Dyalog for Microsoft Windows .NET Framework Interface Guide

Dyalog version 20.0

DYALOG

Dyalog for Microsoft Windows .NET Framework Interface Guide

Dyalog version: 20.0
Document Revision: 2025-10-30 main:e0843eae32

Unless stated otherwise, all examples in this document assume that `⎕IO ⎕ML ← 1`

email: support@dyalog.com
https://www.dyalog.com

Contents

1 Overview

1.1 Introduction

This manual describes the Dyalog interface to the Microsoft .NET Framework. This document does not attempt to explain the features of the .NET Framework, except in terms of their APL interfaces. For information concerning the .NET Framework, see the documentation, articles and help files, which are available from Microsoft and other sources.

The .NET interface features include:

- the ability to create and use objects that are instances of .NET Classes.
- the ability to define new .NET Classes in Dyalog that can then be used from other .NET languages such as C# and VB.NET.
- the ability to write Web Services in Dyalog.
- the ability to write ASP.NET web pages in Dyalog.

.NET Classes

The .NET Framework defines a so-called Common Type System. This provides a set of data types, permitted values, and permitted operations. All cooperating languages are supposed to use these types so that operations and values can be checked (by the Common Language Runtime) at run time. The .NET Framework provides its own built-in class library that provides all the primitive data types, together with higher-level classes that perform useful operations.

Dyalog allows you to create and use instances of .NET Classes, thereby gaining access to a huge amount of component technology that is provided by the .NET Framework.

It is also possible to create Class Libraries (Assemblies) in Dyalog. This allows you to export APL technology packaged as .NET Classes, which can then be used from other .NET programming languages such as C# and Visual Basic.

The ability to create and use classes in Dyalog also provides you with the possibility to design APL applications built in terms of APL (and non-APL) components. Such an approach can provide benefits in terms of reliability, software management and re-usage, and maintenance.

GUI Programming with System.Windows.Forms

One of the most important .NET class libraries is called `System.Windows.Forms`, which is designed to support traditional Windows GUI programming. Visual Studio .NET, which is used to develop GUI applications in Visual Basic and C#, produces code that uses `System.Windows.Forms`. Dyalog allows you to use `System.Windows.Forms`, instead of (and in some cases, in conjunction with) the built-in Dyalog GUI objects such as the Dyalog Grid, to program the Graphical User Interface.

Web Services

Web Services are programmable components that can be called by different applications. Web Services have the same goal as COM, but are technically platform independent and use HTTP as the communications protocol with an application. A Web Service can be used either internally by a single application or exposed externally over the Internet for use by any number of applications.

ASP.NET and WebForms

ASP.NET is a new version of Microsoft Active Server Page technology that makes it easier to develop and deploy dynamic Web applications. To supplement ASP.NET, there are some important new .NET class libraries, including WebForms which allow you to build browser-based user interfaces using the same object-oriented mechanism as you use `Windows.Forms` for the Windows GUI. The use of these component libraries replaces basic HTML programming.

ASP.NET pages are server-side scripts, that are usually written in C# or Visual Basic. However, you can also employ Dyalog directly as a scripting language (*APLScript*) to write ASP.NET web pages. In addition, you can call Dyalog workspaces directly from ASP.NET pages, and write custom server-side controls that can be incorporated into ASP.NET pages.

These features give you a wide range of possibilities for using Dyalog to build browser-based applications for the Internet, or for your corporate Intranet.

1.2 Prerequisites

The Dyalog version 20.0 .NET Framework interface requires version 4.0 or greater of Microsoft .NET Framework. It does *not* operate with earlier versions of the .NET Framework. In addition:

- .NET Framework version 4.5 is needed for full Data Binding support (including support for the `INotifyCollectionChanged` interface, which is used by Dyalog to notify a data consumer when the contents of a variable, that is data bound as a list of items, changes).
- IIS needs to be installed before installing Dyalog APL in order to access the examples in the `Samples/asp.net` sub-directory – if IIS and ASP.NET are not present, the `asp.net` sub-directory will not be installed during the Dyalog installation.

Note that .NET Framework is specific to Microsoft Windows; the cross-platform .NET is also supported (see below).

1.3 Files Installed with Dyalog

NET Interface Components

The components used to support the .NET interface are summarised below. Different versions of each component are supplied according to the target platform.

- The Bridge DLL. This is the interface library through which all calls between Dyalog and the .NET Framework are processed
- The DyalogProvider DLL. This DLL performs the initial processing of an `APLScript`.
- The *APLScript* Compiler. This is itself written in Dyalog and packaged as an executable.
- The DyalogNet DLL; a subsidiary library
- The Dyalog DLL. This is the engine that executes all APL code that is hosted by and called from another .NET application.

For a list of the files associated with each of these components, see *Installation/ Configuration: Files And Directories* .

Code Samples

The `samples` subdirectory contains several sub-directories relating to the .NET interface:

- `aplclasses`; a sub-directory that contains examples of .NET classes written in APL.
- `aplscript`; a sub-directory that contains APLScript examples.
- `asp.net`; a sub-directory that is mapped to the IIS Virtual Directory `dyalog.net`, and contains various sample APL Web applications.
- `winforms`; a sub-directory that contains sample applications that use the `System.Windows.Forms` GUI classes.
- `web.config`: this file specifies Dyalog configuration parameters for ASP.NET. See *Section 12.7*.

2 Accessing .Net Classes

2.1 Introduction

.NET classes are implemented as part of the Common Type System. The Type System provides the rules by which different languages can interact with one another. *Types* include interfaces, value types and classes. The .NET Framework provides built-in primitive types plus higher-level types that are useful in building applications.

A *Class* is a kind of Type (as distinct from interfaces and value types) that encapsulates a particular set of methods, events and properties. The word *object* is usually used to refer to an *instance* of a class. An object is typically created by calling the system function `⎕NEW`, with the class as the first element of the argument.

Classes support inheritance in the sense that every class (but one) is based upon another so-called *Base Class*.

An assembly is a file that contains all of the code and metadata for one or more classes. Assemblies can be dynamic (created in memory on-the-fly) or static (files on disk). For the purposes of this document, the term Assembly refers to a file (usually with a .DLL extension) on disk.

2.2 Locating .NET Classes and Assemblies

Unlike COM objects, which are referenced via the Windows Registry, .NET assemblies and the classes they contain, are generally self-contained independent entities (they can be based upon classes in other assemblies). In simple terms, you can install a class on your system by copying the assembly file onto your hard disk and you can de-install it by erasing the file.

Although classes are arranged physically into assemblies, they are also arranged logically into namespaces. These have nothing to do with Dyalog namespaces and, to avoid confusion, are henceforth referred to in this document as .NET namespaces.

Often, a single .NET namespace maps onto a single assembly and usually, the name of the .NET namespace and the name of its assembly file are the same; for example, the .NET namespace `System.Windows.Forms` is contained in an assembly named `System.Windows.Forms.dll`.

However, it is possible for a .NET Namespace to be implemented by more than one assembly; there is not a one-to-one-mapping between .NET Namespaces and assemblies. Indeed, the main top-level .NET Namespace, `System`, is spread over a number of different assembly files.

Within a single .NET Namespace there can be any number of classes, but each has its own unique name. The full name of a class is the name of the class itself, prefixed by the name of the .NET namespace and a dot. For example, the full name of the `DateTime` class in the .NET namespace `System` is `System.DateTime`.

There can be any number of different versions of an assembly installed on your computer, and there can be several .NET namespaces with the same name, implemented in different sets of assembly files; for example, written by different authors.

To use a .NET Class, it is necessary to tell the system to load the assembly (`dll`) in which it is defined. In many languages (including C#) this is done by supplying the *names* of the assemblies or the *pathnames* of the assembly files as a compiler directive.

Secondly, to avoid the verbosity of programmers having to always refer to full class names, the C# and Visual Basic languages allow the .NET namespace prefix to be elided. In this case, the programmer must declare a list of .NET namespaces with `Using` (C#) and `Imports` (Visual Basic) declaration statements. This list is then used to resolve unqualified class names referred to in the code.

In either language, when the compiler encounters the unqualified name of a class, it searches the specified .NET namespaces for that class.

In Dyalog, this mechanism is implemented by the `⎕USING` system variable. `⎕USING` performs the same two tasks that `Using/Imports` declarations and compiler directives provide in C# and Visual Basic; namely to give a list of .NET namespaces to be searched for unqualified class names, and to specify the assemblies which are to be loaded.

`⎕USING` is a vector of character vectors each element of which contains 1 or 2 comma-delimited strings. The first string specifies the name of a .NET namespace; the second specifies the *pathname* of an assembly file. This may be a full pathname or a relative one, but must include the file extension (`.dll`). If just the file name is specified, it is assumed to be located in the standard .NET Framework directory that was specified when the .NET Framework was installed (for example, `C:\Windows\Microsoft.NET\Framework64\v4.0.30319`)

It is convenient to treat .NET namespaces and assemblies in pairs. For example:

```
⎕USING←'System,mscorlib.dll'
⎕USING,←⊂'System.Windows.Forms,System.Windows.Forms.dll'
⎕USING,←⊂'System.Drawing,System.Drawing.dll'
```

Note that because Dyalog APL automatically loads `mscorlib.dll` (which contains the most commonly used classes in the `System` Namespace), it is not actually necessary to specify it explicitly in `⎕USING`.

`⎕USING` has Namespace scope, that is, each Dyalog namespace, class or instance has its own value of `⎕USING` that is initially inherited from its parent space but which may be separately modified. `⎕USING` may also be localised in a function header, so that different functions can declare different search paths for .NET namespaces/assemblies.

If `⎕USING` is empty (`⎕USING←0⍴⊂''`), APL will not search for .NET classes in order to resolve names which would otherwise give a `VALUE ERROR`.

Assigning a simple character vector to `⎕USING` is equivalent to setting it to the enclose of that vector. The statement (`⎕USING←''`) does not empty `⎕USING`, it sets it to a single empty element, which gives access to `mscorlib.dll` and the Bridge DLL without a namespace prefix.

Within a Class script, you may instead employ one or more `:Using` statements to specify the .NET search path. Each of these statements is equivalent to appending an enclosed character vector to `⎕USING`.

```
    :Using System,mscorlib.dll
    :Using System.Windows.Forms,System.Windows.Forms.dll
    :Using System.Drawing,System.Drawing.dll
```

Classes also inherit from the namespace they are contained in. The statement

```
    :Using
```

Is equivalent to

```
    ⎕USING←0⍴⊂''
```

...and allows a class to clear the inherited value before appending to `⎕USING`, or to state that no .NET assemblies should be loaded.

The equivalent to `⎕USING←''` is a `:Using` statement followed by a comma separator but no namespace prefix and no assembly name:

```
   :Using ,
```

2.3 Using .NET Classes

To create a Dyalog object as an instance of a .NET class, you use the ⎕NEW system function. The ⎕NEW system function is monadic. It takes a 1 or 2-element argument, the first element being a *class*.

If the argument is a scalar or a 1-element vector, an instance of the class is created using the *constructor* that takes NO argument.

If the argument is a 2-element vector, an instance of the class is created using the *constructor* whose argument matches the disclosed second element.

For example, to create a `DateTime` object whose value is the 30th April 2008:

```
      ⎕USING←'System'

      mydt←⎕NEW DateTime (2008 4 30)
```

The result of ⎕NEW is an reference to the newly created instance:

```
      ⎕NC ⊂'mydt'
9.2
```

If you format a reference to a .NET Object, APL calls its `ToString` method to obtain a useful description or identification of the object. This topic is discussed in more detail later in this chapter.

```
      mydt
30/04/2008 00:00:00
```

If you want to use fully qualified class names instead, one of the elements of ⎕USING must be an empty vector. For example:

```
      ⎕USING←,⊂''

      mydt←⎕NEW System.DateTime (2008 4 30)
```

When creating an instance of the `DateTime` class, you are required to provide an argument with two elements: (the class and the *constructor argument*, in our case a 3-element vector representing the date). Many classes provide a default constructor which takes no arguments. From Dyalog, the *default constructor* is called by calling ⎕NEW with only a reference to the class in the argument. For example, to obtain a default `Button` object, we only need to write:

```
      mybtn←⎕NEW Button
```

The above statement assumes that you have defined ⎕USING correctly; there must be a reference to `System.Windows.Forms.dll`, and a namespace prefix which allows the name `Button` to be recognised as `System.Windows.Forms.Button`.

The mechanism by which APL associates the class name with a class in a .NET namespace is described below.

Constructors and Overloading

Each .NET Class has one or more *constructor* methods. A constructor is a method which is called to initialise an instance of the Class. Typically, a Class will support several constructor methods - each with a different set of parameters. For example, `System.DateTime` supports a constructor that takes three `Int32` parameters (year, month, day), another that takes six `Int32` parameters (year, month, day, hour, minute, second), and so forth. These different constructor methods are not distinguished by having different names but by the different sets of parameters they accept.

This concept, which is known as *overloading*, may seem somewhat alien to the APL programmer. After all, we are used to defining functions that accept a whole range of different arguments. However, type checking, which is fundamental to the .NET Framework, requires that a method is called with the correct number of parameters, and that each parameter is of a predefined type. Overloading solves this issue.

When you create an instance of a class in C#, you do so using the `new` operator. This is automatically mapped to the appropriate constructor method by matching the parameters you supply to the various forms of the constructor. A similar mechanism is implemented in Dyalog using the ⎕NEW system function.

How the ⎕NEW System Function is implemented

When APL executes an expression such as:

```
      mydt←⎕NEW DateTime (2008 4 30)
```

the following logic is used to resolve the reference to `DateTime` correctly.

The first time that APL encounters a reference to a non-existent name (that is, a name that would otherwise generate a `VALUE ERROR`), it searches the .NET namespaces/ assemblies specified by ⎕USING for a .NET class of that name. If found, the name (in this case `DateTime`) is recorded in the APL symbol table with a name class of 9.6 and is

associated with the corresponding .NET namespace. If not, `VALUE ERROR` is reported as usual. Note that this search ONLY takes place if `⎕USING` has been assigned a value.

Subsequent references to that symbol (in this case `DateTime`) are resolved directly and do not involve any assembly searching.

If you use `⎕NEW` with only a class as argument, APL will attempt to call the version of its constructor that is defined to take no arguments. If no such version of the constructor exists, the call will fail with a `LENGTH ERROR`.

Otherwise, if you use `⎕NEW` with a class as argument and a second element, APL will call the version of the constructor whose parameters match the second element you have supplied to `⎕NEW`. If no such version of the constructor exists, the call will fail with a `LENGTH ERROR`.

Notes

- The value of `⎕USING` is only used when an object is instantiated. Changing the value of `⎕USING` has no effect on objects that have already been instantiated in the active workspace.
- When a workspace containing .Net objects is saved, the names of the Net objects are saved with it but they are not automatically re-instantiated when the workspace is loaded or copied. A reference to such an orphaned object will report `(NULL)`.
- Some functionality might work with .NET Framework or .NET but not both, for example, SharpPlot requires the .NET Framework and does not work with .NET itself.

Displaying a .NET Object

When you display a reference to a .NET object, APL calls the object's `ToString` method and displays the result. All objects provide a `ToString` method because all objects ultimately inherit from the .NET class `System.Object`. Many .NET classes will provide their own `ToString` that overrides the one inherited from `System.Object`, and return a useful description or identifier for the object in question. `ToString` usually supports a range of calling parameters, but APL always calls the version of `ToString` that is defined to take no calling parameters. Monadic format (`⍕`) and monadic `⎕FMT` have been extended to provide the same result, and provides a quick shorthand method to call `ToString` in this way. The default `ToString` supplied by `System.Object` returns the name of the object's Type. This can be changed using the system function `⎕DF`. For example,

```
      z←⎕NEW DateTime ⎕TS
      z.(⎕DF(⍕DayOfWeek),,'G< 99:99>'⎕FMT 100⊥Hour Minute)
      z
Saturday 09:17
```

Note that if you want to check the type of an object, this can be obtained using the `GetType` method, which is supported by all .NET objects.

Disposing of .NET Objects

.NET objects are managed by the .NET Common Language Runtime (CLR). The CLR allocates memory for an object when it is created, and de-allocates this memory when it is no longer required.

