT}` tells all about the version of Dyalog: version number, 32/64 and Classic/Unicode.

  You might not be interested in this in case you use just one version of Dyalog.

The other pieces of information are less important. For details refer to the page "Window captions" in the "Installation and Configuration Guide". These definitions make sure that most dialog boxes (there are a few exceptions) can be allocated to a particular Dyalog session with ease. This is just an example:



A typical dialog box

However, this cannot be configured in any way, you need to add subkeys and values to the Windows Registry. We do *not* suggest that you add or modify those caption with the Registry Editor. It is a better idea to write them by program, even if you deal with just one version of Dyalog at a time because soon there will be a new version coming along requiring you to carry out the same actions again. See the chapter “The Windows Registry” for how to solve this; that chapters uses this scenario as an example.

Workspace integrity, corruptions and aplcores

The *workspace* (WS) is where the APL interpreter manages all code and all data in memory. The Dyalog tracer / debugger has extensive edit-and-continue capabilities; the downside is that these have been known to occasionally corrupt the workspace. However, there are many other ways how the workspace may get corrupted:

- The interpreter might carry a bug.
- The user uses `⎕NA` incorrectly.
- WS FULL.
- …

The interpreter checks WS integrity every now and then; how often can be influenced by setting certain debug flags; see "The APL Command Line" in the documentation for details. Be warned that…

- the `-DW` flag slows an application down *extremely* even on very fast machines.
- `-Dc` and `-Dw` slow the interpreter down in any case, but the effect depends on the workspace size. You might not notice anything at all with, say, maxws=64MB but you will notice a delay with maxws=2GB.

When the interpreter finds that the WS is damaged it will create a dump file called "aplcore" and exit in order to prevent your application from producing (or storing) incorrect results.

Regularly rebuilding the workspace from source files removes the risk of accumulating damage to the binary workspace.

Note that an aplcore is useful in two ways:

- You can copy from it. Add a colon:

  ```
        )copy aplcore. myObj
  ```

  It's not a good idea to copy the whole workspace; after all something has been wrong with it.

  It may be fine to recover a particular object (or some objects) from it, although you would be advised to rebuild recovered objects from the source (for example via the clipboard) rather than using binary data recovered from an aplcore.
- Send the aplcore to Dyalog. It's kind of a dump, so they might be able to determine the cause of the problem. Naturally it helps when you can provide information about your last actions or, even better, reproduce the aplcore at will.

You can create an aplcore deliberately by executing:

```
      2 ⎕NQ '.' 'dumpws' 'C:\MyAplcore'
```

This might be a useful thing to do just before executing a line you already know will cause havoc in one way or another.

In order to create a “real” aplcore in the sense of corrupting the workspace this will do:

```
Crash;MEMCPY
:Trap 102
    ⎕NA'dyalog32|MEMCPY u4 u4 u4'
:Else
    ⎕NA'dyalog64|MEMCPY u4 u4 u4'
:EndTrap
MEMCPY 0 0 4
```

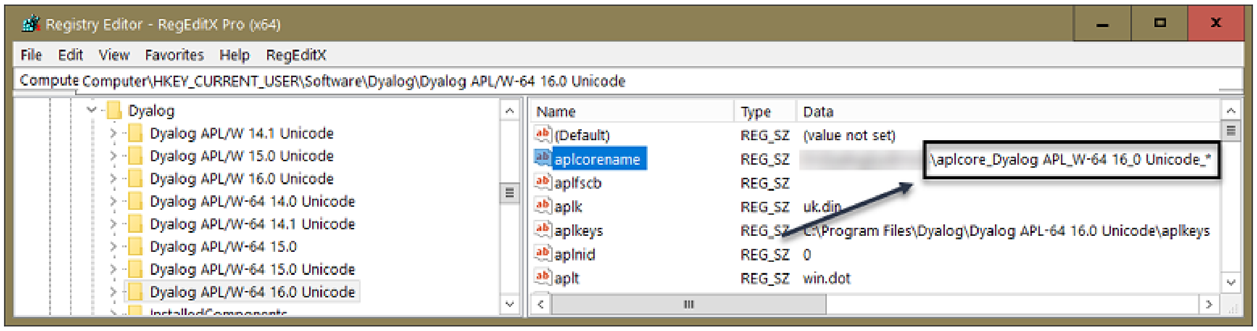
By default an aplcore is saved with the name `aplcore` in what is at that moment the current directory. This is not nice because it means that any aplcore might overwrite the last one. That can become particularly annoying when you try to copy from an aplcore with :

```
      )copy C:\MyAplcore.
```

but this might actually create another aplcore, overwriting the first one. Now it might well be too late to restrict the attempt to copy to what is most important to you: the object or objects you have worked on most recently.

If the aplcore is saved at all that is, because if the current directory is something like `C:\Program files\` then you won’t have the right to save into this directory anyway.

For that reason it is highly recommended to set the value `aplcorename` in the Windows Registry:



Registry Editor - RegEditX Pro (x64)

File Edit View Favorites Help RegEditX

Computer\HKEY_CURRENT_USER\Software\Dyalog\Dyalog APL/W-64 16.0 Unicode

- Dyalog
  - Dyalog APL/W 14.1 Unicode
  - Dyalog APL/W 15.0 Unicode
  - Dyalog APL/W 16.0 Unicode
  - Dyalog APL/W-64 14.0 Unicode
  - Dyalog APL/W-64 14.1 Unicode
  - Dyalog APL/W-64 15.0
  - Dyalog APL/W-64 15.0 Unicode
  - Dyalog APL/W-64 16.0 Unicode

Name	Type	Data
(Default)	REG_SZ	(value not set)
aplcorename	REG_SZ	[redacted]\aplcore_Dyalog APL_W-64 16_0 Unicode_*
aplfscb	REG_SZ	
aplk	REG_SZ	uk.din
aplkeys	REG_SZ	C:\Program Files\Dyalog\Dyalog APL-64 16.0 Unicode\aplkeys
aplnid	REG_SZ	0
aplt	REG_SZ	win.dot

Defining home and names of aplcores

This means that aplcores…

- are going to be saved in a folder of your choice.
- start their names with `aplcore_Dyalog APL_W-64 16_0 Unicode_`.
- will be numbered starting from 1; this as achieved by adding the trailing `*`.