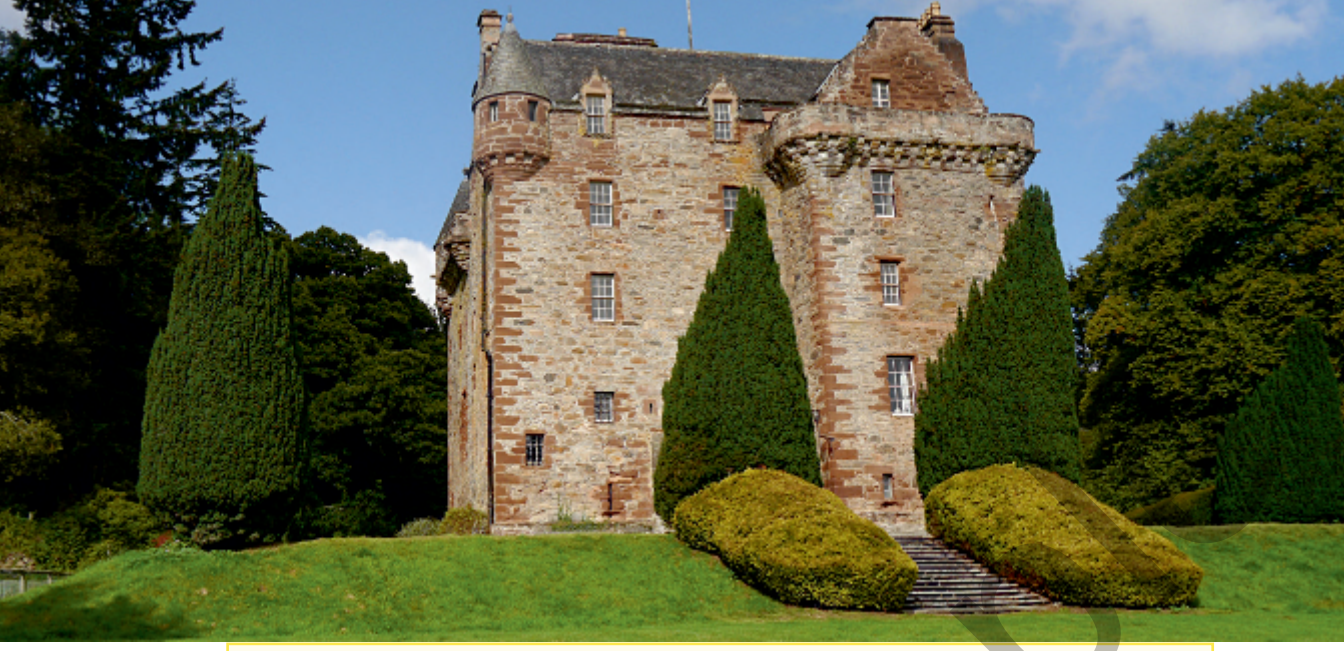

Developing a Core Library

The primary driver for this requirement is that we often deal with "files" be they sourced from a local or UNC networked location or the web. Sometimes, we may also want to deal with directories. It's a common requirement for our potential providers - even in dealing with manifest resources in implementing such as the extended functionality of persisting resources to disk files "somewhere" in a users' temporary path - as defined in the user profile on Windows. In this respect these "files" may be resource-based such as images and icons or may be Xml files and schemas as well as assemblies. Additionally, we've already seen how a Uri is a "common currency" in dealing with "files" in general - and directories, so we'll revisit our Uri extensions for the Type Providers to make them more useful in general-purpose usage - which will also benefit our proposed Dynamic Providers.

As our core library will be referenced by all of our subsequent assemblies then I'm also going to include some "useful" utilities and extensions as well as a range of types including, for example, our `IsoDate` and `IsoTime` types. As for the Common library of our Type Providers, this will then minimise the code in subsequent assemblies by providing a Core of common functionality.

Firstly, I want to run though a couple of scripts in the `Core` project of the `saTrilogy.Core` solution. I've placed these in a solution folder named Scripts and named them, appropriately, **Messing Around.fsx** and **More Messing Around.fsx**! These scripts cover some important as well as useful functionality that can then be incorporated into the source for the core assembly.

Windows Management Instrumentation Queries

Up to this point our dealings with Windows Management Instrumentation (WMI) has been limited to the use of performance counters. This only represents a fraction of the full functionality of WMI! Amongst other things, WMI monitors the hardware of your machine. WMI uses a *large*, proprietary database, buried in the guts of your Windows machine, to maintain information about your machine hardware as well as for everything else it does. Because the database is so large and contains a great diversity of information, it is segmented by what WMI calls a scope. The scope effectively defines a path into a segment of the overall database. The first component of the scope path is actually a server name where, often, you'll see this identified by just a full-stop and that means "the machine I'm running on". You can also get your machine name in .NET via the property `System.Environment.MachineName` and you've seen the usage of this in our ETW implementation. As it happens, you don't need "elevated privileges"/administrative authority to query much of the WMI database. However, if your scope path identifies a server other than "your own machine", then you'll need to be "connected" to that server and that implies an authority inherited by your Windows Login identity as would be characterised in the Active Directory.

In the scope path, following the target server specification, one then specifies the segment of the WMI database to which access is desired. There are a number of these but the one most commonly used for queries - including hardware-related queries, is the default segment called **cimv2**. The Distributed Management Task Force (DTMF)[7] states that the acronym **CIM** stands for the **Common Information Model** - and, presumably because its "common", that's why it's used as the default WMI path query segment - the v2 refers to "Version 2". The DTMF further has the following to say about the CIM...

> "CIM provides a common definition of management information for systems, networks, applications and services, and allows for vendor extensions. CIM's common definitions enable vendors to exchange semantically rich management information between systems throughout the network."

With the segment we can now address, if you will, a "table" that contains the information we're interested in. This is referred to as a WMI class. The class is wrapped by a WMI Provider of which there are four - as specified in the MSDN documentation...

Computer System Hardware Classes Hardware-related objects.

Operating System Classes Operating system related objects.

Performance Counter Classes Raw and calculated performance data from

performance counters.

WMI Service Management Classes Management for WMI.

We will target the Computer Systems Hardware provider within which there are a large number of "sub"-classes[54] of which we'll use the following-with descriptions from MSDN...

`Win32_NetworkConnection` Represents an active network connection in a Windows based environment

`Win32_Share` Represents a shared resource on a computer system running Windows. This may be a disk drive, printer, interprocess communication, or other shareable device.

`Win32_Volume` Represents an area of storage on a hard disk. The class returns local volumes that are formatted, unformatted, mounted, or offline. A volume is formatted by using a file system, such as FAT or NTFS, and might have a drive letter assigned to it. One hard disk can have multiple volumes, and volumes can span multiple physical disks.

With that, let's see how we formulate a WMI Query in F# and deal with its output; this is in the script **Messing Around.fsx** of the `Core` project in the `saTrilogy.Core` solution. Therein, we firstly reference and open a number of assemblies/namespaces and initialise Swenson's FsEye. Thereafter, I've "extended" the Uri class, as in our previous use, for a number of members such as `Mode`, `ModeSpecificPath`, `Server` and `FileName` and declared some "sample" Uri instances.

