Module18: Dyalog.Net Classes

§ 18.1 Writing Dyalog.Net Classes

§§ 18.1.1 Dyalog Namespaces and .NET Namespaces

.NET namespaces are similar to Dyalog namespaces, but unfortunately (for .NET) are not identical. Nevertheless, when creating a .NET class in Dyalog APL, a Dyalog namespace is destined to become a .NET namespace (in a .NET assembly) containing the new .NET class.

18.1.1.1 Create a workspace called GENERAL.DWS containing a single namespace called `#.Maths`.

§§ 18.1.2 Creating a `NetType` Object

In Dyalog.Net, a .NET class is created through a `NetType` object. The `BaseClass` property of a `NetType` object *may* be set to the name of some particular class from which the new class is derived.

```
      CVec ⎕WC'NetType'                    ⍝ Create a new NetType object, with name in CVec
```

18.1.2.1 In the namespace `#.Maths` create a Dyalog GUI `NetType` object called `Spectrum`.
Hint: See GUI Help or *Object Reference* for description of `NetType` object.

§§ 18.1.3 Writing Functions and defining Variables

18.1.3.1 In namespace `#.Maths.Spectrum`, copy in function `∇Fourier∇` from distributed workspace MATH.DWS and write the following two functions:

```
      ∇ R←ft W ⍝ Fourier Transform
[1]     R←Fourier W
      ∇
```

```
      ∇ R←ift W ⍝ Inverse Fourier Transform
[1]     R←¯1 Fourier W
      ∇
```

§ 18.2 Exporting Methods and Properties

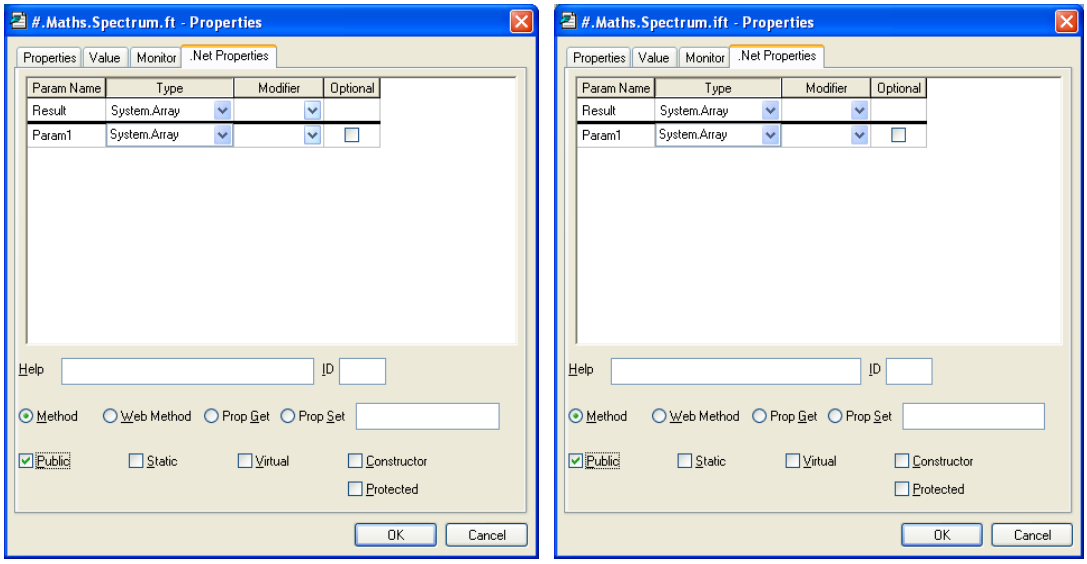
§§ 18.2.1 Arguments and Result *dataTypes*

When exporting methods and properties it is necessary to specify the dataTypes of all interface variables. In order for .NET to interpret correctly the dataTypes being specified it is necessary for the classes associated with the specified dataTypes to be accessible in APL. The basic dataTypes are to be found in the core `System` namespace. They correspond to classes such as `System.Int32`, `System.Int64` and `System.Array`. These classes inherit from `System.ValueType` which itself inherits from `System.Object`.

To make these basic dataTypes visible to APL when exporting class members it is necessary to assign `⎕USING` to the core `System` namespace. Were we to do this then the classes would have to be called Int32, Int64 etc… However, the default names in the .NET properties dialogue are written with the "System." prefix. Therefore assign `⎕USING` to an empty character vector in order to satisfy the default entries.