When the (last) reference from Dyalog to a .NET object is expunged by `⎕EX` or by localisation, the system marks the object as unused, leaving it to the CLR to de-allocate the memory that it had previously allocated to it, when appropriate. Note that even though Dyalog has de-referenced the APL name, the object could potentially still be referenced by another .NET class.

De-allocated memory may not actually be re-used immediately and may indeed never be re-used, depending upon the algorithms used by the CLR garbage disposal.

Furthermore, a .NET object may allocate unmanaged resources (such as window handles) which are not automatically released by the CLR.

To allow the programmer to control the freeing of resources associated with .NET objects in a standard way, objects implement the IDisposable interface which provides a Dispose() method. The C# language provides a `using` control structure that automates the freeing of resources. Crucially, it does so however the flow of execution exits the control structure, even as a result of error handling. This obviates the need for the programmer to call Dispose() explicitly wherever it may be required.

This programming convenience is provide in Dyalog by the `:Disposable ... :EndDisposable` control structure. For further information, see *Programming: Disposable*

2.4 Browsing .NET Classes

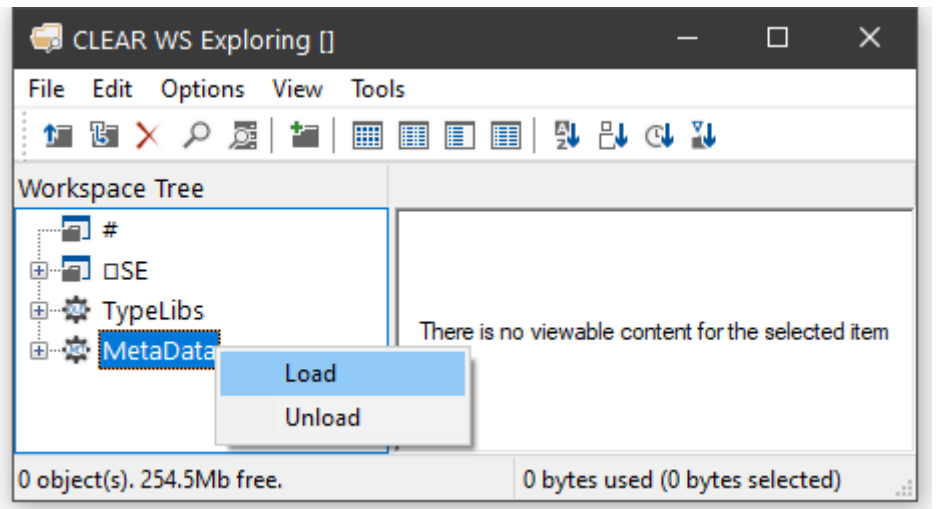
Microsoft supplies a tool for browsing .NET Class libraries called `ILDASM.EXE`[1].

1 ILDASM.EXE can be found in the .NET SDK and is distributed with Visual Studio

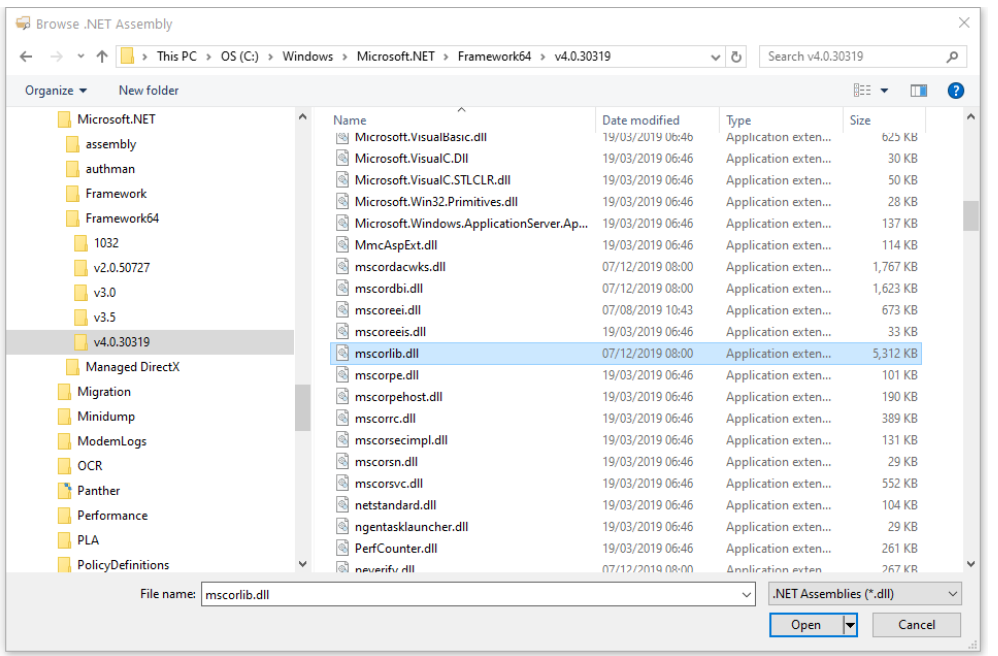
As a convenience, the Dyalog APL Workspace Explorer has been extended to perform a similar task as `ILDASM` so that you can gain access to the information within the context of the APL environment.

The information that describes .NET classes, which is known as its *Metadata*, is part of the definition of the class and is stored with it. This Metadata corresponds to Type Information in COM, which is typically stored in a separate Type Library.

To gain information about one or more .NET Classes, open the Workspace Explorer, right click the *Metadata* folder, and choose *Load*.



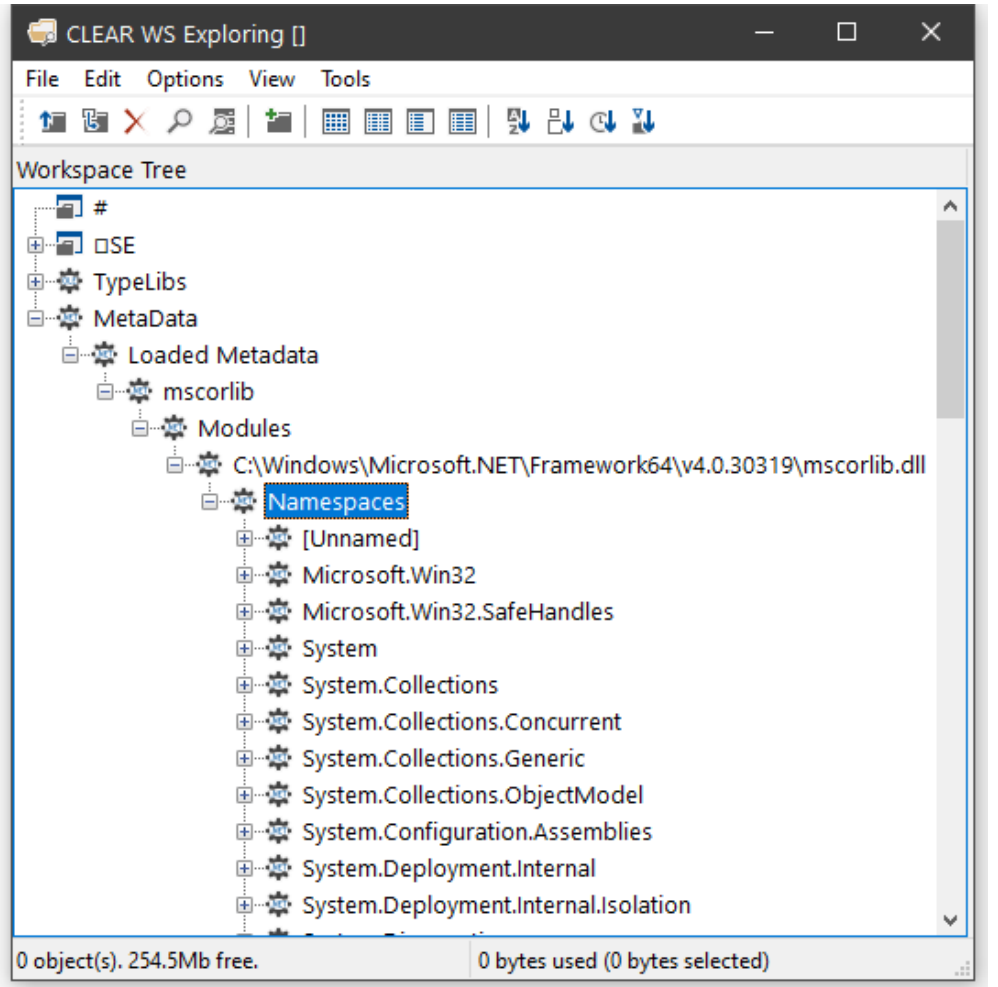
This brings up the *Browse .NET Assembly* dialog box as shown below. Navigate to the .NET assembly of your choice, and click *Open*.



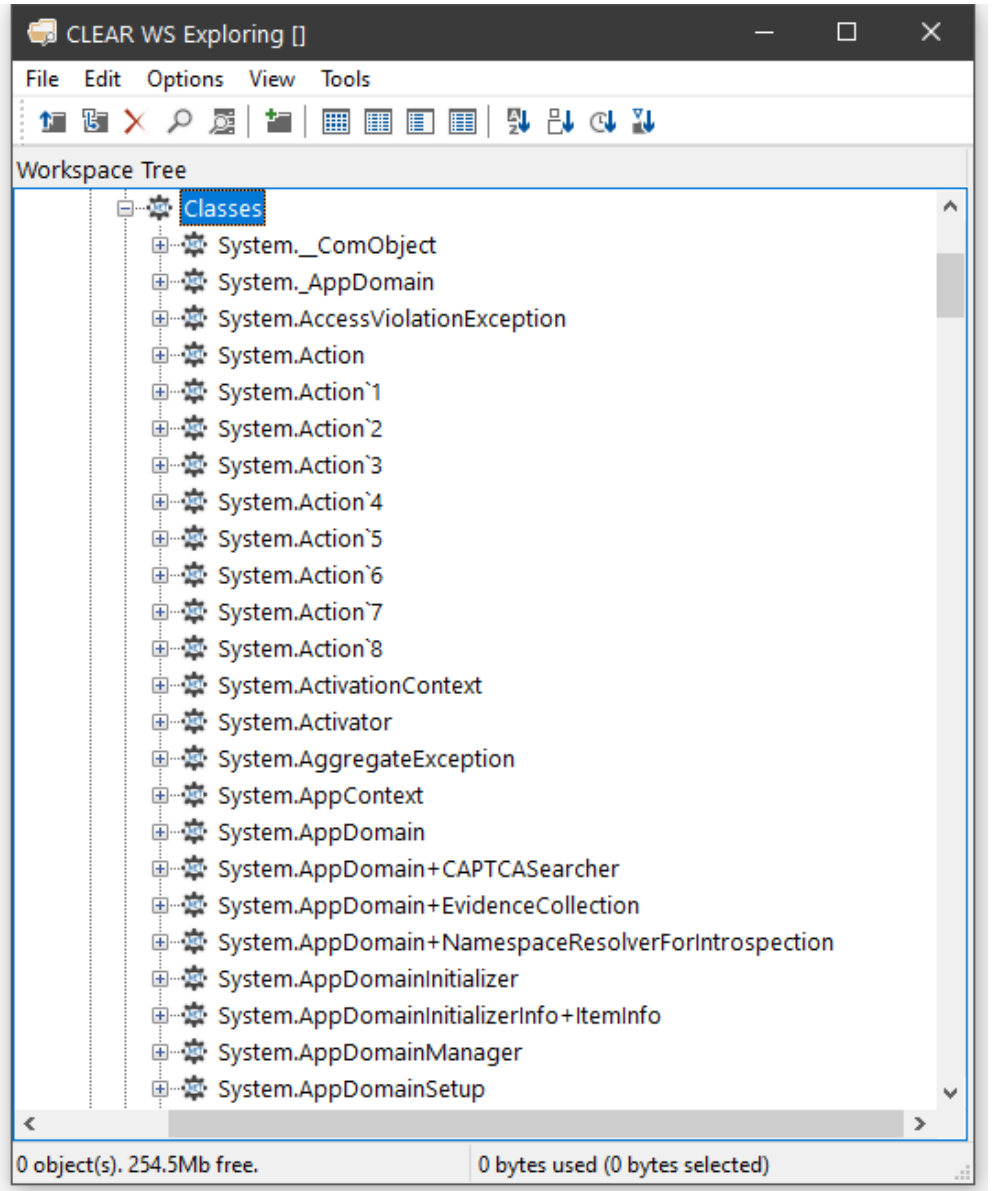
The .NET Classes provided with the .NET Framework are typically located in `C:\WINDOWS\Microsoft.NET\Framework64\V4.0.30319` (on a 64-bit computer). The last named folder is the Version number.

The most commonly used classes of the .NET Namespace System are stored in this directory in an Assembly named `mscorlib.dll`, along with a number of other fundamental .NET Namespaces.

The result of opening this Assembly is illustrated in the following screen shot. The somewhat complex tree structure that is shown in the Workspace Explorer merely reflects the structure of the Metadata itself.

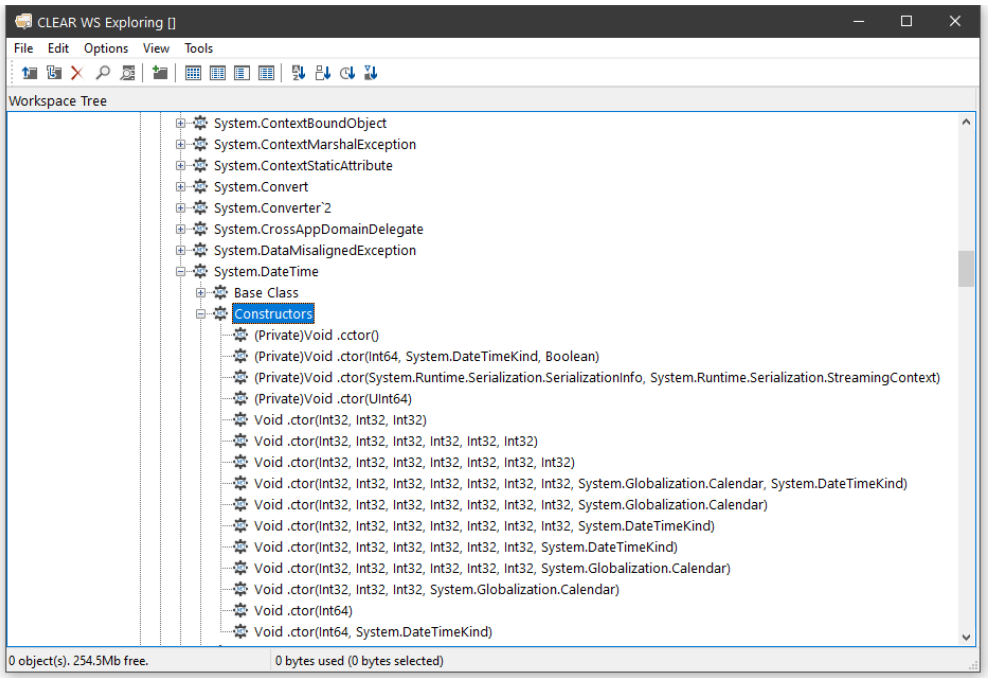


Opening the *System/ Classes* sub-folder causes the Explorer to display the list of classes contained in the .NET Namespace *System* as shown in the picture below.



The *Constructors* folder shows you the list of all of the valid constructors and their parameter sets with which you may create a new instance of the Class by calling `New`. The constructors are those named *.ctor*; you may ignore the one named *.cctor*, (the class constructor) and any labelled as *Private*.