Let's firstly consider a value type that will evaluate the default network server - this implies that you are network connected and "logged into" one or more servers - the fact of logging into a server being sufficient to create an "active network connection". We use the code...

```
let DefaultNetworkServer =
  match
    (new ManagementClass(
      @"\\.\root\cimv2",
       "Win32_NetworkConnection", new ObjectGetOptions())
    ).GetInstances()
     .OfType<ManagementBaseObject>()
     .Select
       ( fun mo ->
           Regex.Match(
             mo.Item("RemoteName").ToString(),
             @"[^\\]+").Value )
    |> Seq.tryHead
    with
```

```
    | Some(value) when not <| String.IsNullOrWhiteSpace
...   value ->
16        value
    | _ -> System.Environment.MachineName
```

The `ManagementClass` class, in the `System.Management` namespace takes three arguments...

1. The WMI scope path comprising the `\root` of the server specification `\\.` and the `cimv2` segment. If the server is not specified in the scope path then it is assumed that one is referring to the "local" machine - as we are in our specification.
2. The WMI class against which we wish to target the query.
3. Generally, herein you'd specify a SELECT statement as, for example, a Query Expression or similarly to T-SQL. I'm being lazy - I'm just telling WMI to give me back an IEnumerable of the "keyed" values in the target database segment by specifying a `new GetObjectOptions()` - the "keys" are called "options". You're going to have to check the MSDN documentation to find out "which" option you want for a particular query! This can be a tedious process so I would refer you to the WMI Explorer software package that you can download from CodePlex (see "Software & Hardware" on page xv) .

Given that this "query" actually returns a collection of "results" I then issue the `GetInstances()` method of the `ManagementClass` class to yield a collection of `ManagementObjectCollection` objects. As usual with .NET entities this resembles a collection of objects so I then use Linq to strongly type each object in the collection as a `ManagementBaseObject`. With Linq I then re-shape the collection using a Linq `Select`. In the lambda expression of the Linq `Select` I use a Regular Expression to match only those items whose remote name starts with a backslash. In dealing with the `ManagementBaseObject` item, with the option name (key) of `RemoteName`, the result will be the server name we're connected to. If you do have a network connection then, in FSI, you can run this query piecemeal to inspect each stage of the match expression's output.

The Linq `Select` output sequence may have zero, one or more entries depending upon the natures of your machines network configuration. Since I want the "default" server then I just to a `Seq.tryHead` to pull the first server (or `None`) then push it through my pattern match option cases where, if there is no server, I'll instead output the name of the local machine.

Next, let's look at network shares; for this one may also check if the user has administrative authority to determine, if you choose to do so, whether or not to include the administrative shares - those share targets whose path terminates with a dollar character. In any event, I'll verify that one has the authority to access a particular share. For this we use the following...

```
let RunningAsAdministrator =
  ( let user = WindowsIdentity.GetCurrent()
  let principal = new WindowsPrincipal(user)
  principal.IsInRole(WindowsBuiltInRole.Administrator) )

type DirectoryInfo with
  static member AccessPermitted path =
    match Directory.Exists path with
    | true ->
        let di = new DirectoryInfo(path)
        let acl = di.GetAccessControl()
        let rules =
          acl.GetAccessRules(
            true,true,typeof<SecurityIdentifier>)
        match
          rules.OfType<FileSystemAccessRule>()
          |> Seq.tryPick
              ( fun r ->
                  if r.AccessControlType =
                     AccessControlType.Allow
                  then Some r else None)
        with
        | Some(_) -> true
        | None -> false
    | false -> false
```

We now define a `NetworkShares` function which will take an argument being the server name of the machine to query for any existing, authorised shares...

```
let NetworkShares(targetMachine : string option) =
  let wmiPath =
    match targetMachine with
    | Some(value) -> """\\""" + value + """\root\cimv2"""
    | None -> """\\.\root\cimv2"""
  (new ManagementClass(
        """\\.\root\cimv2""","Win32_Share",
        new ObjectGetOptions())
  ).GetInstances()
   .OfType<ManagementBaseObject>()
   .Select
     ( fun mo ->
         sprintf """\\%s\%s"""
         mo.ClassPath.Server
         <| mo.Item("Name").ToString() )
  |> Seq.choose
```

```
        ( fun path ->
18          match DirectoryInfo.AccessPermitted path with
            | true -> Some(new Uri(path))
20          | false -> None )
```

You can see here that the scope path is constructed to include a server since, if the option argument is not specified, then I'll explicitly set the local machine. The query is the same as for the default server just that we're now targeting the WMI class `Win32_Share` and looking for the "option" `Name`. The Linq `Select` simply formulates a UNC path given the server name and each share name and the subsequent `Seq.choose` will only take elements that have "`Some` value" and, for these, we create a Uri from the UNC share string.

For our third WMI query I want to determine the cluster size of a disk upon which a targeted Uri exists. I will use this cluster size (or block size) for two purposes...

1. For asynchronous file access I would like to specify a buffer size; the target disk file is "read" in sections and/or each section is written to a target - such as a memory stream. Setting this asynchronous block size is often a hit an miss affair, however, to me, logic indicates that a "sensible" choice would be the block size which the disk upon which the target resides is formatted. The block size determines "how" a file is stored on disk across a number of blocks. Each block can only be "assigned" to a single file so whilst a file can span many blocks no two files can "share" any single block. Some "shops", when storing video or other "large" files make sure that they're maintained on disks that have a formatted block size of 64KB to optimise Disk I/O at the potential expense of "some" wasted space.

   Most files - certainly from an NTFS viewpoint, reside on disks with the default block size of 4KB or 4096 bytes. Even if your file was only, say, 123 bytes in size, then its "size" on disk will actually be 4096 bytes! This is how "space is wasted" on a drive and why some people will distribute different "types" of files on drives with block sizes more appropriate to the file sizes contained thereupon. It seems to me that an appropriate asynchronous buffer size, in order to try and optimise disk I/O should be the block size or a multiple thereof.

2. The foregoing may have caused you to raise an eyebrow; one generally assumes, without fairly detailed hardware knowledge, that the file size **is** the file size! This is not so and the situation gets even more complex when one deals with disks that are "compressed". The former issues we'll address in the next topic.

The WMI class `Win32_Volume` offers one access to the block size of a disk - as follows...

```
let DiskBlockSize (disk : string)
                  (targetMachine : string option) =
  let server =
    match targetMachine with
    | Some(value) -> value
    | None -> System.Environment.MachineName
  let wmiPath =
    sprintf """\\%s\root\cimv2""" server
  ( new ManagementClass(
        wmiPath,"Win32_Volume", new ObjectGetOptions())
  ).GetInstances()
   .OfType<ManagementBaseObject>()
   .Where
      ( fun mo ->
          not <|
...   mo.Item("Name").ToString().StartsWith(disk))
  |> Seq.choose
      ( fun mo ->
          try
            match
              System.UInt64
...   .TryParse(mo.Item("BlockSize").ToString())
              with
            | (true,value) -> value |> Some
            | _ -> None
          with _ -> None )
  |> Seq.tryHead
```

Video

303. WMI Queries - Parts I through III.

You should now recognise the method by which the WMI query is submitted; here we look for the options `Name` and `BlockSize` in the `Win32_Volume` class. Herein we use a Linq `Where` filter whereby we select the disk `Name` that corresponds to the function argument of `disk`. There may be a few of these depending upon your system configuration so we push the filtered output through a `Seq.choose` and for each disk (although the result will be the same) we determine if the option `BlockSize` is an integer and we'll just take the sequence head as an option. In code, I'll likely assign a default block size of 4KB if this function outputs `None`.