We can now specify the calling dataTypes of `∇ft∇` and `∇ift∇` as the default, `System.Array` and identify them as public methods. (Remember, methods may *not* be dyadic functions.)

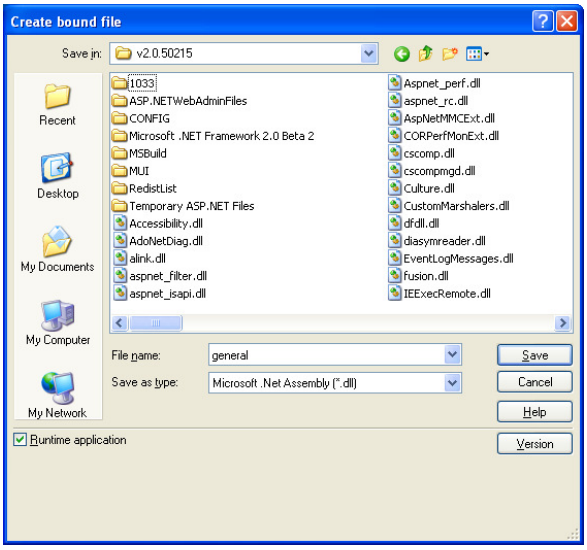
18.2.1.1 Assign `⎕USING←''`, place the cursor on `ft` in the Session and right-click the mouse. Select [Properties][.Net Properties] and set the information as below. Repeat for `ift`.



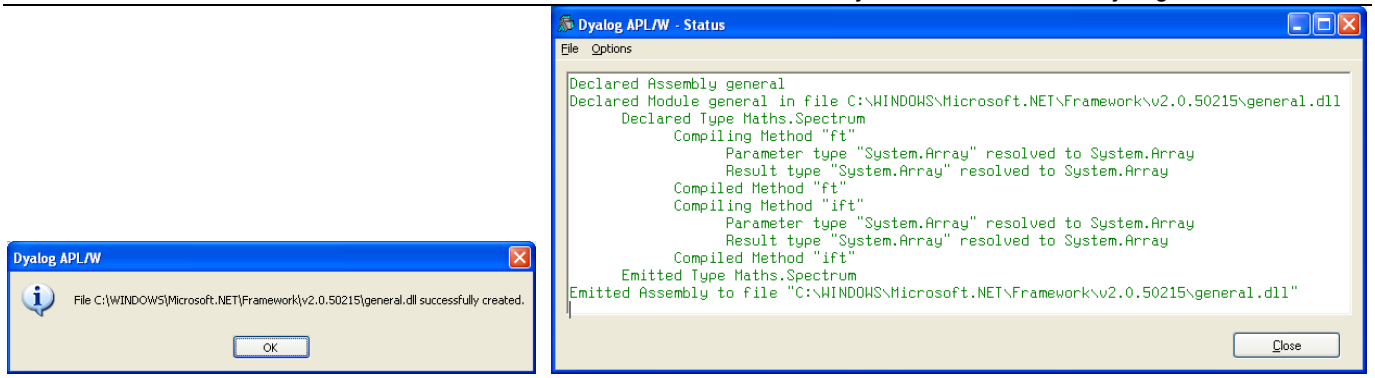
Alternatively, it should be possible to assign this information via `SetMethodInfo`. Also note that we could have identified the methods as static in which case they would be useable directly from the class without the need to create an instance of `Spectrum` before calling them.

§§ 18.2.2 Making an Assembly

18.2.2.1 Select [File][Export] from the Session menu and navigate to the framework directory. Choose a file name – the default is the name of the workspace with .dll rather than .dws. This will be the name of your .NET assembly. The Runtime application check box should be checked in order to create a distributable assembly, otherwise the development .dll (*eg* dyalog110.dll rather than dyalog110rt.dll) will be bound.