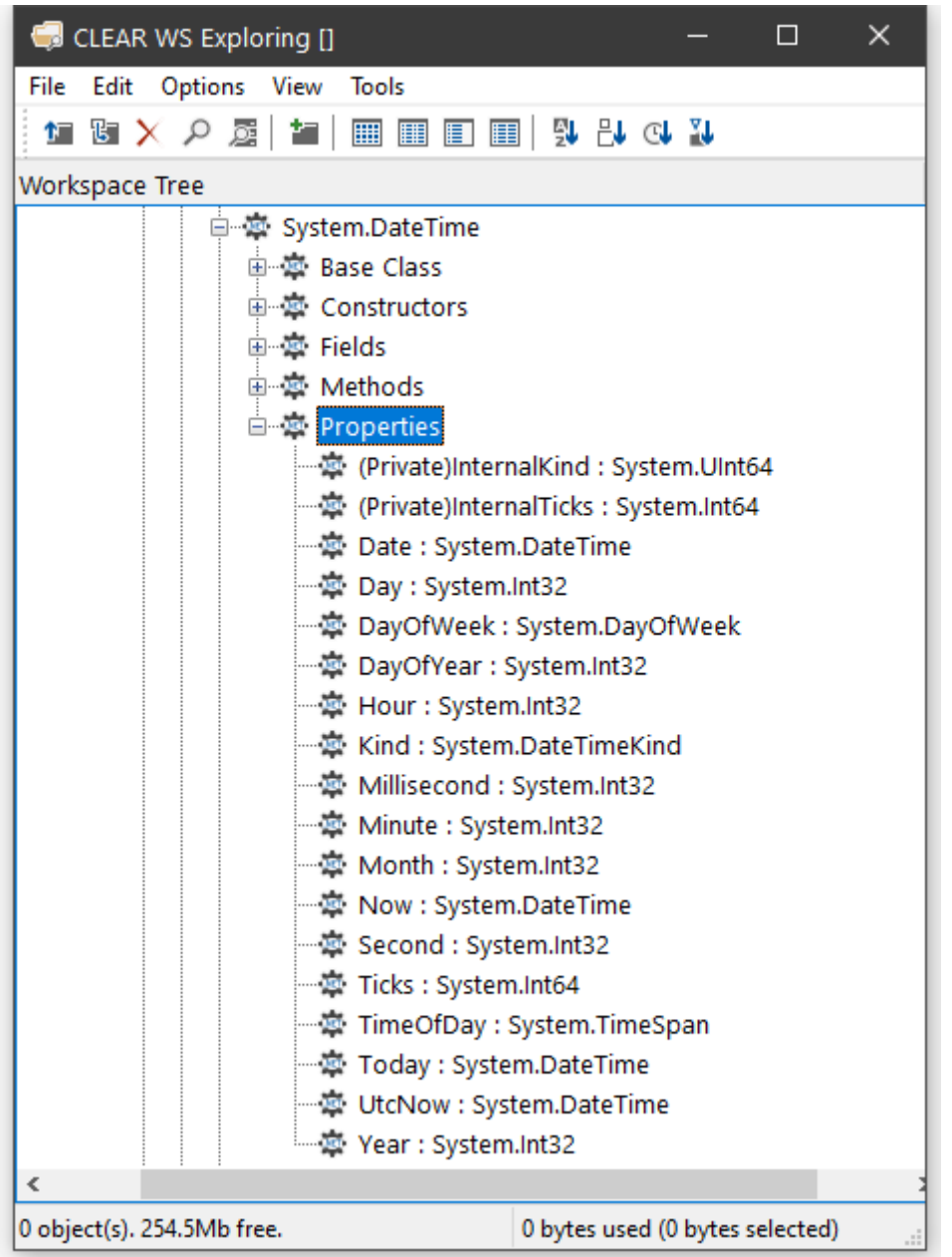
For example, you can deduce that `DateTime.New` may be called with three numeric (`Int32`) parameters, or six numeric (`Int32`) parameters, and so forth. There are in fact seven different ways that you can create an instance of a `DateTime`.



For example, the following statement may be used to create a new instance of `DateTime` (09:30 in the morning on 30th April 2001):

```
      mydt←⎕NEW DateTime (2001 4 30 9 30 0)

      mydt
30/04/2001 09:30:00
```

The *Properties* folder provides a list of the properties supported by the Class. It shows the name of the property followed by its data type. For example, the `DayOfYear` property is defined to be of type `Int32`.



You can query a property by direct reference:

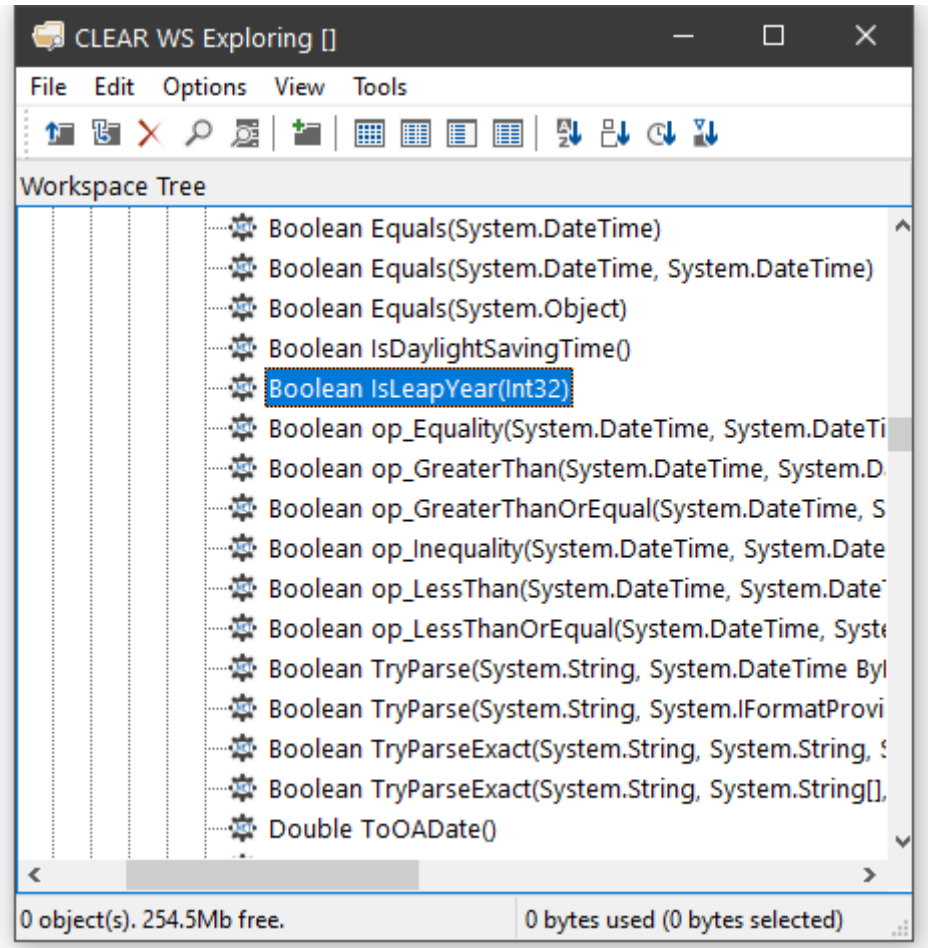
```
      mydt.DayOfWeek
Monday
```

Notice too that the data types of some properties are not simple data types, but Classes in their own right. For example, the data type of the `Now` property is itself `System.DateTime`. This means that when you reference the `Now` property, you get back an object that represents an instance of the `System.DateTime` object:

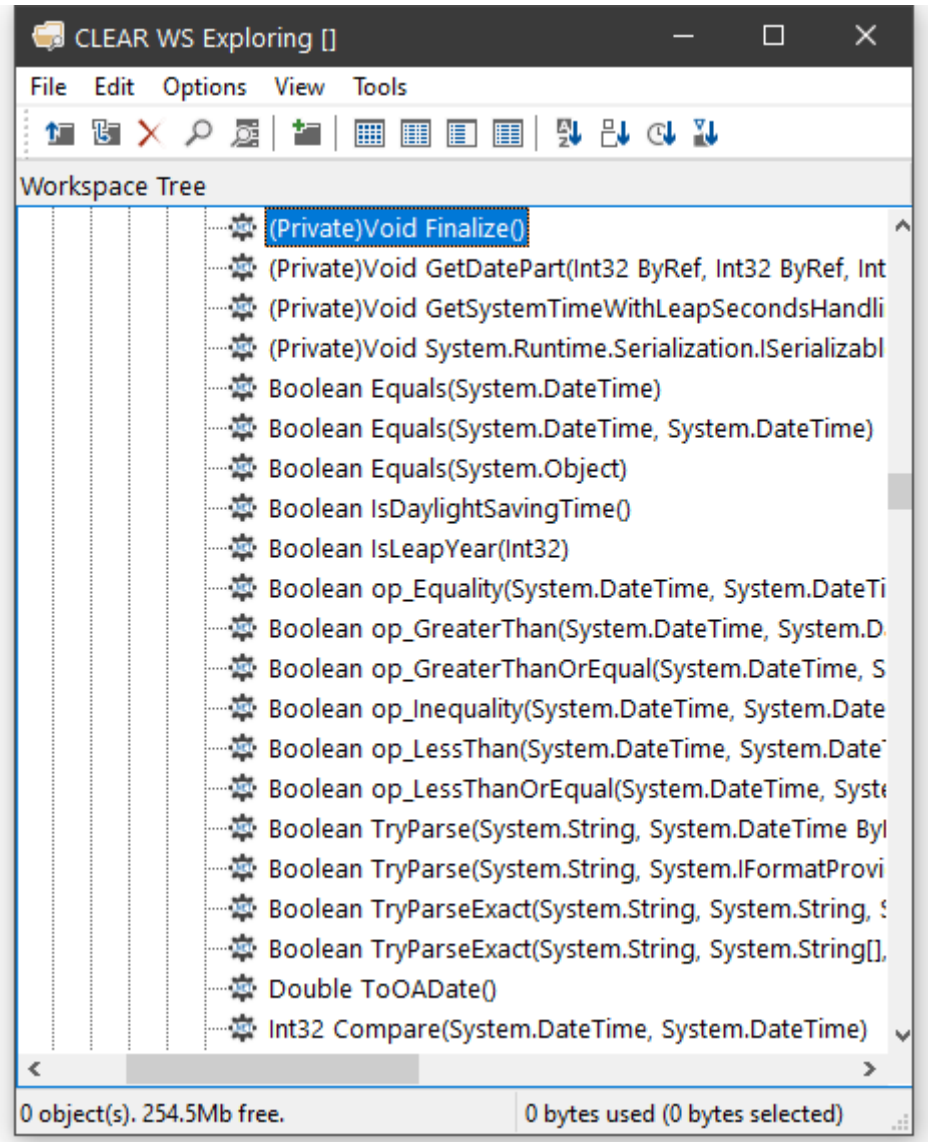
```
      mydt.Now
07/11/2001 11:30:48
      □TS
2001 11 7 11 30 48 0
```

The *Methods* folder lists the methods supported by the Class. The Explorer shows the data type of the result of the method, followed by the name of the method and the types of its arguments. For example, the `IsLeapYear` method takes an `Int32` parameter (year) and returns a `Boolean` result.

```
      mydt.IsLeapYear 2000
1
```



Many of the reported objects are listed as *Private*, which means they are inaccessible to users of the class – you are not able to call them or inspect their value. For more information about classes, see *Programming: Introducing Classes*.

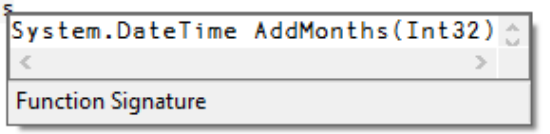


2.5 Value Tips for External Functions

Value Tips can also be used to investigate the syntax of external functions. If you hover over the name of an external function, the Value Tip displays its Function Signature.

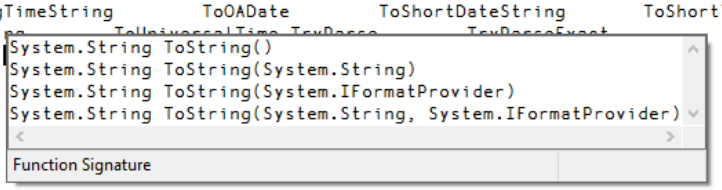
For example, in the example below, the mouse is hovered over the external function `dt.AddMonths` and shows that it requires a single integer as its argument.

```
clear ws
      ⎕USING←'System'
      dt←DateTime.Now
      dt.MethodList
 Add  AddDays  AddHours  AddMilliseconds  AddMinutes  AddMonths
      nMonth  Equals  FromBinary  FromFileTime  FromFileTimeUtc
      TypeCode  IsDaylightSavingTime  IsLeapYear  Parse  ParseExa
      ileTime  ToFileTimeUtc  ToLocalTime  ToLongDateString  ToLo
      ing  ToString  ToUniversalTime  TryParse  TryParseExact
      dt.AddMonths
```



Should the external function provide more than one signature, they are all shown in the Value Tip as illustrated below. Here the function `ToString` has four different overloads.

```
clear ws
      ⎕USING←'System'
      dt←DateTime.Now
      )CS dt
#.[System.DateTime]
      )METHODS
Add     AddDays AddHours        AddMilliseconds AddMinutes      AddMonths
AddSeconds      AddTicks        AddYears        Compare CompareTo       DaysInMonth
Equals  FromBinary      FromFileTime    FromFileTimeUtc FromOADate
GetDateTimeFormats      GetHashCode     GetType GetTypeCode     IsDaylightSavingTime
IsLeapYear      Parse   ParseExact      ReferenceEquals SpecifyKind     Subtract
ToBinary        ToFileTime      ToFileTimeUtc   ToLocalTime     ToLongDateString
ToLongTimeString        ToOADate        ToShortDateString       ToShortTimeString
ToString        ToUniversalTime TryParse        TryParseExact
      dt.ToString
```



2.6 Advanced Techniques

Shared Members

Certain .NET Classes provide methods, fields and properties, that can be called directly without the need to create an instance of the Class first. These *members* are known as shared, because they have the same definition for the class and for any instance of the class.

The methods `Now` and `IsLeapYear` exported by System.DateTime fall into this category. For example:

```
      ⎕USING←,⊂'System'

      DateTime.Now
07/11/2008 11:30:48

      DateTime.IsLeapYear 2000
1
```

APL language extensions for .NET objects

The .NET Framework provides a set of standard operators (methods) that are supported by certain classes. These operators include methods to compare two .NET objects and methods to add and subtract objects.

In the case of the `DateTime` Class, there are operators to compare two `DateTime` objects. For example:

```
      DT1←⎕NEW DateTime (2008 4 30)
      DT2←⎕NEW DateTime (2008 1 1)

     ⍝ Is DT1 equal to DT2 ?
      DateTime.op_Equality DT1 DT2
0
```

The `op_Addition` and `op_Subtraction` operators add and subtract `TimeSpan` objects to `DateTime` objects. For example:

```
      DT3←DateTime.Now
      DT3
07/11/2008 11:33:45

      TS←⎕NEW TimeSpan (1 1 1)
      TS
01:01:01
```

```
      DateTime.op_Addition DT3 TS
07/11/2008 12:34:46

      DateTime.op_Subtraction DT3 TS
07/11/2008 10:32:44
```

The corresponding APL primitive functions have been extended to accept .NET objects as arguments and simply call these standard .NET methods internally. The methods and the corresponding APL primitives are shown in the table below.

Note that calculations and comparisons performed by .NET methods are performed independently from the values of APL system variables (such as ⎕FR and ⎕CT).

.NET Method	APL Primitive Function
op_Addition	+
op_Subtraction	−
op_Multiply	×
op_Division	÷
op_Equality	=
op_Inequality	≠
op_LessThan	<
op_LessThanOrEqual	≤
op_GreaterThan	>
op_GreaterThanOrEqual	≥

So instead of calling the appropriate .NET method to compare two objects, you can use the familiar APL primitive instead. For example:

```
      DT1=DT2
0
      DT1>DT2
1
      DT3+TS
07/11/2008 12:34:46
      DT3-TS
07/11/2008 10:32:44
```

Apart from being easier to use, the primitive functions automatically handle arrays and support scalar extension; for example:

```
      DT1>DT2 DT3
1 0
```

In addition, the monadic form of Grade Up (⍋) and Grade Down (⍒), and the Minimum (⌊) and Maximum (⌈) primitive functions have been extended to work on arrays of references to .NET objects. Note that the argument(s) must be a homogeneous set of references to objects of the same .NET class, and in the case of Grade Up and Grade Down, the argument must be a vector. For example:

```
      ⍋DT1 DT2 DT3
2 1 3
      ⌊/DT1 DT2 DT3
01/01/2008 00:00:00
```

Exceptions

When a .NET object generates an error, it does so by *throwing an exception*. An exception is in fact a .NET class whose ultimate base class is `System.Exception`.

The system constant `⎕EXCEPTION` returns a reference to the most recently generated exception object.