The Real Size of a File!

Bear in mind our comments in the previous topic about the size of a file in relation to the block size of the disk upon which it resides: For a sample file, a

UNC reference from my server, the properties for the file via the Windows File Explorer show two sizes - as depicted below. The disk this file is stored on is not compressed - if it were then there would be an even larger discrepancy between the **Size** and **Size on disk** values. If you do a `new FileInfo(u2.ModeSpecificPath)` and then a `FileInfo.Length` against the file the reported size is 64,275,720 bytes - which is reported in the Windows File Explorer as the **Size**.

The cluster size of the target disk is 4096 bytes. You'll note that the modulus (the remainder after division) of the `Length` from the `FileInfo` with the block size of `4096L` is `1288L` whereas the modulus, `64278528L % 4096L`, is `0L`. The **Size on disk** is an integral number of 4KB blocks and the Size is 2,808 bytes less than the number of blocks occupied by the file on the disk. You'll also note that the size difference of 2,808 bytes is less than the size of a block of 4,096 bytes. Therefore...

> The Windows File Explorer for a File displays two sizes...
>
> **Size on disk** Specifies the number of blocks occupied on the disk by the file. This will always be equal to or greater than the actual file size. If greater, it cannot be greater than the size of a single block on the target disk.
>
> **Size** Specifies the "real" size of the file on the disk. This is always less than, by up to no more than the length of a block size on the target disk, the reported **Size on disk**.
>
> The `FileInfo Length` property outputs the real size of the file - not the number of blocks that it occupies on the target disk.

OK - which one do we use? For me, if Im doing (especially asynchronous) buffered operations using the block size or a multiple thereof of the buffer, then it makes sense to use the **Size on disk** - the total number of blocks occupied by the file. If we load a memory stream thus, unless the real size is the same as the size on disk, then the last buffer will always have some trailing binary zeroes after the end of file since the memory stream is effectively a mirror of the disk storage - albeit the blocks in the memory stream are contiguous - which they may not be on the disk unless disk fragmentation is kept in check - especially for files that are written to often. You may save some memory (of a size less than the block size) if you define a memory stream to be the same length as the real file size but then you'll have to evaluate a buffer which, preferably, is an integral fraction of the real size; all in all, not worth the effort. Alternatively, if you were, for example, transferring a file over a network socket, say, you'd be more interested in the "real" file size.

Both have their uses but how do we evaluate the size on disk? To do so, we must once again turn to the native Windows API. We make an `extern` declaration for

the method `GetCompressedFileSizeW` of the DLL **kernel32.dll** - as follows (again, for such, don't forget the `open` for `System.Runtime.InteropServices`)...

```
[<DllImport("kernel32.dll",SetLastError = false)>]
extern uint32 GetCompressedFileSizeW(
  [<MarshalAs(UnmanagedType.LPWStr)>] string lpFileName,
  [<Out>] uint32& lpFileSizeHigh)
```

Let's firstly get the block size for our sample file - the value type named `u2` in our script...

```
let drive,server =
  let di = new DirectoryInfo(u2.ModeSpecificPath)
  di.Root.Name, u2.Server
```

We can now evaluate the block size...

```
let bsize = int64 (DiskBlockSize drive <|
...   Some(server)).Value
```

FSI reports this as `4096L` - 4KB. Let's now define a `mutable` (`byref`) value type used in invoking the `extern` `GetCompressedFileSizeW`...

```
let mutable hosize = 0u
```

That's a strange name to give a value type! There's a reason for this; regarding the method `GetCompressedFileSizeW`...

- It has an output of `uint32` - an unsigned integer. A `uint32` is 32 bits in size - 4 bytes.
- It has a `byref` input and output value type named `lpFileSizeHigh` of `uint32` - another 32 bit unsigned integer.

Yet, a file's size is reported as a long integer - an `int64` of 64 bits in length. I guess this is a hang-over from "earlier days" when an integer was sufficient to quantify a file size and this method retains retrograde compatibility (especially as it's in a file named **kernel32.dll** - that is, you'd suspect a 32 bit (x86) operating system rather than a 64 bit (x64) system). The method, then, effectively returns *two* 32 bit integers and requires *you* to "combine" them into a 64 bit integer! This involves some jiggery-pokery; the method output is the "low order" 32 bits of the combined 64 bit integer of the length. The `byref` argument, that which will be saved in our value type named `hosize`, are the high-order 32 bits of which, when combined with the low order bits will yield an `int64` for the file size/length.

You probably wont see this "in action" since a `uint32` can hold a file size of a length up to 2^{32} - 1 = 4,294,967,295 bytes. That is, one byte less than 2^{32} = 4GB. It's

only when your file size is, or exceeds 4GB, that the high order bits are set as well as the low order bits - and the low order bits and high order bits, separately are'nt then "real" unsigned integers - you have to glue them together to get a 64 bit integer to yield the "true" size. If the file size is less than 4GB then the high order `uint32` is zero.

Let's invoke the method...

```
let losize = GetCompressedFileSizeW(u2.ModeSpecificPath,
...   &hosize)
```

So; the high order bits are stored in `hosize` and the low order bits, as the output of the method invocation, are stored in `losize`. Let's now "glue" the 32 bits of each of the high and low order bits into a 64 bit integer...

```
let size = Int64.Parse(hosize.ToString()) <<< 32 |||
...   (int64 losize)
```

You see the `<<< 32`? I'm parsing the high-order unsigned, 32 bit integer into a 64 bit integer and since these *are* the "high-order bits", I'm left-shifting the bits of the integer 32 times. this basically places these 32 high-order bits in positions 0 through 31 of the 64 bit integer and leaves bits 32 to 63 as zero. Subsequently I use `|||` to "logically OR" these bits with the low-order bits parsed as a long integer. Parsing the low-order bits into a 64 bit integer means that said integer will have zeros in position 0 to 31 and only the bits from 32 through 63 will be "filled" with the low-order bits. Now, what the logical OR does, in binary terms of dealing with `0` and `1`, is that `0 ||| 0 -> 0` but *any other* combination of `0` and `1` (such as `0 ||| 1`) outputs `1` - you can "read" `0` and `1` as `false` and `true`, respectively. Consequently, OR'ing the two 64 bit integers will not interfere with any "numbers" (set bits) for either the high or low order bits and we have, at the end of it a "real" `int64` file length.

Is this size the "real" file size or the file on disk? FSI shows the output of the invocation as...

```
val size : int64 = 64275720L
```

Aha - the "real" file size! However, what if we want the "size on disk"? To evaluate this we have to use the block size and the formula is as follows...

```
let blocksoccupied = ((size + bsize - 1L) / bsize) * bsize
```

Video

304. A File's Size - Parts I through III.

If you evaluate this then FSI will show you the output of `64278528L` - which is precisely what we require.

When we implement this in the Core assembly, we'll use a switch via an optional Boolean argument so that one can choose to output either the real size or the size on disk - for which the latter will be the default.