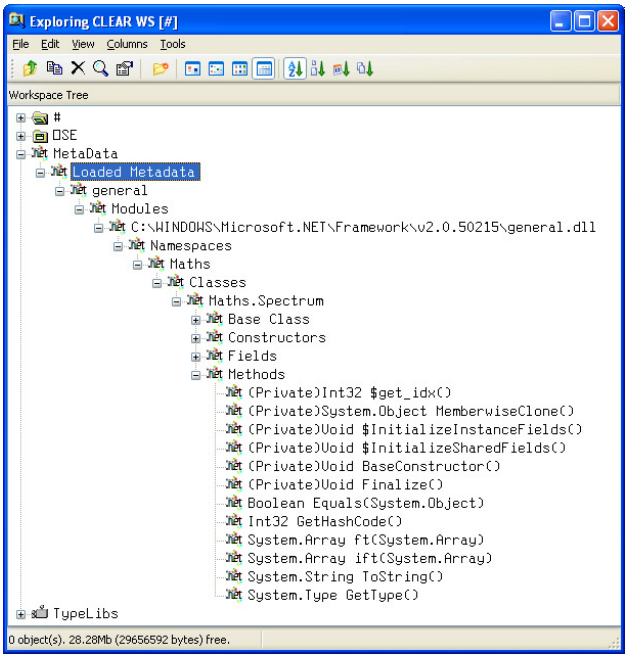


When you hit the Save button, if no problems are identified, then the following message box and status message dialogue appear.



§§ 18.2.3 Checking the MetaData

18.2.3.1 In WS Explorer, load the MetaData of the assembly `General.dll`. Find the exported methods and check their entries.



Note that a number of other methods are present in the list. They have been inherited from `System.Object` ¢ which is the default `BaseClass` of a `NetType` object. It is at the root of every .NET class and therefore every class has the `ToString` and `GetType` methods (although their constructor syntax may be overridden by any of their descendants).

§ 18.3 Calling Dyalog.Net Classes

§§ 18.3.1 Calling your Dyalog.Net Class from Dyalog APL

To use the .NET class which you have just created, you follow exactly the same procedure as for any other .NET class. First you have to set `⎕USING` to the appropriate value:

```
      ⎕USING←'Maths,General.dll'
```

By way of checking that the class is visible to APL, typing its name at this point will print the full namespace-qualified name of the class in the Session, surrounded by parentheses to indicate a class name.

```
      Spectrum
(Maths.Spectrum)
```

An instance of the class is created using the `⎕NEW` system function:

```
      S←⎕NEW Spectrum
```

Create an arbitrary rank array of data whose Fourier Transform is to be found.

```
      ⎕←R←2 2 10⍴400?400
180  95  12 213 350 139 187 273  82 243
150 331 177  34 341  69 371 345 332  28

391 126 301  61 336 319  31 325 254  36
233 263  43 148 246 153 274 333 305 306
```

Applying the transform followed by its inverse reproduces the original data, now with an explicit zero imaginary component.

```
      S.ift⊂S.ft⊂R
180 0  95 0   12 0   213 0  350 0  139 0  187 0  273 0  82 0   243 0
150 0  331 0  177 0  34 0   341 0  69 0   371 0  345 0  332 0  28 0

391 0  126 0  301 0  61 0   336 0  319 0  31 0   325 0  254 0  36 0
233 0  263 0  43 0   148 0  246 0  153 0  274 0  333 0  305 0  306 0
```

(From Dyalog version 10 onwards it is even possible to display ActiveX controls and .NET classes in the cells of a `Grid` object.)

§§ 18.3.2 Calling your Dyalog.Net Class from C# and VB.NET

.NET classes created using Dyalog APL may be called from any .NET language just like any other .NET class. The assembly .dll file, in this case general.dll, and any supporting .dll files, in this case fftw.dll, have to be shipped. The only other Dyalog files that have to be shipped to the machine intending to use the class are bridge110.dll, dyalognet.dll and dyalog110rt.dll, the Dyalog APL runtime engine.

§§ 18.3.3 Complications

We have attempted to give a straight-forward view of .NET from an APL point of view. Many of the 'new' concepts have been part of Dyalog APL since its inception – data encapsulation, method overloading, exception handling … - and many have already been incorporated in a natural way into Dyalog APL – namespaces, threads, sockets …

The reality of .NET is not as clear-cut as an APL programmer would wish. There are inevitably a number of complications. We have neglected at least those we can discern in the belief that the straightforward picture is the best foundation on which to build.

However, we note here a couple of the grey areas that you will encounter on deeper investigation of .NET.

- .NET namespaces may be spread over more than one assembly. Excuse me! Why?
- Some **C** functions do not return their results, but rather they store them in memory at specific locations indicated by pointers, which they *do* return. In Dyalog.Net this sad **C** eventuality is resolved using `ByRef` ¢ in `Dyalog` ñ supplied in `bridge110.dll` à.
- Classes are not the only members of assemblies. Assemblies also contain Enumerations, Interfaces and Structs. We have ignored these topics because they do not appear to introduce anything particularly exciting, but just complicate an already complicated picture.

18.3.3.1 See *Dylog.Net Interface Guide* chapters 1 to 4 and ..samples\APLClasses\… for further study. Please ask for the next module ☺.