For example, if you attempt to create an instance of a `DateTime` object with a year that is outside its range, the constructor throws an exception. This causes APL to report a (trappable) `EXCEPTION` error (error number 90) and access to the exception object is provided by `⎕EXCEPTION`.

```
      ⎕USING←'System'
      DT←⎕NEW DateTime (100000 0 0)
EXCEPTION
      DT←⎕NEW DateTime (100000 0 0)

      ⎕EN
90
      ⎕EXCEPTION.Message
Year, Month, and Day parameters describe an un-representable DateTime.

      ⎕EXCEPTION.Source
mscorlib

      ⎕EXCEPTION.StackTrace
   at System.DateTime.DateToTicks(Int32 year,
                                  Int32 month,
                                  Int32 day)

   at System.DateTime..ctor(Int32 year,
                            Int32 month,
                            Int32 day)
```

Specifying Overloads and Casts

If a .NET function is overloaded in terms of the types of arguments it accepts, Dyalog chooses which overload to call depending upon the data types of the arguments passed to it. For example, if a .NET function `foo()` is declared to take a single argument either of type `int` or of type `double` APL would call the first version if you called it with an integer value and the second version if you called it with a non-integer value.

In some circumstances it may be desirable to override this mechanism and explicitly specify which overload to use.

A second requirement is to be able to specify to what .NET types APL should coerce arrays before calling a .NET function. For example, if a parameter to a .NET function is declared as type `System.Object`, it might be necessary to force the APL argument to be cast to a particular *type* of `Object` before the function is called.

Both these requirements are met by calling the function via the Variant operator ⍠. There are two options, **OverloadTypes** (the Principle Option) and **CastToTypes**. Each option takes an array of refs to .NET types, the same length as the number of parameters to the function.

OverloadTypes Examples

To force APL to call the double version of function `foo()` regardless of the type of the argument `val`:

```
      (foo ⌷('OverloadTypes'Double))val
```

or more simply:

```
      (foo ⌷Double)val
```

Note that `Double` is a ref to the .NET type `System.Double`.

```
      ⎕USING←'System'
      Double
(System.Double)
```

Taking this a stage further, suppose that `foo()` is defined with 5 overloads as follows:

```
foo()
foo(int i)
foo(double d)
foo(double d, int i)
foo(double[] d)
```

The following statements will call the niladic, double, (double, int) and double[] overloads respectively.

```
(foo ⌷ (⊂⍬)) ⍬                                  ⍝ niladic
(foo ⌷ Double) 1                                ⍝ double
(foo ⌷(⊂Double Int32))1 1                       ⍝ double,int
(foo ⌷(Type.GetType ⊂'System.Double[]'))⊂1 1 ⍝ double[]
```

Note that in the niladic case, an enclosed empty vector is used to represent a null reference to a .NET type.

CastToTypes Example

The .NET function `Array.SetValue()` sets the value of a specified element (or elements) of an array. The first argument, the new value, is declared as `System.Object`, but the value supplied must correspond to the type of the `Array` in question. APL has no means to know what this is and will therefore pass the value *as is*, that is, in whatever internal format it happens to be at the time. For example:

```
      ⎕USING←'System'

      ⍝ create a Boolean  array with 2 elements
      BA←Array.CreateInstance Boolean 2
      BA.GetValue 0 ⍝ get the 0th element
0

      ⍝ attempt to set the 0th element to 1 (AKA true)

      BA.SetValue 1 0
EXCEPTION: Cannot widen from source type to target type
either because the source type is a not a primitive type or the conversi
on cannot be accomplished.
test[5] BA.SetValue 1 0
       ^
```

The above expression failed because APL passed the first argument 1 ,unchanged from its current internal representation, as a 1-byte integer which does not fit into a Boolean element.

To rectify the situation, APL must be told to cast the argument to a Boolean as follows:

```
      (BA.SetValue ⍠ ('CastToTypes'(Boolean Int32)))1 0
      BA.GetValue 0 ⍝ get the 0th element
1
```

Overloaded Constructors

If a class provides constructor overloads, a similar mechanism is used to specify which of the constructors is to be used when an instance of the class is created using ⎕NEW.

For example, if `MyClass` is a .NET class with an overloaded constructor, and one of its constructors is defined to take two parameters; a `double` and an `int`, the following statement would create an instance of the class by calling that specific constructor overload:

```
      (⎕NEW ⍠ (⊂Double Int32)) MyClass (1 1)
```

2.7 More Examples

Directory and File Manipulation

The .NET Namespace `System.IO` (also in the Assembly `mscorlib.dll`) provides some useful facilities for manipulating files. For example, you can create a `DirectoryInfo` object associated with a particular directory on your computer, call its `GetFiles` method to obtain a list of files, and then get their `Name` and `CreationTime` properties.

```
      ⎕USING←,⊂'System.IO'
      d←⎕NEW DirectoryInfo (⊂'C:\Dyalog')
```

`d` is an instance of the `Directory` Class, corresponding to the directory `c:\Dyalog`[2].

```
      d
C:\Dyalog
```

The `GetFiles` method returns a list of files; actually, `FileInfo` objects, that represent each of the files in the directory: Its optional argument specifies a filter; for example:

```
      d.GetFiles ⊂'*.exe'
 evalstub.exe  exestub.exe  dyalog.exe  dyalogrt.exe
```

The `Name` property returns the name of the file associated with the `File` object:

```
      (d.GetFiles ⊂'*.exe').Name
 evalstub.exe  exestub.exe  dyalog.exe  dyalogrt.exe
```

And the `CreationTime` property returns its creation time, which is a `DateTime` object:

```
      (d.GetFiles ⊂'*.exe').CreationTime
```

01/04/2004 09:37:01 01/04/2004 09:37:01 08/06/2004 ...

If you call `GetFiles` without an argument (in APL, with an argument of `⍬`), it returns a complete list of files:

```
      files←d.GetFiles ⍬
```

2 In this document, we will refer to the location where Dyalog is installed as C:\Dyalog. Your installation of Dyalog might be in a different folder or even on a different drive but the examples should work just the same it you replace C:\Dyalog by your folder name

Taking advantage of namespace reference array expansion, an expression to display file names and their creation times is as follows.

```
      files,[1.5]files.CreationTime
relnotes.hlp       03/02/2004 11:47:02
relnotes.cnt       03/02/2004 11:47:02
def_uk.dse         22/03/2004 12:13:31
DIALOGS.HLP        22/03/2004 12:13:31
dyares32.dll       22/03/2004 12:13:40
...
```

Sending an email

The .NET Namespace `System.Web.Mail` provides objects for handing email.

You can create a new email message as an instance of the `MailMessage` class, set its various properties, and then send it using the `SmtpMail` class.

Please note that these examples will only work if your computer is configured to allow you to send email in this way.

```
      ⎕USING←'System.Web.Mail,System.Web.dll'
      m←⎕NEW MailMessage
      m.From←'tony.blair@uk.gov'
      m.To←'sales@dyalog.com'
      m.Subject←'order'
      m.Body←'Send me 100 copies of Dyalog now'

      SmtpMail.Send m
```

However, note that the `Send` method of the `SmtpMail` object is overloaded and may be called with a single parameter of type `System.Web.Mail.MailMessage` as above, or four parameters of type `System.String`:

So instead, you can just say:

```
      SmtpMail.Send 'tony.blair@uk.gov'
                    'sales@dyalog.com'
                    'order'
                    'Send me the goods'
```

Web Scraping

The .NET Framework provides a whole range of classes for accessing the internet from a program. The following example illustrates how you can read the contents of a web page. It is complicated, but realistic, in that it includes code to cater for a firewall/proxy connection to the internet. It is only 9 lines of APL code, but each line requires careful explanation.

First we need to define ⎕USING so that it specifies all of the .NET Namespaces and Assemblies that we require.

```
      ⎕USING←'System,System.dll' 'System.Net' 'System.IO'
```

The `WebRequest` class in the .NET Namespace `System.Net` implements the .NET Framework's request/response model for accessing data from the Internet. In this example we create a `WebRequest` object associated with the URI `https://www.example.com`. Note that `WebRequest` is an example of a static class. You don't make instances of it; you just use its methods.

```
      wrq←WebRequest.Create ⊂'https://www.example.com'
```

In fact (and somewhat confusingly) if the URI specifies a scheme of "http://" or "https://", you get back an object of type `HttpWebRequest` rather than a plain and simple `WebRequest`. So, at this stage, `wrq` is an `HttpWebRequest` object.

```
      wrq
System.Net.HttpWebRequest
```

This class has a `Proxy` property through which you specify the proxy information for a request made through a firewall. The value assigned to the `Proxy` property has to be an object of type `System.Net.WebProxy`. So first we must create a new `WebProxy` object specifying the hostname and port number for the firewall. You will need to change this statement to suit your own internet configuration (it may even not be necessary to do this).

```
      PX←⎕NEW WebProxy(⊂'http://dyagate.dyadic.com:8080')
      PX
System.Net.WebProxy
```

Having set up the `WebProxy` object as required, we then assign it to the `Proxy` property of the `HttpRequest` object `wrq`.

```
      wrq.Proxy←PX
```

The `HttpRequest` class has a `GetResponse` method that returns a response from an internet resource. No it's not HTML (yet), the result is an object of type `System.Net.HttpWebResponse`.

```
      wr←wrq.GetResponse
      wr
System.Net.HttpWebResponse
```

The `HttpWebResponse` class has a `GetResponseStream` method whose result is of type `System.Net.ConnectStream`. This object, whose base class is `System.IO.Stream`, provides methods to read and write data both synchronously and asynchronously from a data source, which in this case is physically connected to a TCP/IP socket.

```
      str←wr.GetResponseStream
      str
System.Net.ConnectStream
```

However, there is yet another step to consider. The `Stream` class is designed for byte input and output; what we need is a class that reads characters in a byte stream using a particular encoding. This is a job for the `System.IO.StreamReader` class. Given a `Stream` object, you can create a new instance of a `StreamReader` by passing it the `Stream` as a parameter.

```
      rdr←⎕NEW StreamReader str
      rdr
System.IO.StreamReader
```

Finally, we can use the `ReadToEnd` method of the `StreamReader` to get the contents of the page.

```
      s←rdr.ReadToEnd
      ⍴s
45242
```

Note that to avoid running out of connections, it is necessary to close the Stream:

```
      str.Close
```

2.8 Enumerations

An enumeration is a set of named constants that may apply to a particular operation. For example, when you open a file you typically want to specify whether the file is to be opened for reading, for writing, or for both. A method that opens a file will take a parameter that allows you to specify this. If this is implemented using an enumerated

constant, the parameter may be one of a specific set of (typically) integer values; for example, 1=read, 2=write, 3=both read and write. However, to avoid using meaningless numbers in code, it is conventional to use names to represent particular values. These are known as *enumerated constants* or, more simply, as *enums*.

In the .NET Framework, *enums* are implemented as classes that inherit from the base class `System.Enum`. The class as a whole represents a set of enumerated constants; each of the constants themselves is represented by a static field within the class.

The next chapter deals with the use of `System.Windows.Forms` to create and manipulate the user interface. The classes in this .NET Namespace use *enums* extensively.

For example, there is a class named `System.Windows.Forms.FormBorderStyle` that contains a set of static fields named `None`, `FixedDialog`, `Sizeable`, and so forth. These fields have specific integer values, but the values themselves are of no interest to the programmer.

Typically, you use an enumerated constant as a parameter to a method or to specify the value of a property. For example, to create a Form with a particular border style, you would set its `BorderStyle` property to one of the members of the `FormBorderStyle` class, viz.

```
⎕USING←'System'
  ⎕USING,←⊂'System.Windows.Forms,system.windows.forms.dll'
f1←⎕NEW Form
f1.BorderStyle←FormBorderStyle.FixedDialog
FormBorderStyle.⎕NL ¯2 ⍝ List enum members
Fixed3D  FixedDialog  FixedSingle  FixedToolWindow  None
Sizable  SizableToolWindow
```

An enum has a value, which you may use in place of the enum itself when such usage is unambiguous. For example, the `FormBorderStyle.Fixed3D` enum has an underlying value is 2:

```
     Convert.ToInt32 FormBorderStyle.Fixed3D
2
```

You could set the border style of the Form `f1` to `FormBorderStyle.Fixed3D` with the expression:

```
      f1.BorderStyle←2
```

However, this practice is not recommended. Not only does it make your code less clear, but also if a value for a property or a parameter to a method may be one of several different enum types, APL cannot tell which is expected and the call will fail.

For example, when the constructor for `System.Drawing.Font` is called with 3 parameters, the 3rd parameter may be either a `FontStyle` enum or a `GraphicsUnit` enum. If you were to call Font with a 3rd parameter of 1, APL cannot tell whether this refers to a `FontStyle` enum, or a `GraphicsUnit` enum, and the call will fail.

2.9 Handling Pointers with Dyalog.ByRef

Certain .NET methods take parameters that are pointers.

An example is the `DivRem` method that is provided by the `System.Math` class. This method performs an integer division, returning the quotient as its result, and the remainder in an address specified as a pointer by the calling program.

APL does not have a mechanism for dealing with pointers, so Dyalog provides a .NET class for this purpose. This is the `Dyalog.ByRef` class, which is a provided by an Assembly that is loaded automatically by the Dyalog program.

Firstly, to gain access to the Dyalog .NET Namespace, it must be specified by `⎕USING`. Note that you need not specify the Assembly (DLL) from which it is obtained (the Bridge DLL), because (like `mscorlib.dll`) it is automatically loaded by when APL starts.

```
      ⎕USING←'System' 'Dyalog'
```

The `Dyalog.ByRef` class represents a pointer to an object of type `System.Object`. It has a number of constructors, some of which are used internally by APL itself. You only need to be concerned about two of them; the one that takes no parameters, and the one that takes a single parameter of type `System.Object`. The former is used to create an empty pointer; the latter to create a pointer to an object or some data.

For example, to create an empty pointer:

```
      ptr1←⎕NEW ByRef
```

Or, to create pointers to specific values,

```
      ptr2←⎕NEW ByRef 0
      ptr3←⎕NEW ByRef (⊂⍳10)
      ptr4←⎕NEW ByRef (⎕NEW DateTime (2000 4 30))
```

Notice that a single parameter is required, so you must enclose it if it is an array with several elements. Alternatively, the parameter may be a .NET object.

The ByRef class has a single property called `Value`.

```
      ptr2.Value
0
      ptr3.Value
1 2 3 4 5 6 7 8 9 10
      ptr4.Value
30/04/2000 00:00:00
```

Note that if you reference the Value property without first setting it, you get a `VALUE ERROR`.

```
      ptr1.Value
VALUE ERROR
      ptr1.Value
     ^
```

Returning to the example, we recall that the `DivRem` method takes 3 parameters:

1. the numerator
2. the denominator
3. a pointer to an address into which the method will write the remainder after performing the division.

```
      remptr←⍬NEW ByRef
      remptr.Value
VALUE ERROR
      remptr.Value
      ^
      Math.DivRem 311 99 remptr
3
      remptr.Value
14
```

In some cases a .NET method may take a parameter that is an Array and the method expects to fill in the array with appropriate values. In APL there is no syntax to allow a parameter to a function to be modified in this way. However, we can use the `Dyalog.ByRef` class to call this method. For example, the `System.IO.FileStream` class contains a `Read` method that populates its first argument with the bytes in the file.

```
      ⎕USING←'System.IO' 'Dyalog' 'System'
      fs←⎕NEW FileStream ('c:\tmp\jd.txt' FileMode.Open)
      fs.Length
25
      fs.Read(arg←⎕NEW ByRef,⊂⊂25⍴0)0 25
25
      arg.Value
104 101 108 108 111 32 102 114 111 109 32 106 111 104 110 32 100 97 105
110 116 114 101 101 10
```

2.10 DECF Conversion

Incoming .NET data types VT_DECIMAL (96-bit integer) and VT_CY (currency value represented by a 64-bit two's complement integer, scaled by 10,000) are converted to 126-bit decimal numbers (DECFs). This conversion is performed independently of the value of ⎕FR.

If you want to perform arithmetic on values imported in this way, then you should set ⎕FR to 1287, at least for the duration of the calculations.

Note that the .NET interface converts System.Decimal to DECFs but does not convert System.Int64 to DECFs.

3 Using Windows.Forms

3.1 Introduction

`System.Windows.Forms` is a .NET namespace that provides a set of classes for creating the Graphical User Interface for Windows applications.