Searching for Files

This is a topic we've already covered in dealing with the Type Providers in searching for a relatively specified Uri. I just want to tidy this up a bit especially with regard to searching the `AppDomain` for assemblies.

```
let codeBases (includeGAC : bool) =
  AppDomain.CurrentDomain.GetAssemblies()
           .Where(
             fun asm ->
              asm.GlobalAssemblyCache = includeGAC )
               .OfType<Assembly>()
  |> Seq.choose
      ( fun asm ->
          try
            let fi = new FileInfo(asm.Location)
            match
              fi.DirectoryName
                .StartsWith
                  (Environment.GetEnvironmentVariable
                    ("LOCALAPPDATA"))
              with
            | true -> None
            | false ->
                match
                  asm.GetCustomAttribute
                    (typeof<AssemblyCompanyAttribute>)
                  with
                | :? AssemblyCompanyAttribute as attr ->
                    match
                     attr.Company.StartsWith("Microsoft")
                      with
                    | true -> None
                    | false -> Some(fi.DirectoryName)
                | _ -> Some(fi.DirectoryName)
          with _ -> None)
```

I point this out since now there are three features that we have, before, either excluded or ignored...

1. Rather than excluding the Global Assembly Cache (GAC) from a search for loaded assemblies I now add a switch to the function invocation that enables the searching of GAC loaded assemblies.

2. When using FSI then, as intimated, the FSI assembly will dynamically create assemblies from your execution code. These dynamic assemblies

are saved to disk in the system **AppData\Local** folder of your user profile settings - the path is specified via the environment variable named LOCALAPPDATA. Any `#r` referenced assemblies are also copied to this "target" path. The path is not fixed in its entirety as FSI will will create and manage subdirectories dynamically as its needs dictate. On my machine LOCALAPPDATA resolves as **C:\Users\Chris.Shattock\AppData\Local**; enabling the showing of hidden files and protected operating system files in my Control Panel **Folder & Search options**, shows me, for example, an inclusion of the assembly **saTrilogy.Configuration.dll** in the subdirectory **assembly\dl3\J8MOL1CQ.GNR\HCGC4POZ.4M9\bdb342c6\5a90e15f_65ebd101** of the LOCALAPPDATA directory.

Unlike the IDE, FSI does not also copy files marked in the solution explorer that are copied into the **bin** path during a build of a project/solution process. In fact, even attempting to include a .NET **.config** file with you FSI referenced assembly is a nightmare. As no assembly "allied" files are copied by FSI into its dynamically generated folders, then I see no point in conducting a search for any assembly in the `AppDomain` where its path is in LOCALAPPDATA - sure, you may find the assembly if you've referenced it in FSI, but it won't enable you to pick the likes of a configuration file or referenced assemblies at its LOCALAPPDATA location since they won't exist therein! Consequently I'm filtering all such LOCALAPPDATA assemblies out of the `AppDomain` search.

The issue here is that having copied the referenced assemblies to the required LOCALAPPDATA path then their codebase *becomes* that LOCALAPPDATA path. It would be far more useful, to my mind and in promoting FSI functionality, for FSI to reference assemblies at execution time from their *originally referenced location* rather than their copied LOCALAPPDATA location - or at least allowed the user to query the origin of the FSI referenced assembly. It's possible such a wish would be rejected by the FSI developers as a security risk since then one is permitting access to entities outside of the FSI sandbox but, since FSI "knows" about its own sandboxes, it can easily intercept any such "outgoing" referencing request and "monitor" it. Just like an earlier "suggestion" that FSI should allow one to declare logical units of work to define scope boundaries for code execution, it's unlikely that either of the "suggestions" will be taken on-board in upgrading FSI so one will still have to work-around the issues generated by throwing FSI "into the mix".

3. Given the primary reason for wanting to search the domain loaded assemblies is to pick up an assembly codebase from where "associated" files and assemblies (like third-party WPF-related assemblies) exist, then it further seems improbable that one would ever be interesting in yielding any Microsoft assemblies in the search output. Consequently, any assembly that has its company attribute so set will also be filtered out of the search.

In the script I can now declare several file-search related functions...

```
let searchFile (root : string) (name : string) =
  let rec fileScan dir file =
    if DirectoryInfo.AccessPermitted dir then
      let target = Path.Combine(dir,file)
      match File.Exists target with
      | true -> [ target ]
      | false ->
          [ for sd in Directory.GetDirectories(dir) do
              yield! fileScan sd file ]
  else []

  match fileScan root name with
  | [] -> None
  | value -> Some(value.Head)

let SearchFilter (root : string) (filter : string) =
  let rec fileScan dir =
    seq {
      if DirectoryInfo.AccessPermitted dir then
      yield! Directory.EnumerateFiles(dir,filter)
      for sd in Directory.GetDirectories(dir) do
      yield! fileScan sd }

  fileScan root

let searchAppDomain (fileName : string) (includeGAC :
... bool) =
  codeBases includeGAC
  |> Seq.tryPick(fun cb -> searchFile cb fileName)
```

In the script, for `searchAppDomain`, I included a commented section where we don't filter-out the Microsoft assemblies since, in testing in FSI, `codeBases` is guaranteed to always yield an empty sequence since any referenced assemblies are placed in the `LOCALAPPDATA` path which is excluded by `codeBases`.

Video

305. File searching revisited - Parts I & II.

For the `searchFile` function you can see that it's simplified with regard to prior implementations as I'm not now using a mutable value type to "hold" the found output but rather doing a recursive yield of a list from which we try to take the head. It may now be "simpler" however the function is no longer tail-recursive so executing it "inappropriately" - such as from a root with a large number of nested subdirectories, can have a severe performance impact on your system and it may even thereby exhaust your system resources and cause a stack overflow exception.

From the principle of using a recursive yield for a list we can, for the hell of it, now also search for files using a filter - as in the function `searchFilter`. Once again, with regard to potential escalation of system resource usage, use this with caution with regard to how many nested directories may need to be searched for files whose name accommodates the argument specified filter - such as in, for example, ***.xml**.

A Miscellany of Utility Extensions

I want to just quickly run through some of the extensions I'll be putting in place to form general-purpose, utilitarian functions. These and more I'll incorporate into the core assembly in order to make them "centrally available" to any and all referencing assemblies and applications.

String Outputs

I generally refer to such a "stringifying" objects and use these extensions a lot in order to better express output in terms of both error and status reporting and for use in `ToString()` overrides so FSI and FsEye display of an entity is somewhat "more meaningful". Firstly, let's take the requirement to appropriately display the contents of an object array - for which the `ToString` output is useless and displays nothing of the "content" of such a value type. We previously considered an extension toHexString to "hexify" the display of a byte array (that I will also incorporate into the Core assembly) but, "more generally", consider the extension...

```
type 'a``[]`` with
  member μ.Stringify(?len : uint16) =
    let length =
      match
        match len with
        | None -> 1
        | Some(value) -> int value
      with
      | x when x > μ.Length -> μ.Length
      | x -> x
    ( μ |> Array.take length
      |> Array.fold
          ( fun acc elem ->
              elem.ToString() + "," + acc)
          String.Empty
    ).Trim(',') +
    if μ.Length > length
```

```
        then "..."
        else ""
```

There's nothing spectacular about this - it's just "useful". Now, in previous work, in validating "something" as a valid string we assumed a string or string option type and declared extensions to the `String` type. I tire of this; to me it's more obvious to have such as an extension of the `Object` type and thence, since everything derives from `object`, the Intellisense pop-up will show our extensions for *everything* - like the `ToString() object` method. Consequently, in the script I define an `Object` instance member extension called `IsValidString`.

Additionally, we have used a `CaseName` function to get the string representation of a union case name; let's now try an `Object` instance extension method, `CaseName`, as follows...

```
member μ.CaseName =
  match FSharpType.IsUnion <| μ.GetType() with
  | true ->
      match
        FSharpValue.GetUnionFields(μ, μ.GetType(), true)
        with
      | uci, o ->

...   String.Format("{0}({1})",uci.Name,o.Stringify())
  | false -> μ.ToString()
```

Again, "nothing special" about this other than its use as an `object` extension property. However, we also deal with `enum` types and, if we're doing this for union cases then we may as well use a single object property to evaluate the ToString() for either a union case name or an `enum` member - as follows...

```
member μ.ItemName =
  match μ with
  | :? Enum as e ->
    Enum.GetName(e.GetType(),e)
  | typ when FSharpType.IsUnion <| μ.GetType() ->
      match
        FSharpValue.GetUnionFields(μ, μ.GetType(), true)
        with
      | uci, o ->
          match o with
          | [||] -> uci.Name
          | _ ->

...   String.Format("{0}({1})",uci.Name,o.Stringify())
```

```
    | _ -> μ.ToString()
```

Now, when you invoke the `ItemName` property against an object instance it will show...

- For an `enum` member its name.
- For a union case the case name *and*, if the case has an underlying type, we'll attempt to `Stringify` that type and show its "content" in the output string. For example, for the union...

```
type Aunion =
  | First
  | Second of string
  | Third of int
```

Then FSI output for an `ItemName` against each shows, for example...

```
val it : string = "First"
val it : string = "Second(content)"
val it : string = "Third(9)"
```

- For anything else we just use the "normal" `object ToString()` method.

Video

306. Object Extensions

For error reporting in the past we'd often defined a Cases member for a union to show what cases are incorporated into a union - such as for our `FileAccessMode`. In line with the `ItemName` extension we can incorporate another object extension that will deal with union cases as well as `enum` members as follows...

```
member μ.Cases =
  match μ with
  | :? Enum as e ->
      ( Enum.GetNames(e.GetType())
        |> Seq.fold(fun acc em -> acc + ";" + em)
...   String.Empty
      ).TrimStart(';')
  | typ when FSharpType.IsUnion <| μ.GetType() ->
      ( FSharpType.GetUnionCases(μ.GetType())
        |> Seq.fold
            (fun acc uci -> acc + ";" + uci.Name)
            String.Empty
      ).TrimStart(';')
  | _ -> μ.ToString()
```

Converting a String to a Typed Array

Especially in dealing with the expressions required by the XSettings Type Provider for converting Xml strings to strongly-typed arrays I was frustrated by the apparent need to hard-code a a method invocation such as `DateTime.Parse` for each element in a collection to yield an output array of the appropriate type - even then, for the Type Provider, I had to "pre-qualify" the output type of such a provided property. I was pretty sure there was a better way to do this and, given the .NET interaction, I was pretty sure that it meant using generics and casting.

It took a while - until I "remembered" about Linq and its prevalence throughout both .NET and F#, to find a "simple" way to split a string into a typed array. The moral of this story is that, once again, you can be pretty sure that there's "something" in the .NET Framework that will address "most" of your application problems - that is, don't attempt to "re-invent the wheel" until you've had a thorough look through both the F# documentation and the .NET Framework.

Consider the code...

```
type String with
  member μ.ArrayOf1<'a>(?delimiter : char) =
    let splitter =
      match delimiter with
      | None -> Unchecked.defaultof<char>
      | Some(value) -> value
    [|  for txt in μ.Split(splitter) ->
          System.Convert.ChangeType(txt,typeof<'a>)
     |].Cast<'a>()
      .ToArray()
```

This, to me, demonstrates elegance much in the same way as we found in the recursive union we developed for our WPF Graph. It shows how well Linq is integrated into F# and .NET: There is a generic type argument `'a` that will specify the ultimate type of the output array and a single argument of what character delimiter is to be used in undertaking a `Split` for the string instance. But wait! I'm allowing the delimiter as an optional argument and, if not specified, it will be a `null char`! That's fine - since the Split still works with a null char delimiter - it just doesn't *do* a Split but we still get a typed array but only with one element. There is method in this madness - we use this "feature" in the XResources Dynamic Provider so it can handle singleton settings values as well as array settings values.

Anyway, we simply formulate an array comprehension such that each item will be the element output of the `Split` to a string array and, for each item, we invoke the `Convert` class method of `ChangeType` in the `System` namespace. For `ChangeType`, we feed in the split element item as a string and the type we want it converted to via the generic type argument.

Being .NET, that yields an object array `obj[]`, so we then use the Linq `Cast` method to strongly-type it as a collection of type `'a`. `Cast` output is actually then an IEnumerable so we use the Linq `ToArray` method to convert the IEnumerable into a `'a[]` - which is precisely what we want.

In the script there's a number of tests of the `ArrayOf1` string extension where the value types are taken from our XSettings file. However, you'll note the absence in our script of declarations for the `IsoDate` and `IsoTime` types so I've added a `#load "..\\Types.fs"` to include them. For `IsoDate` and `IsoTime` the `ArrayOf1` fails with an invalid cast exception - fair enough, these are, after all, user-defined types so I haven't taken care to define them as exhaustively as the .NET types. However, `IsoDate` and `IsoTime` both have static `Parse` methods that take a string and output a class instance. I want to extend the extension so that if such conversion fails I'll look for a static Parse method for the type then invoke it with the argument of the split item string. Recall, we did similar in dealing with WPF where we wished to be able, for third party assemblies, to emulate the invocation of a default class constructor and static method invocation from F# as we'd effectively removed such in the C# initialisation of the WPF object by removing the "parent" WPF item binding to a C# "code-behind" class - our "Slimline WPF" methodology.

To effect such functionality is a more holistic way I want to create two `Assembly` static member extensions as follows...

```
type Assembly with
  static member DefaultInstance<'a>() =
    try
      match typeof<'a>.GetConstructor(Type.EmptyTypes)
...   with
      | null -> None
      | _ ->
          Some(typeof<'a>.Assembly
                         .CreateInstance(
                            typeof<'a>.FullName,
                            true,
                            BindingFlags.CreateInstance,
                            null,
                            null,
                            Threading.Thread
                                     .CurrentThread
                                     .CurrentCulture,
                            null))
    with _ -> None

  static member StaticMethodInvoke<'a>(methodName :
...   string,
```

```
                                                ?parms : obj[]) =
    try
      let args =
        match parms with
        | None -> null
        | Some(value) -> value
      match typeof<'a>.GetMethod(methodName) with
      | null -> None
      | methodInfo ->
          if methodInfo.IsStatic
            then Some(methodInfo.Invoke(null, args))
            else None
    with _ -> None
```

The `GetConstructor` is much as it was before - the difference being that we specify the type required via the generic type argument. For the `StaticMethodInvoke` we again specify the type required via a generic type argument but we also permit an optional argument, `args`, an `obj[]`, so that if the target static method requires arguments we can give these to it. Here the subsequent essential difference in the code is that we evaluate the optional argument and if not specified we set it to `null` - otherwise the `obj[]` used in the `methodInfo.Invoke` method call.