As an alternative to the built-in Dyalog GUI, Windows Forms has been superseded by Windows Presentation Foundation which is described in the next Chapter. This section is included to support existing Dyalog applications that make use of Windows Forms.

Unless otherwise specified, all the examples described in this Chapter may be found in the `samples\winforms\winforms.dws` workspace.

3.2 Creating GUI Objects

GUI objects are represented by .NET classes in the .NET Namespace `System.Windows.Forms`. In general, these classes correspond closely to the GUI objects provided by Dyalog, which are themselves based upon the Windows API.

For example, to create a form containing a button and an edit field, you would create instances of the `Form`, `Button` and `TextBox` classes.

3.3 Object Hierarchy

The most striking difference between the `Windows.Forms` GUI and the Dyalog GUI is that in `Windows.Forms` the container hierarchy represented by forms, group boxes, and controls is not represented by an object hierarchy. Instead, objects that represent GUI controls are created stand-alone (that is, without a parent) and then associated with a container, such as a `Form`, by calling the `Add` method of the parent's Controls collection. Notice too that `Windows.Forms` objects are associated with APL symbols that are namespace references, but `Windows.Forms` objects do not have implicit names.

3.4 Positioning and Sizing Forms and Controls

The position of a form or a control is specified by its `Location` property, which is measured relative to the top left corner of the client area of its container.

`Location` has a data type of `System.Drawing.Point`. To set `Location`, you must first create an object of type `System.Drawing.Point` then assign that object to `Location`.

Similarly, the size of an object is determined by its `Size` property, which has a data type of `System.Drawing.Size`. This time, you must create a `System.Drawing.Size` object before assigning it to the `Size` property of the control or form.

Objects also have `Top(Y)` and `Left(X)` properties that may be specified or referenced independently. These accept simple numeric values.

The position of a `Form` may instead be determined by its `DeskTopLocation` property, which is specified relative to the taskbar. Another alternative is to set the `StartPosition` property whose default setting is `WindowsDefaultLocation`, which represents a computed best location.

3.5 Modal Dialog Boxes

Dialog Boxes are displayed modally to prevent the user from performing tasks outside of the dialog box.

To create a modal dialog box, you create a `Form`, set its `BorderStyle` property to `FixedDialog`, set its `ControlBox`, `MinimizeBox` and `MaximizeBox` properties to false, and display it using `ShowDialog`.

A modal dialog box has a `DialogResult` property that is set when the `Form` is closed, or when the user presses OK or Cancel. The value of this property is returned by the `ShowDialog` method, so the simplest way to handle user actions is to check the result of `ShowDialog` and proceed accordingly. Example 1 illustrates a simple modal dialog box.

Example 1

Function `EG1` illustrates how to create and use a simple modal dialog box. Much of the function is self-explanatory, but the following points are noteworthy.

`EG1[1-2]` set `USING` to include the .NET Namespaces `System.Windows.Forms` and `System.Drawing`.

`EG1[6,8,9]` create a `Form` and two `Button` objects. As yet, they are unconnected. The constructor for both classes is defined to take no arguments, so the `⎕NEW` system function is only called with a class argument.

`EG1[14]` shows how the `Location` property is set by first creating a new `Point` object with a specific pair of (x and y) values.

`EG1[18]` computes the values for the `Point` object for `button2.Location`, from the values of the `Left`, `Height` and `Top` properties of `button1`; thus positioning `button2` relative to `button1`.

```
     ∇ EG1;form1;button1;button2;true;false;⎕USING;Z
[1]    ⎕USING←,⊂'System.Windows.Forms,
                 System.Windows.Forms.dll'
[2]    ⎕USING,←⊂'System.Drawing,System.Drawing.dll'
[3]    true false←1 0
[4]
[5]    ⍝ Create a new instance of the form.
[6]    form1←⎕NEW Form
[7]    ⍝ Create two buttons to use as the accept and cancel btns
[8]    button1←⎕NEW Button
[9]    button2←⎕NEW Button
[10]
[11]   ⍝ Set the text of button1 to "OK".
[12]   button1.Text←'OK'
[13]   ⍝ Set the position of the button on the form.
[14]   button1.Location←⎕NEW Point,⊂10 10
[15]   ⍝ Set the text of button2 to "Cancel".
[16]   button2.Text←'Cancel'
[17]   ⍝ Set the position of the button relative to button1.
[18]   button2.Location←⎕NEW Point,
                    ⊂button1.Left button1.(Height+Top+10)
[19]
```

`EG1[21,23]` sets the `DialogResult` property of `button1` and `button2` to `DialogResult.OK` and `DialogResult.Cancel` respectively. Note that `DialogResult` is an enumeration with a predefined set of member values.

Similarly, `EG1[32]` defines the `BorderStyle` property of the form using the `FormBorderStyle` enumeration.

`EG1[38 40]` defines the `AcceptButton` and `CancelButton` properties of the Form to `button1` and `button2` respectively. These have the same effect as the Dyalog GUI Default and Cancel properties.

`EG1[42]` sets the `StartPosition` of the Form to be centre screen. Once again this is specified using an enumeration; `FormStartPosition`.

```
[20]  ⍝ Make button1's dialog result OK.
[21]   button1.DialogResult←DialogResult.OK
[22]  ⍝ Make button2's dialog result Cancel.
[23]   button2.DialogResult←DialogResult.Cancel
[24]
[25]
[26]   ⍝ Set the title bar text of the form.
[27]   form1.Text←'My Dialog Box'
[28]   ⍝ Display a help button on the form.
[29]   form1.HelpButton←true
[30]
[31]   ⍝ Define the border style of the form to that of a
                                                  dialog box.
[32]   form1.BorderStyle←FormBorderStyle.FixedDialog
[33]   ⍝ Set the MaximizeBox to false to remove the
                                                maximize box.
[34]   form1.MaximizeBox←false
[35]   ⍝ Set the MinimizeBox to false to remove the
                                                minimize box.
[36]   form1.MinimizeBox←false
[37]   ⍝ Set the accept button of the form to button1.
[38]   form1.AcceptButton←button1
[39]   ⍝ Set the cancel button of the form to button2.
[40]   form1.CancelButton←button2
[41]   ⍝ Set the start position of the form to the centre
                                               of the screen.
[42]   form1.StartPosition←FormStartPosition.CenterScreen
[43]
```

`EG1[45 46]` associate the buttons with the Form. The `Controls` property of the `Form` returns an object of type `Form.ControlCollection`. This class has an `Add` method that is used to add a control to the collection of controls that are owned by the `Form`.

`EG1[50]` calls the `ShowDialog` method (with no argument; hence the `⍬`). The result is an object of type `Form.DialogResult`, which is an enumeration.

`EG1[52]` compares the result returned by `ShowDialog` with the enumeration member `DialogResult.OK` (note that the primitive function = has been extended to compare objects).

```
[44]   ⍝ Add button1 to the form.
[45]   form1.Controls.Add button1
[46]   ⍝ Add button2 to the form.
[47]   form1.Controls.Add button2
[48]
[49]   ⍝ Display the form as a modal dialog box.
[50]   Z←form1.ShowDialog ⍬
[51]   ⍝ Determine if the OK button was clicked on the
                                              dialog box.
[52]   :If Z=DialogResult.OK
[53]      ⍝ Display a message box saying that the OK
                                         button was clicked.
[54]       Z←MessageBox.Show⊂'The OK button on the form
                                             was clicked.'
[55]   :Else
[56]      ⍝ Display a message box saying that the Cancel
                                         button was clicked.
[57]       Z←MessageBox.Show⊂'The Cancel button on the
                                           form was clicked.'
[58]   :EndIf
     ∇
```

> **Warning**
>
> The use of modal forms in .NET can lead to problematic situations while debugging. As the control is passed to .NET the APL interpreter cannot regain control in the event of an unforeseen error. It is advisable to change the code to something like the following until the code is fully tested:

```
[52]   form1.Visible←1
[53]   :While form1.Visible ⋄ :endwhile
```

Example 2

Functions `EG2` and `EG2A` illustrate how the Each operator (¨) and the extended namespace reference syntax in Dyalog may be used to produce more succinct, and no less readable, code.

```
     ∇ EG2;form1;label1;textBox1;true;false;⎕USING;Z
[1]    ⎕USING←,⊂'System.Windows.Forms,
                 System.Windows.Forms.dll'
[2]    ⎕USING,←⊂'System.Drawing,System.Drawing.dll'
[3]    true false←1 0
[4]
[5]    ⍝ Create a new instance of the form.
[6]    form1←⎕NEW Form
[7]
[8]    textBox1←⎕NEW TextBox
[9]    label1←⎕NEW Label
[10]
[11]   ⍝ Initialize the controls and their bounds.
[12]   label1.Text←'First Name'
[13]   label1.Location←⎕NEW Point (48 48)
[14]   label1.Size←⎕NEW Size (104 16)
[15]   textBox1.Text←''
[16]   textBox1.Location←⎕NEW Point (48 64)
[17]   textBox1.Size←⎕NEW Size (104 16)
[18]
[19]   ⍝ Add the TextBox control to the form's control
          collection.
[20]   form1.Controls.Add textBox1
[21]   ⍝ Add the Label control to the form's control
          collection.
[22]   form1.Controls.Add label1
[23]
[24]   ⍝ Display the form as a modal dialog box.
[25]   Z←form1.ShowDialog ⍬
     ∇
```

`EG2A[7]` takes advantage of the fact that .NET classes are namespaces, so the expression `Form TextBox Label` is a vector of namespace refs, and the expression `⎕NEW¨Form TextBox Label` runs the `⎕NEW` system function on each of them.

Similarly, `EG2A[10 11 12]` combine the use of extended namespace reference and the *Each* operator to set the `Text`, `Location` and `Size` properties in several objects together.

```
     ∇ EG2A;form1;label1;textBox1;true;false;⎕USING;Z
[1]    ⍝ Compact version of EG2 taking advantage of ref
          syntax and ¨
[2]    ⎕USING←'System.Windows.Forms,System.Windows.Forms.dll'
[3]    ⎕USING,←⊂'System.Drawing,System.Drawing.dll'
[4]    true false←1 0
[5]
[6]    ⍝ Create a new instance of the form, TextBox and Label.
[7]    (form1 textBox1 label1)←⎕NEW¨Form TextBox Label
[8]
[9]    ⍝ Initialize the controls and their bounds.
[10]   (label1 textBox1).Text←'First Name' ''
[11]   (label1 textBox1).Location←⎕NEW¨Point,¨⊂¨(48 48)(48 64)
[12]   (label1 textBox1).Size←⎕NEW¨Size,¨⊂¨(104 16)(104 16)
[13]
[14]   ⍝ Add the Label and TextBox controls to the form's
          control collection.
[15]   form1.Controls.AddRange⊂label1 textBox1
[16]
[17]   ⍝ Display the form as a modal dialog box.
[18]   Z←form1.ShowDialog ⍬
     ∇
```

3.6 Non-Modal Forms

Non-modal Forms are displayed using the `Run` method of the `System.Windows.Forms.Application` object. This method is designed to be called once, and only once, during the life of an application and this poses problems during APL development. Fortunately, it turns out that, in practice, the restriction is that `Application.Run` may only be run once on a single system thread. However, it may be run successively on different system threads. During development, you may therefore test a function that calls `Application.Run`, by running it on a new APL thread using Spawn (`&`). See Chapter 13 for further details.

DataGrid Examples

Three functions in the `samples\winforms\winforms.dws` workspace provide examples of non-modal Forms. These examples also illustrate the use of the WinForms.DataGrid class.

Function `Grid1` is an APL translation of the example given in the help file for the DataGrid class in the .NET SDK Beta1. The original code has been slightly modified to work with the current version of the SDK.

Function `Grid2` is an APL translation of the example given in the help file for the DataGrid class in the .NET SDK Beta2.

Function `Grid` is an APL translation of the example given in the file:

```
C:\Program Files\Microsoft.NET\SDK\v1.1\...
QuickStart\winforms\samples\Data\Grid\vb\Grid.vb
```

This example uses Microsoft SQL Server 2000 to extract sample data from the sample NorthWind database. To run this example, you must have SQL Server running and you must modify function `Grid_Load` to specify the name of your server.

GDIPLUS Workspace

The `samples\winforms\gdiplus.dws` workspace contains a sample that demonstrates the use of non-rectangular Forms. It is a direct translation into APL from a C# sample (WinForms-Graphics-GDIPlusShape) that was distributed on the Visual Studio .NET Beta 2 Resource CD.

TETRIS Workspace

The `samples\winforms\tetris.dws` workspace contains a sample that demonstrates the use of graphics. It is a direct translation into APL from a C# sample (WinForms-Graphics-Tetris) that was distributed on the Visual Studio .NET Beta 2 Resource CD.

WEBSERVICES Workspace

An example of a non-modal Form is provided by the `WFGOLF` function in the `samples\asp.net\webservices\webservices.dws` workspace. This function performs exactly the same task as the `GOLF` function in the same workspace, but it uses Windows.Forms instead of the built-in Dyalog GUI.

`WFGOLF`, and its callback functions `WFBOOK` and `WFSS` perform exactly the same task, with almost identical dialog box appearance, of `GOLF` and its callbacks `BOOK` and `SS` that are described in Chapter 7.

Note that when you run `WFGOLF` or `GOLF` for the first time, you must supply an argument of 1 to force the creation of the proxy class for the `GolfService` web service.

4 WPF

4.1 Introduction

Windows Presentation Foundation is a graphical system that includes a programmable Graphical User Interface. It is supplied as a set of Microsoft .NET assemblies and is supported on all current Windows platforms.

The WPF GUI is in many ways more sophisticated and powerful than either Dyalog's own built-in GUI or the GUI provided by Windows Forms.

Like any other set of .NET classes, WFP can be integrated into Dyalog applications via the .NET interface. Dyalog users may therefore develop GUI applications that are based upon WPF as an alternative to the built-in Dyalog GUI or Windows Forms.

Quite apart from its advanced GUI capabilities, WPF supports *data binding*. This is a complex subject, but putting it very simply, data binding allows a property of a user-interface object (such as the Text property of a TextBox object) to be bound to some data. When the data changes, the bound property of the object changes and vice versa.

Dyalog includes a data binding function (`2015⌶`[3]) which supports data binding to APL arrays and namespaces.

A WPF GUI can be built dynamically by creating a set of component objects (using `⎕NEW`) in a similar way to the Dyalog GUI and Windows Forms. However, the same user-interface can instead be specified statically using XAML, a text markup system that describes the GUI using XML. Along with data binding, this feature allows the application logic and the user-interface to be developed and maintained separately.

The examples described in this section are provided in the workspace `WPFIntro.dws`

[3] This function may remain as an I-beam or be replaced by one or more system functions in a future Version of Dyalog.

4.2 Temperature Converter

4.2.1 Temperature Converter Tutorial

This tutorial illustrates how to go about developing a simple WPF application in Dyalog. It is functionally identical to the GUI tutorial example that illustrates how to develop a GUI application using the built-in Dyalog Graphical User Interface. See Interface Guide:

GUI Tutorial.

Like the GUI Tutorial, this is necessarily an elementary example, but illustrates the principles that are involved. The example is a simple Temperature Converter.

The user may enter a temperature value in either Fahrenheit or Centigrade and have it converted to the other scale.

No attempt has been made to update the WPF example, in terms of its user-interface, from the original version which was developed for Windows 3. This allows a direct comparison to be made between using the WPF and using the built-in Dyalog GUI.

There are two versions provided. The first uses XAML to describe the user-interface with code to drive it. The second version is written entirely in APL code. The two versions of this example may be found in `WPFIntro.dws` in the namespaces `UsingXAML` and `UsingCode` respectively.

4.2.2 Using XAML

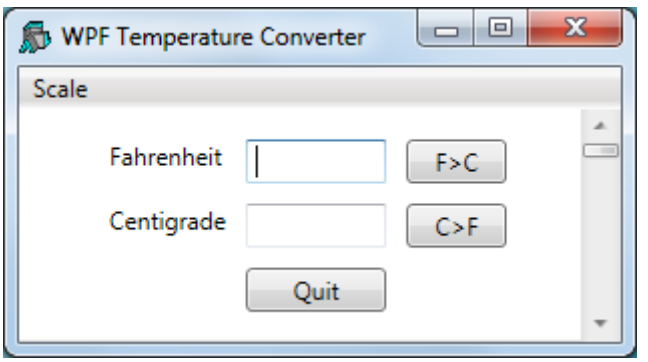
The functions and data for this example are provided in the workspace `WPFIntro.dws` in the namespace `WPF.UsingXAML`. To run the example:

```
      )LOAD wpfintro
      WPF.UsingXAML.TempConverter
```

Arguably the easiest way to create a WPF GUI is to define it using XAML. The XAML defines the structure, layout and appearance of the user-interface in a very concise manner. It is still necessary to write code to display the XAML and to respond to user actions, but the amount of code involved is minimal.

The XAML for the Temperature Converter is shown below.

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Name="Temp"
 Title="WPF Temperature Converter"
 SizeToContent="WidthandHeight">
  <DockPanel LastChildFill="False">
    <Menu DockPanel.Dock="Top">
        <MenuItem Header="_Scale">
            <MenuItem Name="mnuFahrenheit" Header="_Fahrenheit"
             IsCheckable="True" IsChecked="True"/>
            <MenuItem Name="mnuCentigrade" Header="_Centigrade"
             IsCheckable="True"/>
        </MenuItem>
    </Menu>
    <Grid Width="230" Margin="40,10,10,10">
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="80"/>
        <ColumnDefinition Width="60"/>
      </Grid.ColumnDefinitions>
    <Label Grid.Row="0" Grid.Column="0" Content="Fahrenheit"/>
    <Label Grid.Row="1" Grid.Column="0" Content="Centigrade"/>
    <TextBox Name="txtFahrenheit" Grid.Row="0" Grid.Column="1"
     Margin="5"/>
    <TextBox Name="txtCentigrade" Grid.Row="1" Grid.Column="1"
     Margin="5"/>
    <Button Name="btnF2C" Grid.Row="0" Grid.Column="2"
     Content="F>C" Margin="5"/>
    <Button Name="btnC2F" Grid.Row="1" Grid.Column="2"
     Content="C>F" Margin="5"/>
    <Button Name="btnQuit" Grid.Row="2" Grid.Column="1"
     Content="Quit" Margin="5"/>
    </Grid>
    <ScrollBar Name="scrTemp" DockPanel.Dock="Right"  Width="20"
     Orientation="Vertical" Minimum="1" Maximum="213">
    </ScrollBar>
  </DockPanel>
</Window>
```



The window defined by this XAML is illustrated in the screen image shown above. Let us examine the XAML, component by component.

Parent and Child Controls

First, notice how the structure of the GUI is defined by enclosing the child components inside the opening and closing tags of its parent. So:

```
<Window
...
  <DockPanel>
...
  </DockPanel>
</Window>
```

specifies a Window control that *contains* a DockPanel control.

Similarly,

```
    <Menu>
        <MenuItem ... >
            <MenuItem ... />
            <MenuItem ... />
        </MenuItem>
    </Menu>
```

defines a Menu that contains a MenuItem, that itself contains two other MenuItem objects.

Named and Un-named Controls

Secondly, notice that certain objects are *named* whereas others are not. For example: `TextBox Name="mnuFahrenheit` defines a TextBox named *txtFahenheit*; whereas `<DockPanel ...>` defines an unnamed DockPanel object.

Objects are given names so that they can be referenced from the code that displays content in the user-interface or handles the user actions. In this case, the code will read the content of the *txtFahrenheit* TextBox but has no need to reference the DockPanel.

The Main Window

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Name="Temp"
 Title="WPF Temperature Converter"
 SizeToContent="WidthandHeight">
...
</Window>
```

This extract of XAML defines a Window control; a top-level window that is equivalent to a Dyalog GUI Form.

The *xmlns* attributes define the XML namespaces (effectively the vocabulary of the xml scheme) and are mandatory in an XAML document.

The name of the TextBox is *Temp*, and its caption is *WFP Temperature Converter*. The SizeToContent property is set to "WidthandHeight", which causes the TextBox to automatically size itself to fit its content in both horizontal and vertical directions.

The DockPanel

```
 <DockPanel LastChildFill="False">
..
 </DockPanel>
```

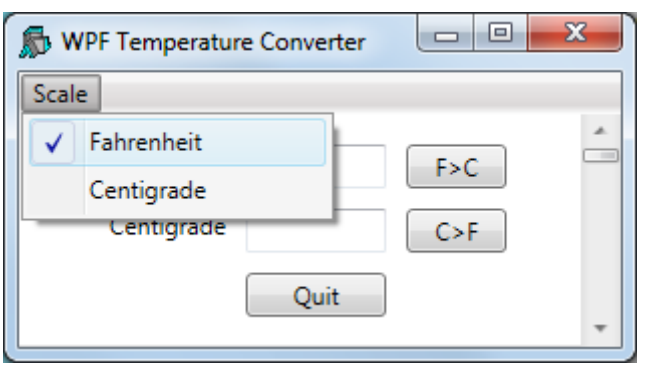
WPF provides a number of *layout controls*. These are containers whose only purpose is to arrange child controls in a particular way, and to dictate how they are re-arranged when the parent window is resized. The DockPanel is one of the simplest of the WPF layout controls.

In this case, the DockPanel is controlling 3 child windows a Menu, a Grid and a ScrollBar.

The attachment of a particular child control is specified by setting its DockPanel.Dock property. By default, the last control added to a DockPanel is stretched to fill the remaining space when the window is expanded. In this case, the requirement is for a fixed-width scrollbar attached to the right edge, so the default is overridden by setting the LastChildFill property to "False".

The Menu

```
    <Menu DockPanel.Dock="Top">
        <MenuItem Header="_Scale">
            <MenuItem Name="mnuFahrenheit" Header="_Fahrenheit"
             IsCheckable="True" IsChecked="True"/>
            <MenuItem Name="mnuCentigrade" Header="_Centigrade"
             IsCheckable="True"/>
        </MenuItem>
    </Menu>
```



The above extract from the XAML defines a Menu. Setting Dock to "Top" causes the Menu as a whole to be docked, so that it appears like a menubar, along the top of the DockPanel. The Menu contains a single MenuItem labelled *Scale* which itself contains two sub-items labelled *Fahrenheit* and *Centigrade* respectively. The IsCheckable property specifies whether or not the user can check the MenuItem, and the IsChecked property sets and reports its checked state. The underscore characters (for example, as in "_Scale") identify the following character as a keyboard shortcut.

The Grid

```
<Grid Width="230" Margin="40,10,10,10">
...
</Grid>
```

The Grid object is another WPF layout control that organises other controls in rows and columns. Here, the XAML defines a Grid with a width of 230; a left margin if 40, and a top, right and bottom margin of 10. As there is no explicit unit specified, the system uses the default device-independent unit (*px*) of 1/96th inch.

The rows and columns of a Grid are defined by collections of RowDefinition and ColumnDefinition objects.

Here the XAML specifies that the Grid contains 3 rows, each of which has a Height set to "Auto" which means that its height depends upon the height of its content.

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto"/>
  <RowDefinition Height="Auto"/>
  <RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
```

Similarly, there are 3 columns. The first column (which will contain labels) takes its width from its content, that is, it will be just wide enough to display the longest label. The other columns for the edit boxes and buttons are specified to be 80px and 60px wide respectively. In this case, the content (TextBox and Button objects) will take their widths from that of the column.

```
<Grid.ColumnDefinitions>
   <ColumnDefinition Width="Auto"/>
   <ColumnDefinition Width="80"/>
   <ColumnDefinition Width="60"/>
</Grid.ColumnDefinitions>
```

The Label Objects(Column 1)

```
<Label Grid.Row="0" Grid.Column="0" Content="Fahrenheit"/>
<Label Grid.Row="1" Grid.Column="0" Content="Centigrade"/>
```

Here the XAML specifies Label objects Fahrenheit and Centigrade. Because they are defined within the `<Grid> ...</Grid>` tags, they are child objects of the Grid. In

addition it is necessary to specify in which cells they are displayed using their Grid.Row and Grid.Column properties. Note that the cell coordinates have zero origin.

The TextBox Objects(Column 2)

```
    <TextBox Name="txtFahrenheit" Grid.Row="0" Grid.Column="1"
     Margin="5"/>
    <TextBox Name="txtCentigrade" Grid.Row="1" Grid.Column="1"
     Margin="5"/>
```

The XAML specifies two TextBox objects named *txtFahrenheit* and *txtCentigrade* respectively. Setting Margin to "5" means that a margin of 5px is applied around each edge; otherwise the text boxes would occupy the entire width of the column (80px). The effective width of each TextBox will therefore be 70px `(80-2×5)`.

The Button Objects (Column 3)

```
    <Button Name="btnF2C" Grid.Row="0" Grid.Column="2"
     Content="F>C" Margin="5"/>
    <Button Name="btnC2F" Grid.Row="1" Grid.Column="2"
     Content="C>F" Margin="5"/>
    <Button Name="btnQuit" Grid.Row="2" Grid.Column="1"
     Content="Quit" Margin="5"/>
```

The XAML specifies three named Button controls. Note that the caption on a Button is specified by its Content property.

The ScrollBar Object

This example uses a ScrollBar which the user may scroll to input a value, either in Fahrenheit or Centigrade depending upon which of the two menu items (*Fahrenheit* or *Centigrade*) is checked.[4]

```
    <ScrollBar Name="scrTemp" DockPanel.Dock="Right"  Width="20"
     Orientation="Vertical" Minimum="1" Maximum="213">
    </ScrollBar>
```

4 A ScrollBar is not the ideal choice of control for this type of user interation, but this example is designed to look and behave like the original Dyalog GUI example, which was written for the original version of Dyalog for Microsoft Windows.

This XAML snippet defines a ScrollBar named *scrTemp*.

Setting DockPanel.Dock to "Right" means that it will be docked (aligned) on the right edge of the DockPanel. It will be a vertical scrollbar, have a fixed width of 20px and a default height. The range of the ScrollBar is defined by its Minimum and Maximum properties which are set so that the ScrollBar will specify a value in Fahrenheit.

Note that in order to cause the ScrollBar to be docked (aligned) along the right edge of the DockPanel it is necessary to set LastChildFill to "False" (for the DockPanel) and Dock to "Right" (for the ScrollBar), because the value of LastChildFill (default "True") overrides the Dock value of the last defined child of the DockPanel.

> **Note**
>
> The XAML that defines this user-interface is at the same time both simple and complex. It is simple because (in this case) it is readily understood. It is complex because in order to write it, the user-interface designer must understand precisely how the various controls and their properties behave and work together. For these details, you should refer to the appropriate documentation and check out the large number of examples published on the internet.

The Code to display the XAML

The function `TempConverter` shown below contains the code needed to display and operate the user interface whose layout is defined by the XAML described above.

```
      ∇
TempConverter;str;xml;win;txtFahrenheit;txtCentigrade;mnuFahrenheit;mnuC
entigrade;btnF2C;btnC2F;btnQuit;scrTemp;sink
[1]     ⎕USING←'System'
[2]     ⎕USING,←⊂'System.IO'
[3]     ⎕USING,←⊂'System.Windows.Markup'
[4]     ⎕USING,←⊂'System.Xml,system.xml.dll'
[5]     ⎕USING,←⊂'System.Windows.Controls.Primitives,WPF/
PresentationFramework.dll'
[6]
[7]     str←⎕NEW StringReader(⊂XAML)
[8]     xml←⎕NEW XmlTextReader str
[9]     win←XamlReader.Load xml
[10]
[11]    txtFahrenheit←win.FindName⊂'txtFahrenheit'
[12]    txtCentigrade←win.FindName⊂'txtCentigrade'
[13]    mnuFahrenheit←win.FindName⊂'mnuFahrenheit'
[14]    mnuFahrenheit.onClick←'SET_F'
[15]    mnuCentigrade←win.FindName⊂'mnuCentigrade'
[16]    mnuCentigrade.onClick←'SET_C'
[17]    (btnF2C←win.FindName⊂'btnF2C').onClick←'f2c'
[18]    (btnC2F←win.FindName⊂'btnC2F').onClick←'c2f'
[19]    (btnQuit←win.FindName⊂'btnQuit').onClick←'Quit'
[20]    (scrTemp←win.FindName⊂'scrTemp').onScroll←'F2C'
[21]    sink←win.ShowDialog
      ∇
```

The variable `XAML` (a character vector) contains the XAML described previously.

Note that apart from the names given to the objects by the XAML and used by the function, the XAML and the code are independent.

`TempConverter[7-8]` create a XamlReader object from the character vector via StringReader and XmlTextReader objects.

```
[7]     str←⎕NEW StringReader(⊂XAML)
[8]     xml←⎕NEW XmlTextReader str
```

`TempConverter[9]` instantiates the XAML content by calling its Load method, which returns a reference `win` to the top-level control (in this case a Window) defined therein. The Window is not yet visible.

```
[9]     win←XamlReader.Load xml
```

Earlier, it was explained that objects defined by the XAML must be *named* in order that they can be referenced (used) by the code. The mechanism to achieve this is to call the FindName method of the Window, which returns a reference to the specified (named) object. So these statements:

```
[11]   txtFahrenheit←win.FindName⊂'txtFahrenheit'
[12]   txtCentigrade←win.FindName⊂'txtCentigrade'
```

obtain refs (in this case named `txtFahrenheit` and `txtCentigrade`) to objects named *txtFahrenheit* and *txtCentigrade*. It is convenient (but not essential) to use the same name for the ref as is used for the control.

Most of the remaining statements obtain refs to the MenuItem, Button and ScrollBar objects and attach callback functions to their Click and Scroll events respectively.

```
[13]   mnuFahrenheit←win.FindName⊂'mnuFahrenheit'
[14]   mnuFahrenheit.onClick←'SET_F'
[15]   mnuCentigrade←win.FindName⊂'mnuCentigrade'
[16]   mnuCentigrade.onClick←'SET_C'
[17]   (btnF2C←win.FindName⊂'btnF2C').onClick←'f2c'
[18]   (btnC2F←win.FindName⊂'btnC2F').onClick←'c2f'
[19]   (btnQuit←win.FindName⊂'btnQuit').onClick←'Quit'
[20]   (scrTemp←win.FindName⊂'scrTemp').onScroll←'F2C'
```

Finally the code displays the Window and hands it over to the user by calling the `ShowDialog` method of the top-level Window.

```
[21]   sink←win.ShowDialog
```

`ShowDialog` displays the Window *modally*; that is, until it is closed, the user may interact only with that Window. It is equivalent to `⎕DQ 'win'` or `win.Wait` in the Dyalog built-in GUI.

The CallBack Functions

The callback functions are named as they are in the basic Dyalog GUI example and are remarkably similar. See Interface Guide:

GUI Tutorial.

Callback function `f2c` which is attached to the Click event of the `btnF2C` button (labelled *F>C*) reads the character string in the `txtFahrenheit` TextBox, converts it to a number using `Text2Num`, calculates the equivalent in centigrade and then displays the result in the `txtCentigrade` TextBox.

```
     ∇ f2c;value
[1]    ⍝ Callback to convert Fahrenheit to Centigrade
[2]    :If 1=⍴,value←Text2Num txtFahrenheit.Text
[3]        txtCentigrade.Text←2⍕(value-32)×5÷9
[4]    :Else
[5]        txtCentigrade.Text←'invalid'
[6]    :EndIf
     ∇
```

For completeness, the `Text2Num` function is shown below. Note that if the user enters an invalid number, `Text2Num` returns an empty vector, and the callback displays the text *invalid* instead.

```
     ∇ num←Text2Num txt;val
[1]    val num←⍎⍞VFI txt
[2]    num←val/num
     ∇
```

The `c2f` function converts from Centigrade to Fahrenheit when the user presses the button labelled *C>F*.