Now, with the ability of invoking `IsoDate.Parse(string)`, for example, let's review the conversion of a string to a typed array...

```
type String with
  member μ.ArrayOf<'a>(?delimiter : char) =
    let splitter =
      match delimiter with
      | None -> Unchecked.defaultof<char>
      | Some(value) -> value
    [| for txt in μ.Split(splitter) ->
        try
          System.Convert.ChangeType(txt,typeof<'a>)
        with _ ->
          match
            Assembly.StaticMethodInvoke<'a>(
              "Parse",[|box txt|])
          with
          | None -> null
          | Some(value) -> value
    |].Cast<'a>()
      .ToArray()
```

Video

307. Convert a String to a Typed Array - Parts I & II.

Now, should the cast raise an exception we'll trap it and instead try to invoke a static `Parse` method for the target type with an argument of an object array comprising the split element's text. You should now find this will work with any user-defined type, such as `IsoDate` and `IsoTime` where that type defines a static `Parse` method with arguments from which it may construct and output an instance of itself.

Mutation of a Result Output Type

This is not about code that appears in this script but it is about important changes that I've made in our Result structure commonly used to wrap function output and it deserves its own topic if for no other reason than such a change is easily found in this material.

The change arises primarily from the fact of the potential usage of our `ArrayOf<'a>` String type extension. A consequence of using this with a `null` delimiter is that output is still converted to an array - albeit with only the one element. This may give rise to difficulties if the output is to be Result wrapped - one may wish the output to be a `Result<string>` but via `ArrayOf` it would become `Result<string[]>`. Consequently I have decided to encode an additional member in the Result structure to "mutate" the output option value. In order to support any exceptions raised in this mutation process I've now made the `Errors` field mutable. We then declare the member...

```
/// Given the current result instance use a function/lambda
/// expression to mutate the Result&lt;'a&gt; instance into
/// a Result&lt;'b&gt; instance.
member μ.Mutate<'b>(f : ('a -> 'b)) =
  try
    match μ.Success with
    | false ->
        Result(μ.Errors)
    | true ->
        Result(f μ.Output.Value |> Some )
  with exc -> Result(μ.Errors.AppendException exc)
```

`Mutate` is a generically typed function so it's output is "strongly" expressed when used in user code. It takes a single argument and that is a function that takes an argument of type `'a` - which is the generic type with which the Result<'a> is declared - and outputs the type `Result<'b>` - using generic type argument of the `Mutate` member.

The member constructs a new `Result<'b>` instance by applying the `Output.V`

alue of the Result<'a> instance to the function argument. If the Result<'a> Output has no value, then we simply output a Result<'b> using the Result<'a> Errors field - which is "expected" to contain exceptions from a prior function evaluation that produces the Result<'a> instance. Any exceptions encountered along the way are appended to any existing exceptions recorded in the Errors instance of Result<'a> and the output is then a Result<'b> containing these accumulated exceptions.

We will "see this in action" for our XSettings Dynamic Provider but, to give you a taste of what's to come, here's some of it's members - first one that outputs an 'a[], then one that uses Mutate to change a 'a[] to a 'b via a function and then one that uses Mutate with a lambda expression (the ArrayOf<'a> is invoked in the xidValue member) ...

```
/// Dynamically evaluate the XId value from the underlying
/// XSettings file and output it as a type of float[].
member μ.FloatArray (xid : string) =
  μ.xidValue<float>(NodeType.FloatArray,xid)

/// Dynamically evaluate the XId value from the underlying
/// XSettings file and output it as a type of float.
member μ.Float (xid : string) =
  μ.xidValue<float>(NodeType.Float,xid).Mutate<float>((Array.head))

/// Dynamically evaluate the XId value from the underlying
/// XSettings file and output it as a type of byte[].
member μ.ByteArray (xid : string) =
  μ.xidValue<string>(NodeType.ByteArray,xid)
   .Mutate<byte[]>(fun arr -> Convert.FromBase64String(arr.[0]))
```

File System Watcher, Observables & Observers

To implement a Dynamic Provider as opposed to a Type Provider, it helps if the code can monitor any changes to the source data and react accordingly. I won't incorporate this "feature" into the Manifest Dynamic Provider - although I should, since it's conceivable that a primary and even satellite assemblies might change during the life-cycle of an active Dynamic Provider such as one one running via a Windows Service. Our concern is changes in either an XSet-

tings or XResources file for our Dynamic Providers but the principle may be extended for dealing with changes in the Manifest Dynamic Provider targeted assemblies.

We're being pretty specific here since we'll deal with a .NET `FileSystemWatcher` class instance wherein events are triggered under certain conditions - the event we're interested in is the `Changed` event of a file system watcher target whereby the underlying file has been modified - reflected by, if not a change in its size, then a changed in the file attribute regarding the time-stamp of when the file was last "written to". We could, if we so chose, simply "tag onto" the .NET `Changed` event but, for "general-purpose usage" your data source for a Dynamic Provider may not be "a file" - it could, for example, be a database and such would not raise a .NET event to signify a change in its content - and, even then, "which" content. You would have to manually code a .NET Event to monitor your source and raise the event appropriately and this is sometimes not possible and is always a royal pain where it is possible to "integrate" your data source "into the .NET environment" to register events.

This then, is why we use the "simpler" methodology of Observables and Observers and, fortunately, F# Observables are tightly integrated into the .NET Framework so it is trivial to identify a .NET event as the "subject" of an Observable. It is a relatively straightforward task to code an F# Observable even for non-.NET aware data sources as opposed to a .NET "event". For example, with SQL Server, we could fairly easily code an Observable to occasionally fire a stored procedure or SQL function to detect changes we're interested in - it would, effectively, just be a delegate function or a lambda expression that submits the stored procedure invocation and acts upon its output or return code. In fact, such could even emulate or use a Mailbox processor for asynchronous usage. You should note that it's slightly disingenuous of me to use SQL Server as "an example" since there *is*, in fact, very tight coupling between SQL Server and .NET as it's possible to incorporate .NET assemblies into the SQL Server core and use them *within* the SQL Server runtime context. That's fine - if *you* "own" the SQL Server instance in question - but try doing it though a shared hosting provider or even an Azure SQL instance and you won't get very far - unless you're *very* rich! If "anybody's listening" - wouldn't it be nice if one could use F# "natively" in SQL Server in the same way as T-SQL - even to the extent of coding and saving functions and stored procedures written in F#! Couple that with Logical Units of Work in FSI and a "native" F# interlink between external assemblies and SQL Server. Enough dreaming - let's now consider the creation of an Observable for the file system watcher's `Changed` event - from the **Messing Around.fsx** script...

```
type FileChangeWatcher() =
  let lastWritten =
    new ConcurrentDictionary<string,DateTime>()
  let invoker =
    new ConcurrentDictionary<string,Invoker>()
```

```
  static let singleton = lazy (new FileChangeWatcher())
  static member Singleton =
    if singleton.IsValueCreated
      then singleton.Value
      else singleton.Force()

  member val LastWritten = lastWritten with get
  member val Invoker = invoker with get

  member μ.Initialise(invoker : Invoker,path,name) =
    let target = Path.Combine(path,name)
    if not <| FileChangeWatcher.Singleton
                                .Invoker
                                .ContainsKey(target)
      then FileChangeWatcher.Singleton
                            .Invoker
                            .AddOrUpdate
                              ( target,
                                invoker,
                                fun _ _ -> invoker)
                              |> ignore
    let fsw =
      new FileSystemWatcher(Path = path, Filter = name,
                            EnableRaisingEvents = true)

    fsw.NotifyFilter <-
      fsw.NotifyFilter ||| NotifyFilters.LastWrite

    fsw.Changed
    |> Observable.map
       ( fun eventArgs ->
           let fi = new FileInfo(eventArgs.FullPath)
           if  fi.LastWriteTime <>
               FileChangeWatcher.Singleton
                                .LastWritten
                                .GetOrAdd
                                  (fi.FullName,
                                DateTime.MinValue) then
             FileChangeWatcher.Singleton
                              .LastWritten
                              .AddOrUpdate
                                ( fi.FullName,
                                  fi.LastWriteTime,
```

```
                                        fun _ _ ->
                                          fi.LastWriteTime)
                                    |> ignore
                        FileChangeWatcher.Singleton
                                        .Invoker.[fi.FullName],
                        fi.FullName )
```

Firstly, you'll note in the script that I've added some `#load` statements to pick up some "core" types in the core assembly that support this types functionality. Now, I define a type - a class and you'll see that I'm once again using the Singleton pattern (as for our ETW Event Source) for this class. I'm doing this since I want to maintain some state information regarding the Observable we define. This state information comprises two parts expressed via concurrent dictionaries...

1. The details of who triggered the Observable - the concurrent dictionary named `invoker`. This isn't really necessary for anything other than auditing purposes. Once you've declared an Observable then anyone who has access to the value type whereby its declared can "subscribe" to that Observable. The dictionary key is the fully-qualified path and name of the file against which we will be declaring an Observable and its value is an instance of the Invoker type. The Invoker type is an extension to the `Origin` type of our ETW functionality - we've just accumulated the origin information with the invoker (value type) name into a structure. This is defined in the **Origin.fs** source file of the `Core` project.
2. More importantly, the .NET `FileSystemWatcher` has a habit of firing changed events twice so in order to try and ameliorate this overhead (which I fail to do) I'll maintain a concurrent dictionary named `lastWritten` that saves the target file's attribute of when it was last written to - the `DateTime` thereof.

These 'backing fields' are exposed, via the Singleton, as public, read-only properties. This leaves us with the `Initialise` method whereby we configure the file system watcher and create an Observable for its `Changed` event. This takes three arguments...

1. The invoker of the method. I "cheat" slightly in this because, for our Dynamic Providers exposed as classes, I'll just inherit the `Invoker` structure with a constructor argument of `Mark` - the same function we used for ETW as in `let Mark _ = ()`. This will then show the invoker as `ctor` - the constructor of the class. Remember, this is "just" used for auditing purposes - the next arguments are those that are required for creating a `FileSystemWatcher` instance from the `System.IO` namespace...
2. The full path (less the file name) that contains the file(s) against which to instantiate a file system watcher and...
3. The file name against which we wish to instantiate a file system watcher.

We then `Path.Combine` the path and name to yield the fully qualified file name; note that we are effectively thus excluding the possibility of instantiating the subsequent file system watcher against more than one file - but, we should actually not only check this but also ensure that no exceptions occur in referencing said target file. Generally, one may use a file system watcher with a number of files specified via a filter - such as, for example, ***.xml**. I then want to see if our invoker dictionary contains a reference to the invoker specified in the argument for the target file. If the target is already in the dictionary then I don't want to change the information about who the *original* invoker was - if it isn't, then I'll use the concurrent dictionary `AddOrUpdate` to stick this information in the dictionary.

We then declare a `new FileSystemWatcher` instance using the argument specified path and name whilst also telling the instance that we *do* want to enable it to raise events; in the subsequent expression we use the logical OR to ensure that the triggers registered by the file system watcher instance include the `LastWrite` attribute for the target file. With this, any change in the target file's last written `DateTime` will give rise to a `Changed` event - which we subsequently add an event handler to.

Now, the event handler for the changed event is actually piped into an instance of an F# Observable wherein we encode a lambda expression that is the sequence of actions to undertake in the event that a `Changed` event has been triggered - this is the `Observable.map` function whereby we actually "define" an observable's actions.

The Observable actually takes the same event arguments object for its lambda expressions anonymous function that the .NET `Changed` event handler would have to specify (as its second argument - the first being the "sender" of the event). The Changed event arguments encapsulate, as a property, the `FullPath` of the target file against which the event was raised - we pull this out and feed it through a `FileInfo` to extract more useful information about the argument as a file. With the `FileInfo` instance we check whether the file attribute for the last written `DateTime` is different from that in our Singleton accessed concurrent dictionary of `LastWritten` - if not, we effectively discard the event -there is an implicit `else ()` in the code. Otherwise, we'll update the last written `DateTime` in the concurrent dictionary and then spit out a tuple of the original invoker of the Observable and the target fully qualified file name.

So, we have defined an Observable - let's bring one into existence by using the expression...

```
let fsw =
  FileChangeWatcher.Singleton
                   .Initialise(new Invoker(Mark),
                               @"T:\","temp.txt")
```

Somewhere "out there" is now an observable "waiting" for a file change on the

file **T:\Temp.txt** at which point its lambda expression (or a delegate) will be invoked. You can go ahead and create and change such a file - but you won't "see" anything happening - even if you attempt to debug the `fsw` expression in FSI rather than execute it. The *Observable* is declared and active, but we have yet to define any *Observers* by which we can benefit from the lambda expression of the Observable. Let's now "subscribe" a value type to this Observable; use the following expression...

```
fsw.Subscribe(fun (o,f) -> printfn "File = %s, Owner =
...   %A" f o)
```

Video

308. Observables and Observers - Parts I through III.

You see; with an "observer", we invoke the `Subscribe` method against the value type and use a lambda expression (or delegate) that responds to the output of the Observable. Now, you can only use `Subscribe` against a value type whose signature is `IObservable<'a>`. The type of `fsw` is `IObservable<Invoker * string>` - since we declared it explicitly against the output of the Initialise member whose Observable output comprises a tuple of an Invoker instance and a string instance - being the fully qualified name of the "target" file against which the Observable is defined. Our `fsw.Subscribe` anonymous function of the lambda expression's anonymous function will then receive, as its input argument, this tuple that we've encoded as `(o,f)` with type `<Invoker * string>`. Our function just pushes out a text string detailing the event but, as you can appreciate, when it comes to coding the Dynamic Providers, we'll have to do a little more than that!

Computation Expressions/Workflows - Overview

There's no denying that this is a complex subject and, moreover, there's nothing in this material to date that we've done that really "justifies" the use of a workflow. Consequently, in this topic, I'll just introduce the concept and its asynchronous ramifications in comparison to "what we already know". As the first part, then, of this overview, let's go way, way back to when we were investigating the "binding" of a function in such a way as to be able to trap any exception it incurred within the scope of a class within which function evaluation is encapsulated. Consider (from the **More Messing Around.fsx** script in the `Core` project)...

```
type Compose() =
  static member Bind f arg =
    try f arg
    with exc -> raise exc
```

This shouldn't raise any eyebrows - by now we're well used to dealing with function composition using such a methodology. Consider the function...

```
let f x y =
  printfn "arg1 = %i, arg2 = %i" x y
  x + y
```

It's simple enough - the function takes two arguments, `x` and `y`, prints out what they are and then outputs their sum. Let's "compose" iterative invocations of this function...

```
let Composition =
  Compose.Bind (f 42)
  >> Compose.Bind (f 43)
```

So, we've now declared a composite function that firstly partially evaluates `f 42`, then applies the `Composition` function argument - presumed to be another integer. In the following composition, the prior output is then used as the secondary input after the partial evaluation of `f 43`. You can test this composite function by evaluating Composition 0 and FSI will show you the output of...

```
arg1 = 42, arg2 = 0
arg1 = 43, arg2 = 42
val it : int = 85
```

This much, given experience to date, you should have no problem with. Now, I'm going to refer to Scot Wlaschin's website, F# for Fun & Profit - his page "Computation expressions: Introduction"[64]. I've slightly modified Scott's introductory Computation Expression and declared it in my script as...

```
type LoggingBuilder() =
  let log p = printfn "expression is %A" p

  member this.Bind(x, f) =
    log x
    f x

  member this.Return(x) =
    x
```

This may "look similar" to our `Compose Bind` - except that;

- Scott uses an instance `Bind` member - ours is static.
- Scott's `Bind` invokes a side-effect of printing out what it's evaluating.
- Both appear to permit an "arbitrary" function as an argument.
- Scott's `Bind` arguments are "reversed" - argument first, then function.
- Scott's `Bind` doesn't use a `try/with` exception trap.