```
     ∇ c2f;value
[1]   ⍝ Callback to convert Centigrade to Fahrenheit
[2]    :If 1=⍴,value←Text2Num txtCentigrade.Text
[3]        txtFahrenheit.Text←2⍕32+value÷5÷9
[4]    :Else
[5]        txtFahrenheit.Text←'invalid'
[6]    :EndIf
     ∇
```

The callbacks `F2C` and `C2F`, one of which at a time is attached to the Scroll event of the ScrollBar object are shown below. The argument `Msg` contains two items, namely:

`[1]`	Object	a ref to the ScrollBar object
`[2]`	Object	a ref to an object of type System.Windows.Controls.Primitives.ScrollEventArgs

In this case the code uses the NewValue property of the ScrollEventArgs object. An alternative would be to refer to the Value property of the ScrollBar object

```
     ∇ F2C Msg;C;F;val
[1]   ⍝ Callback for Fahrenheit input via scrollbar
[2]    txtFahrenheit.Text←2⍕val←213-(2⊃Msg).NewValue
[3]    txtCentigrade.Text←2⍕(val-32)×5÷9
     ∇
```

```
     ∇ C2F Msg;C;F;val
[1]   ⍝ Callback for Centigrade input via scrollbar
[2]    txtCentigrade.Text←2⍕val←101-(2⊃Msg).NewValue
[3]    txtFahrenheit.Text←2⍕32+val÷5÷9
     ∇
```

The callbacks `SET_F` and `SET_C` which are attached to the Click events of the two MenuItem objects are shown below.

```
     ∇ SET_F
[1]   ⍝ Sets the scrollbar to work in Fahrenheit
[2]    scrTemp.(Minimum Maximum)←1 213
[3]    scrTemp.onScroll←'F2C'
[4]    mnuFahrenheit.IsChecked←1
[5]    mnuCentigrade.IsChecked←0
     ∇

     ∇ SET_C
[1]   ⍝ Sets the scrollbar to work in Centigrade
[2]    scrTemp.(Minimum Maximum)←1 101
[3]    scrTemp.onScroll←'C2F'
[4]    mnuCentigrade.IsChecked←1
[5]    mnuFahrenheit.IsChecked←0
     ∇
```

Finally, the callback function `Quit` which is attached to the Click event on the *Quit* button, simply calls the Close method of the Window:

```
     ∇ Quit arg
[1]    win.Close
     ∇
```

Notice that unlike its equivalent in the Dyalog GUI, it is not appropriate to close the Window using the expression `⎕EX 'win'`. This would expunge the ref to the Window but have no effect on the Window itself.

4.2.3 Using Code

The functions for this example are provided in the workspace `WPFIntro.dws` in the namespace `WPF.UsingCode`. To run the example:

```
)LOAD wpfintro
WPF.UsingCode.TempConverter
```

The following function `TempConverter` performs *exactly* the same task of defining and manipulating the user-interface for the Temperature Converter example using XAML which was discussed previously.

The callback functions it uses are identical.

```
    ∇
TempConverter;⎕USING;win;dp;mnu;mnuFahrenheit;mnuCentigrade;gr;tn;rd1;rd
2;rd3;rc1;rc2;rc3;l1;l2;txtFahrenheit;txtCentigrade;btnF2C;btnC2F;btnQui
t;sink;mnuScale;scrTemp
[1]
[2]    ⎕USING←,⊂'System.Windows.Controls,WPF/PresentationFramework.dll'
[3]    ⎕USING,←⊂'System.Windows.Controls.Primitives,WPF/
PresentationFramework.dll'
[4]    ⎕USING,←⊂'System.Windows,WPF/PresentationFramework.dll'
[5]    ⎕USING,←⊂'System.Windows,WPF/PresentationCore.dll'
[6]
[7]    win←⎕NEW Window
[8]    win.SizeToContent←SizeToContent.WidthAndHeight
[9]    win.Title←'WPF Temperature Converter'
[10]
[11]   dp←⎕NEW DockPanel
[12]   dp.LastChildFill←0
[13]
[14]   mnu←⎕NEW Menu
[15]
[16]   mnuScale←⎕NEW MenuItem
[17]   mnuScale.Header←'_Scale'
[18]   sink←mnu.Items.Add mnuScale
[19]
[20]   mnuFahrenheit←⎕NEW MenuItem
[21]   mnuFahrenheit.Header←'Fahrenheit'
[22]   mnuFahrenheit.IsCheckable←1
[23]   mnuFahrenheit.IsChecked←1
[24]   mnuFahrenheit.onClick←'SET_F'
[25]   sink←mnuScale.Items.Add mnuFahrenheit
[26]
[27]   mnuCentigrade←⎕NEW MenuItem
[28]   mnuCentigrade.Header←'_Centigrade'
[29]   mnuCentigrade.IsCheckable←1
[30]   mnuCentigrade.IsChecked←0
[31]   mnuCentigrade.onClick←'SET_C'
[32]   sink←mnuScale.Items.Add mnuCentigrade
[33]
[34]   sink←dp.Children.Add mnu
[35]   dp.SetDock mnu Dock.Top
[36]
[37]   gr←⎕NEW Grid
[38]   gr.Width←230
[39]   gr.Margin←⎕NEW Thickness(40 10 10 10)
```

```
[40]
[41]   rd1←⎕NEW RowDefinition
[42]   rd1.Height←GridLength.Auto
[43]   rd2←⎕NEW RowDefinition
[44]   rd2.Height←GridLength.Auto
[45]   rd3←⎕NEW RowDefinition
[46]   rd3.Height←GridLength.Auto
[47]   gr.RowDefinitions.Add¨rd1 rd2 rd3
[48]
[49]   rc1←⎕NEW ColumnDefinition
[50]   rc1.Width←GridLength.Auto
[51]   rc2←⎕NEW ColumnDefinition
[52]   rc2.Width←⎕NEW GridLength 80
[53]   rc3←⎕NEW ColumnDefinition
[54]   rc3.Width←⎕NEW GridLength 60
[55]   gr.ColumnDefinitions.Add¨rc1 rc2 rc3
[56]
[57]   l1←⎕NEW Label
[58]   l1.Content←'Fahrenheit'
[59]   sink←gr.Children.Add l1
[60]   gr.SetRow l1 0
[61]   gr.SetColumn l1 0
[62]
[63]   l2←⎕NEW Label
[64]   l2.Content←'Centigrade'
[65]   sink←gr.Children.Add l2
[66]   gr.SetRow l2 1
[67]   gr.SetColumn l2 0
[68]
[69]   txtFahrenheit←⎕NEW TextBox
[70]   txtFahrenheit.Margin←⎕NEW Thickness 5
[71]   sink←gr.Children.Add txtFahrenheit
[72]   gr.SetRow txtFahrenheit 0
[73]   gr.SetColumn txtFahrenheit 1
[74]
[75]   txtCentigrade←⎕NEW TextBox
[76]   txtCentigrade.Margin←⎕NEW Thickness 5
[77]   sink←gr.Children.Add txtCentigrade
[78]   gr.SetRow txtCentigrade 1
[79]   gr.SetColumn txtCentigrade 1
[80]
[81]   btnF2C←⎕NEW Button
[82]   btnF2C.Content←'F>C'
[83]   btnF2C.Margin←⎕NEW Thickness 5
```

```
[84]   btnF2C.onClick←'f2c'
[85]   sink←gr.Children.Add btnF2C
[86]   gr.SetRow btnF2C 0
[87]   gr.SetColumn btnF2C 2
[88]
[89]   btnC2F←⎕NEW Button
[90]   btnC2F.Content←'C>F'
[91]   btnC2F.Margin←⎕NEW Thickness 5
[92]   btnC2F.onClick←'c2f'
[93]   sink←gr.Children.Add btnC2F
[94]   gr.SetRow btnC2F 1
[95]   gr.SetColumn btnC2F 2
[96]
[97]   btnQuit←⎕NEW Button
[98]   btnQuit.Content←'Quit'
[99]   btnQuit.Margin←⎕NEW Thickness 5
[100]  btnQuit.onClick←'Quit'
[101]  sink←gr.Children.Add btnQuit
[102]  gr.SetRow btnQuit 2
[103]  gr.SetColumn btnQuit 1
[104]
[105]  sink←dp.Children.Add gr
[106]
[107]  scrTemp←⎕NEW ScrollBar
[108]  scrTemp.Width←20
[109]  scrTemp.Orientation←Orientation.Vertical
[110]  scrTemp.Minimum←1
[111]  scrTemp.Maximum←213
[112]  scrTemp.onScroll←'F2C'
[113]
[114]  sink←dp.Children.Add scrTemp
[115]  dp.SetDock scrTemp Dock.Right
[116]
[117]  win.Content←dp
[118]
[119]  sink←win.ShowDialog
     ∇
```

Although this approach appears at first sight to be considerably more verbose than using XAML (a 120-line function compared with a 21-line function and a 44-line block of XAML) each line of code performs only one very simple task, and no attempt has been made to write utility functions to perform the same task for similar controls, as might be done in a real application.

As before, let us examine the code line-by-line.

`TempConverter[2-5]` define `⎕USING` so that the appropriate .NET assemblies are on the search-path. Note that the ScrollBar control is in `System.Windows.Controls.Primitives` and not `System.Windows.Controls` like the others.

```
[2]    ⎕USING←,⊂'System.Windows.Controls,WPF/PresentationFramework.dll'
[3]    ⎕USING,←⊂'System.Windows.Controls.Primitives,WPF/
PresentationFramework.dll'
[4]    ⎕USING,←⊂'System.Windows,WPF/PresentationFramework.dll'
[5]    ⎕USING,←⊂'System.Windows,WPF/PresentationCore.dll'
```

`TempConverter[8-9]` creates a Window and sets its SizeToContent and Title properties as in the XAML example. Notice however that whereas using XAML the string `SizeToContent="WidthandHeight"` is sufficient, when using code it is necessary to get the *Type* right. In this case, the SizeToContent property must be set to a specific member (in this case WidthAndHeight) of the System.Windows.SizeToContent enumeration. Other members of this Type are Width, Height and Manual (the default).

```
[7]    win←⎕NEW Window
[8]    win.SizeToContent←SizeToContent.WidthAndHeight
[9]    win.Title←'WPF Temperature Converter'
```

`TempConverter[11-12]` create a DockPanel control and set its LastChildFill property to 0. In this case the APL value 0 is used instead of the string "False" in XAML.

```
[11]   dp←⎕NEW DockPanel
[12]   dp.LastChildFill←0
```

`TempConverter[14]` creates a Menu control.

```
[14]   mnu←⎕NEW Menu
```

`TempConverter[16-18]` create a MenuItem control with the caption *Scale*, and then add the control to the Items collection of the main Menu using its Add method. This illustrates one significant difference between using XAML and code. In XAML, the parent/child relationships between controls are defined by the structure and order of the XML. Using code, child controls must be explicitly added to the appropriate list of child controls managed by the parent.

```
[16]   mnuScale←⎕NEW MenuItem
[17]   mnuScale.Header←'_Scale'
[18]   sink←mnu.Items.Add mnuScale
```

`TempConverter[20-25]` create a MenuItem control labelled *Fahrenheit*. The IsCheckable and IsChecked properties are set to 1, which is equivalent to "True" in XAML. The callback function `SET_F` is assigned to the Click event exactly as in the XAML version of this example. The last line in this section makes the *Fahrenheit* MenuItem a child of the *Scale* MenuItem.

```
[20]    mnuFahrenheit←⎕NEW MenuItem
[21]    mnuFahrenheit.Header←'Fahrenheit'
[22]    mnuFahrenheit.IsCheckable←1
[23]    mnuFahrenheit.IsChecked←1
[24]    mnuFahrenheit.onClick←'SET_F'
[25]    sink←mnuScale.Items.Add mnuFahrenheit
```

The code used to create the *Centigrade* MenuItem is more or less the same.

`TempConverter[34-35]` adds the top-level Menu to the DockPanel. Note that in the case of a DockPanel, the list of its child controls is represented by its Children property. Furthermore, to define how it is docked this is done, using code, by the SetDock method of the DockPanel. This contrasts with the way this is achieved using XAML (`DockPanel.Dock="Top"`). Note too that the argument to SetDock is not just a simple string as in XAML, but a member of the System.Windows.Controls.Dock enumeration.

```
[34]    sink←dp.Children.Add mnu
[35]    dp.SetDock mnu Dock.Top
```

`TempConverter[37-39]` create the Grid control. Its Width property will accept a simple numeric value, but its Margin property must be given an instance of a System.Windows.Thickness structure. In this case, the ThickNess constructor is given a 4-element numeric vector that specifies its Left, Top, Right and Bottom members respectively.

```
[37]    gr←⎕NEW Grid
[38]    gr.Width←230
[39]    gr.Margin←⎕NEW Thickness(40 10 10 10)
```

`TempConverter[41-47]` create instances of 3 RowDefinition classes and add them to the RowDefinitions collection of the Grid. Note that whereas in XAML the Height can be specified as a string, using code it is necessary once again to use the correct Type. In this case, Height must be specified by a member of the System.Windows.GridLength structure.

```
[41]    rd1←⍞NEW RowDefinition
[42]    rd1.Height←GridLength.Auto
[43]    rd2←⍞NEW RowDefinition
[44]    rd2.Height←GridLength.Auto
[45]    rd3←⍞NEW RowDefinition
[46]    rd3.Height←GridLength.Auto
[47]    gr.RowDefinitions.Add¨rd1 rd2 rd3
```

Similarly, `TempConverter[49-55]` create instances of 3 ColumnDefinition classes and add them to the ColumnDefinitions collection of the Grid. Note that The Width property will not accept a simple numeric value, it must be a member of the GridLength structure. To set the Width to 80, it is necessary first to create an instance of a GridLength structure giving this value as the argument to its constructor.

```
[49]    rc1←⍞NEW ColumnDefinition
[50]    rc1.Width←GridLength.Auto
[51]    rc2←⍞NEW ColumnDefinition
[52]    rc2.Width←⍞NEW GridLength 80
[53]    rc3←⍞NEW ColumnDefinition
[54]    rc3.Width←⍞NEW GridLength 60
[55]    gr.ColumnDefinitions.Add¨rc1 rc2 rc3
```

`TempConverter[57-61]` create a Label control with the caption *Fahrenheit*. To display the Label in a Grid it is necessary to first add it to the Children collection of the Grid, and then set its position in the Grid using its SetRow and SetColumn methods. Similar code is used to create and position the second Label.

```
[57]    l1←⍞NEW Label
[58]    l1.Content←'Fahrenheit'
[59]    sink←gr.Children.Add l1
[60]    gr.SetRow l1 0
[61]    gr.SetColumn l1 0
```

`TempConverter[69-73]` create and position a TextBox control, in the same way as the Label controls. Notice that in this case, the constructor for the Thickness structure is given a single value that specifies all four of its Left, Top, Right and Bottom members.

```
[69]    txtFahrenheit←⍞NEW TextBox
[70]    txtFahrenheit.Margin←⍞NEW Thickness 5
[71]    sink←gr.Children.Add txtFahrenheit
[72]    gr.SetRow txtFahrenheit 0
[73]    gr.SetColumn txtFahrenheit 1
```

`TempConverter[81-87]` create and position a Button control. The callback function `f2c` is attached to the Click event in the same way as in the XAML version of this example.

```
[81]   btnF2C←⎕NEW Button
[82]   btnF2C.Content←'F>C'
[83]   btnF2C.Margin←⎕NEW Thickness 5
[84]   btnF2C.onClick←'f2c'
[85]   sink←gr.Children.Add btnF2C
[86]   gr.SetRow btnF2C 0
[87]   gr.SetColumn btnF2C 2
```

`TempConverter[105]` adds the Grid to the list of Children to be managed by the DockControl.

```
[105]  sink←dp.Children.Add gr
```

`TempConverter[107-112]` create a ScrollBar control. Its Width, Minimum and Maximum properties all accept simple numeric values. However, its Orientation property must be set to a member of the System.Windows.Controls.Orientation enumeration.

```
[107]  scrTemp←⎕NEW ScrollBar
[108]  scrTemp.Width←20
[109]  scrTemp.Orientation←Orientation.Vertical
[110]  scrTemp.Minimum←1
[111]  scrTemp.Maximum←213
[112]  scrTemp.onScroll←'F2C'
```

`TempConverter[114-115]` add the ScrollBar to the list of Children managed by the DockPanel, and use its SetDock method to cause it to be right-aligned.

```
[114]  sink←dp.Children.Add scrTemp
[115]  dp.SetDock scrTemp Dock.Right
```

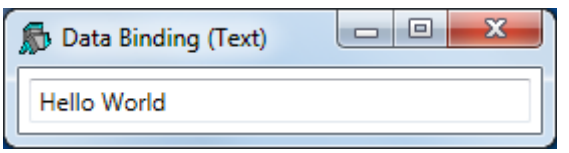
Finally, the DockPanel is assigned to the Content property of the Window, and the Window displayed as in the XAML version of this example. Note that a Window may contain just one control.