- the `LoggingBuilder` contains a "mysterious" member, `Return` that doesn't actually appear to do anything other than output its argument.

Let's create an instance of the `LoggingBuilder` class...

```
let logger = new LoggingBuilder()
```

Now, Scott's workflow that encapsulates a series of Computation Expressions against this `logger` instance...

```
let loggedWorkflow =
  logger
  {
    let! x = 42
    let! y = 43
    let! z = x + y
    return z
  }
```

Now, this *is* different! It looks like we're declaring a sequence with the `{ }` expression but remember that `{}` is also used to encapsulate things like Query Expressions as well as sequences. The other main difference is it's not a "keyword" appearing before `{}` - it's an instance of a class. Be that as it may, this can be regarded as a sequence - a sequence of both synchronous and asynchronous computation expressions that, combined, express a "complete workflow". You'll recognise the `let!` as in our Mailbox processor usage - it means, the way I put it, "loiter with intent" - the intent being an assignment. Does it mean the same with regard to a computation expression? Assuming that it's an asynchronous assignment just that it so happens that the value type `x` is evaluated via a constant, `42`, rather than some "other", possibly more complex expression - but it *could* use such in the evaluation of `x` - the same for the value type `y`.

What about the value type `z`? This too is purportedly evaluated asynchronously given our current understanding of `let!` - and, by *that* understanding of the evaluation of `z` as `x + y`, then `z` **cannot be evaluated** until `after` **both** `x` and `y` have been asynchronously assigned. Only then, is the workflow capable of evaluating the value of `z`.

Ultimately the workflow uses the "function" `return` to, clearly, output the asynchronously evaluated `z` as the consequence of the workflow. Indeed the output type of `loggedWorkflow` is `int`. Up to the point of the `return` invocation this is all relatively "clear" given our current understanding - even if we were to replace `42` and `43` by other, more complex expressions and if, for example, the value type `x` takes longer to evaluate asynchronously than `y`, then we still can't evaluate `z` *until* the evaluation of `x` is complete.

If you execute this workflow - a.k.a. sequence of computation expressions - in FSI then you'll get the output...

```
expression is 42
expression is 43
expression is 85

val loggedWorkflow : int = 85
```

Now, I don't know about you, bit I don't like this - in fact I find it extremely off-putting. I can "see" how the function `return` in the workflow may end up invoking the `Return` member of the `LoggingBuilder` instance named `logger` used in the "encapsulation" of the workflow and its apparently synchronous as opposed to asynchronous. However, where the hell does the `Bind` get involved? We *know* it *is* involved, since we can see the consequences of its side-effects in the FSI output as messages detailing what it's working against. If one "manually" invoked the `Bind` with the argument `x = 42`, it would fall over in a heap. `Bind` expects two arguments so "something", somewhere along the line is "converting" `let! x = 42` into an asynchronous unit of work that, "under the covers", invokes `logger.Bind`. We can't invoke it as `Bind 42 x` because that makes no sense, since the `Bind` output would then be the evaluation of the expression `x 42` - which is meaningless as `x` is clearly not typed as a function! We can't make the function argument `let` since that's an F# keyword not a function and, even then, the argument would have to be `x=42` and that, too, would be meaningless as anything other than a string.

Video

309. Workflows and Computation Expressions - Parts I through IV.

Now, digging in the appalling mess that is the MSDN "documentation" about Computation Expressions[34] (or, largely, elsewhere) we find that the keywords `let!` and `return`, with regard to Computation Expressions are referred to as "syntactic sugar" - even more, that `let!`, with regard to a computation expression, bears no similarity to our understanding of its usage in, say, a Mailbox processor. Therefore, trying to analyse a workflow by using what you "already know" about asynchronous evaluations is a fruitless and pointless task! From the documentation a computation expression `let!` is "syntactic sugar" for `builder.Bind(expr, (fun pattern -> {| cexpr |}))` where, here "builder" refers to the `logger` instance and I don't have the inclination, right now, to untangle this bowl of spaghetti for a "trivial requirement".

For a workflow using computation expressions you are **required** to declare a "builder class" such as `LoggingBuilder`, and the members of that class must correspond to the "syntactic sugar" required methods for each type of expression you may use in your workflow - `let!` and `do!` map to a `Bind` member, `yield` maps to a `Yield` member, `yield!` maps to a `YieldFrom` member and so on. There are a total of 14 of these members required to encompass full computation expression "syntactic sugar" functionality in a workflow. In Scott's example, we only require `Bind` to act upon the "syntactic sugar" in our workflow of `let!` and `Return` against the workflow "syntactic sugar" or `return`.

You have to ask yourself, depending upon how generically one can declare such a "builder class", do I have a functional requirement that closely mirrors a workflow and is complex enough in its "branches" to justify such a "builder class" definition and, even then, can I "re-use" such a class for "similar" workflows? You'd think that the answer to this question is - yes, I can create a single, sufficiently "generic" class that will encapsulate a wide variety of workflows. However, if this were so, then why would there be a need to create such a "builder class" in the first instance? It would be a core part of F# functionality - would it not? A "basic builder class" in the F# core that one can inherit from or use directly to compose workflows. One reason appears to be that in forcing one to explicitly create a "builder class", one then has the opportunity of expanding upon the functionality of the workflow - such as in Scott's side-effect of logging and, bear in mind, this is a "trivial" workflow requirement and such functionality may better be incorporating using "other" mechanisms. That latter statement is the other reason; generally, you may well find that a trivial or simplistic workflow is better undertaken using a different strategy for the evaluation of its outcome.

The bottom line is that, ultimately, your "builder class" and the methods it exposes to handle the "syntactic sugar" used in your workflow is going to be fairly specific to a single range of similar requirements. In Scott's series he ends up with a `TraceBuilder()` "builder class" that exposes full "syntactic sugar" functionality - as in...

```
type TraceBuilder() =
  member this.Bind(m, f) =
    Option.bind f m
  member this.Return(x) = Some x
  member this.ReturnFrom(x) = x
  member this.Yield(x) = Some x
  member this.YieldFrom(x) = x
  member this.Zero() = this.Return ()
  member this.Delay(f) = f
  member this.Run(f) = f()
  member this.While(guard, body) =
    if not (guard())
      then this.Zero()
      else this.Bind( body(), fun () ->
    this.While(guard, body))
  member this.TryWith(body, handler) =
    try this.ReturnFrom(body())
    with e -> handler e
  member this.TryFinally(body, compensation) =
    try this.ReturnFrom(body())
    finally compensation()
```