```
[117]  win.Content←dp
[118]
[119]  sink←win.ShowDialog
```

4.3 Data Binding

4.3.1 Example 1

This example illustrates data binding using XAML to specify the user-interface coupled with an APL function to drive it and handle the data binding.



The XAML

The XAML shown below, describes a Window containing a TextBox.

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Name="Temp"
 Title="Data Binding (Text)"
 SizeToContent="WidthandHeight">
     <TextBox Name="txt" Width="300" Margin="5"
      Text="{Binding txtSource,Mode=TwoWay,
             UpdateSourceTrigger=PropertyChanged}"/>
</Window>
```

It contains a data binding expression, namely:

```
      Text="{Binding txtSource,Mode=TwoWay,
             UpdateSourceTrigger=PropertyChanged}"
```

This specifies that the Text property of the TextBox is bound to a value in the Binding Source (which has yet to be defined) whose path is `txtSource`. The binding mode is set to `TwoWay` which means that any change in the TextBox will be reflected in a new value in the Binding Source, and vice-versa. The value in the Binding Source will be updated when the property (in this case the Text Property) changes.

The APL Code

The function `Text` which generates this example is shown below.

The argument `txt` is the text to be displayed initially in the TextBox. Note that the variable `XAML_Text` contains the XAML that describes the user-interface listed above.

```
     ∇ Text txt;⎕USING;str;xml;win
[1]    ⎕USING←,⊂'System.Windows.Controls,WPF/PresentationFramework.dll'
[2]    win←LoadXAML XAML
[3]    win.txtBox←win.FindName⊂'txt'
[4]
[5]    ⎕EX'txtSource'
[6]    txtSource←txt
[7]    win.txtBox.DataContext←2015⌶'txtSource'
[8]
[9]    win.Show
     ∇
```

The utility function `LoadXAML` incorporates the 3 lines of code, used to create a WPF window from XAML, that were coded in-line in previous examples in this chapter.

```
     ∇ win←LoadXAML xaml;⎕USING;str;xml
[1]    ⎕USING←'System.IO'
[2]    ⎕USING,←⊂'System.Windows.Markup'
[3]    ⎕USING,←⊂'System.Xml,system.xml.dll'
[4]    ⎕USING,←⊂'System.Windows.Controls,
                 WPF/PresentationFramework.dll'
[5]    str←⎕NEW StringReader(⊂xaml)
[6]    xml←⎕NEW XmlTextReader str
[7]    win←XamlReader.Load xml
     ∇
```

`Text[1]` defines the .NET search path needed to access the WPF controls.

```
[1]    ⎕USING←,⊂'System.Windows.Controls,WPF/PresentationFramework.dll'
```

`Text[2-3]` uses the utility function `LoadXAML` to load a WPF user-interface from the XAML and then uses the FindName method to obtain a reference to the object named *txt*.

```
[2]    win←LoadXAML XAML
[3]    win.txtBox←win.FindName⊂'txt'
```

`Text[5-6]` initialise a new global variable named `txtSource` to the value of the argument. When using a *global* variable as a data binding source, it is generally advisable to establish a new variable by first expunging it.[5]

```
[5]   ⎕EX'txtSource'
[6]   txtSource←txt
```

`Text[7]`creates a Binding Source object using `2015⌶` and assigns it to the DataContext property of the TextBox object. Because it is a character vector, the exported Type for the bound variable `txtSource` is System.String which is appropriate for the Text property of a TextBox.

```
[7]    win.txtBox.DataContext←2015⌶'txtSource'
```

`Text[9]` displays the Window. Note that although the APL local variable `win` goes out of scope when the function terminates, the Window remains visible until the user has closed it.

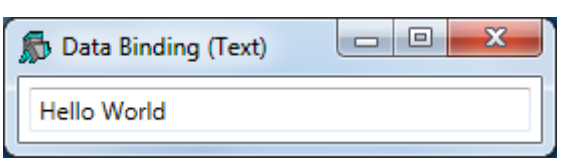
```
[9]   win.Show
```

Testing the Data Binding

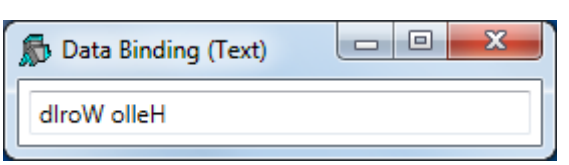
The following expressions may be used to explore the effect of data binding.

```
      )LOAD wpfintro
      )CS DataBinding.Text
```

```
      Text 'Hello World'
```

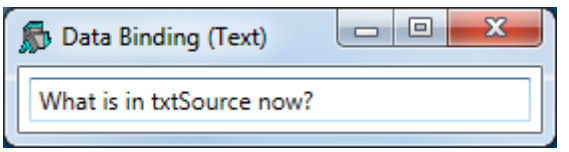



```
     txtSource←⌽txtSource
```



5 This is because its binding type (the exported type of the data bound variable) is stored in the workspace along with its value, and the binding type (were it to be incorrect) may not be changed once it has been established.

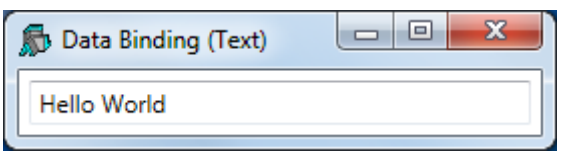
Typing into the TextBox changes the value of the bound variable.




```
      txtSource
What is in txtSource now?
```

4.3.2 Example 2

This example illustrates the use of the optional left argument to `2015I` to specify the data type used to export the value of the bound variable.



The XAML

The XAML shown below, describes the same Window containing a TextBox as before.

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Name="Temp"
 Title="Data Binding (FontSize)"
 SizeToContent="WidthandHeight">
     <TextBox Name="txt" Text="Hello World" Width="300"
      Margin="5"
      FontSize="{Binding sizeSource,Mode=OneWay}"/>
</Window>
```

This time, the data binding expression is:

```
      FontSize="{Binding sizeSource,Mode=OneWay}"/>
```

This specifies that the FontSize property of the TextBox is bound to a value in the Binding Source (which has yet to be defined) whose path is `sizeSource`. The binding mode is set to `OneWay` which means that the FontSize property depends on the data value but not vice versa. Were the FontSize to change for any external reason (which is

admittedly unlikely in the case of FontSize), it would not alter the value in `sizeSource` to which it is bound.

The APL Code

The function `FontSize` is almost identical to the function `Text` which is described in Example 1.

```
     ∇ FontSize size;⎕USING;win
[1]    ⎕USING←'System'
[2]    ⎕USING,←⊂'System.Windows.Controls,WPF/PresentationFramework.dll'
[3]    win←LoadXAML XAML
[4]    win.txtBox←win.FindName⊂'txt'
[5]
[6]    ⎕EX'sizeSource'
[7]    sizeSource←size
[8]    win.txtBox.DataContext←Int32(2015I)'sizeSource'
[9]
[10]   win.Show
     ∇
```

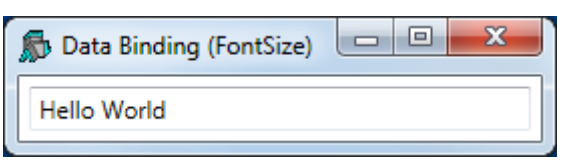
The key difference is in `FontSize[8]`. Here the left argument of `(2015I)` is `Int32`. This means that the exported Type of the variable `sizeSource` will be Int32. This Type (a 32-bit integer) is required by the FontSize property of a TextBox; no other Type will do. If this were omitted, APL would export the value of the variable using a Type dependent on its internal format (most likely Int16) and the binding would fail.

```
[8]    win.txtBox.DataContext←Int32(2015I)'sizeSource'
```

Testing the Data Binding

```
      )LOAD wpfintro
      )CS DataBinding.FontSize
```

```
      FontSize 12
```



```
      sizeSource
12
      sizeSource←30
```

4.3.3 Example 3

This example uses APL code to both build the user-interface (instead of using XAML) and handle the data binding. In this case both the Text and the FontSize properties are bound to APL variables. The function is shown below:

```
     ∇ TextFontSize(txt size);⎕USING;win;sink
[1]    ⎕USING←'System'
[2]    ⎕USING,←,⊂'System.Windows.Controls,WPF/PresentationFramework.dll'
[3]    ⎕USING,←⊂'System.Windows.Controls.Primitives,WPF/
PresentationFramework.dll'
[4]    ⎕USING,←⊂'System.Windows,WPF/PresentationFramework.dll'
[5]    ⎕USING,←⊂'System.Windows,WPF/PresentationCore.dll'
[6]
[7]   ⍝ Create a Window, DockPanel and TextBox
[8]    win←⎕NEW Window
[9]    win.SizeToContent←SizeToContent.WidthAndHeight
[10]   win.Title←'Data Binding (Text and FontSize)'
[11]   win.txtBox←⎕NEW TextBox
[12]   win.txtBox.Width←350
[13]   win.Content←win.txtBox
[14]
[15]  ⍝ Define data binding from variable "txtSource"
[16]  ⍝ to the Text property of TextBox win.txtBox
[17]   ⎕EX'txtSource'
[18]   txtSource←txt
[19]   win.txtbinding←⎕NEW Data.Binding(⊂'txtSource')
[20]   win.txtbinding.Source←2015⌶'txtSource'
[21]   win.txtbinding.Mode←Data.BindingMode.TwoWay
[22]   win.txtbinding.UpdateSourceTrigger←Data.UpdateSourceTrigger.Prope
rtyChanged
[23]   sink←win.txtBox.SetBinding TextBox.TextProperty win.txtbinding
[24]
[25]  ⍝ Define data binding from variable "sizeSource"
[26]  ⍝ to the FontSize property of TextBox win.txtBox
[27]   ⎕EX'sizeSource'
[28]   sizeSource←size
[29]   win.fntbinding←⎕NEW Data.Binding(⊂'sizeSource')
[30]   win.fntbinding.Source←Int32(2015⌶)'sizeSource'
[31]   win.fntbinding.Mode←Data.BindingMode.OneWay
[32]   sink←win.txtBox.SetBinding TextBox.FontSizeProperty win.fntbindin
g
[33]
[34]   win.Show
     ∇
```

Apart from the code that creates the controls, the only material difference between this and the previous examples is the way that the bindings are handled.

In code (as opposed to using XAML) this is done using explicit Binding objects[6] The code for binding the Text property to the `txtSource` variable is as follows:

```
[19]   win.txtbinding←⍞NEW Data.Binding(⊂'txtSource')
[20]   win.txtbinding.Source←2015⌶'txtSource'
[21]   win.txtbinding.Mode←Data.BindingMode.TwoWay
[22]   win.txtbinding.UpdateSourceTrigger←Data.UpdateSourceTrigger.Prope
rtyChanged
[23]   sink←win.txtBox.SetBinding TextBox.TextProperty win.txtbinding
```

Line [19] creates a Binding object, passing the constructor the name of the APL variable `txtSource` as the Path to the binding value.

```
[19]   win.txtbinding←⍞NEW Data.Binding(⊂'txtSource')
```

Line [20] creates a Binding Source object using `2015⌶` as before, but this time assigns it to the Source property of the Binding object.

```
[20]   win.txtbinding.Source←2015⌶'txtSource'
```

Line [21] sets the Mode property of the Binding object to TwoWay (a field of the BindingMode Type). As in Example 1, this specifies two-way binding.

```
[21]   win.txtbinding.Mode←Data.BindingMode.TwoWay
```

Line [22] sets the UpdateSourceTrigger property of the Binding object to PropertyChanged (a field of the UpdateSourceTrigger Type). This causes the value in the Binding Source (in this case `txtSource`) to be changed whenever the property (in this case the Text property) of the TextBox changes. This will occur on every keystroke.

```
[22]
win.txtbinding.UpdateSourceTrigger←Data.UpdateSourceTrigger.PropertyChan
ged
```

(Note that the three types Binding, BindingMode and UpdateSourceTrigger are located in System.Windows.Data)

The code that establishes the binding between the `sizeSource` variable and the FontSize property is very similar.

6 Binding objects are implicit in all binding operations, but are created declaratively when using XAML.

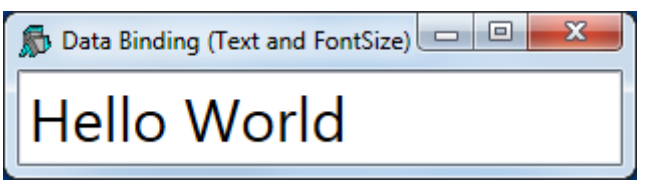
```
[29]   win.fntbinding←⍞NEW Data.Binding(⊂'sizeSource')
[30]   win.fntbinding.Source←Int32(2015⌶)'sizeSource'
[31]   win.fntbinding.Mode←Data.BindingMode.OneWay
[32]   sink←win.txtBox.SetBinding
               TextBox.FontSizeProperty win.fntbinding
```

Note however that (as in Example 2) the left-argument to `(2015⌶)` specifies that the exported data type of the `sizeSource` variable is to be Int32.

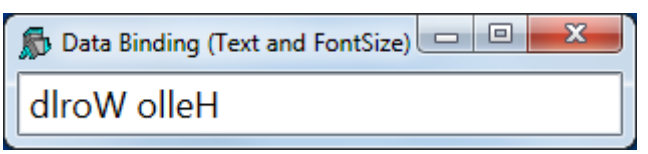
Testing the Data Binding

```
      )LOAD wpfintro
      )CS DataBinding.TextFontSizeCode
```

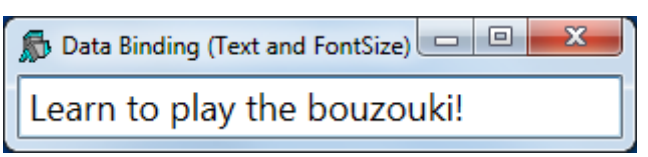
```
      TextFontSize 'Hello World' 30
```




```
      txtSource sizeSource←(⍎txtSource) 18
```



As in previous examples, when the user changes the text, the new text appears in `txtSource`.




```
      txtSource
Learn to play the bouzouki!
```

> **Note**
>
> It is perhaps worth mentioning that if you want to bind two properties *of the same object* to two APL variables, it has to be done by writing code as shown in this example, using two separate Binding Source objects. This is because using XAML you may only associate a single Binding Source to an object.
>
> However, this minor restriction is easily surmounted by using an APL namespace as a Binding Source as illustrated in the next Example.

4.3.4 Example 4

This example uses XAML to specify the user-interface and the main components of the data binding.

The XAML

The XAML is much the same as in Example 1 and 2 except that it connects two properties Text and FontSize of the same TextBox to two Paths *txtSource* and *sizeSource*.

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Name="Temp"
 Title="Data Binding (Text and FontSize)"
 SizeToContent="WidthandHeight">
     <TextBox Name="txt" Width="350" Margin="5"
      Text="{Binding txtSource,Mode=TwoWay,
             UpdateSourceTrigger=PropertyChanged}"
      FontSize="{Binding sizeSource,Mode=OneWay}"/>
</Window>
```

The APL Code

The function `TextFontSize` is shown below.

```
     ∇ TextFontSize(txt size);⎕USING;str;xml;win;options
[1]    ⎕USING←'System'
[2]    ⎕USING,←⊂'System.Windows,WPF/PresentationFramework.dll'
[3]
[4]    win←LoadXAML XAML
[5]
[6]    src←⎕NS''
[7]    src.(txtSource sizeSource)←txt size
[8]    options←2 2⍴'txtSource'String'sizeSource'Int32
[9]
[10]   win.DataContext←options(2015⌶)'src'
[11]
[12]   win.Show
     ∇
```

Lines [6-7] create a new namespace src containing two variables `txtSource` and `sizeSource` which are initialised to the arguments of the function.

```
[6]    src←⎕NS''
[7]    src.(txtSource sizeSource)←txt size
```

Line [8] creates a local variable named options which will be used as the left argument of `2015⌶)`. It is a 2-column matrix. The first column is a list of the names of the variables which are to be exported by the namespace when used as a Binding Source. The second column specifies their data types.

```
[8]    options←2 2⍴'txtSource'String'sizeSource'Int32
```

Line [10] creates a Binding Source object from the namespace `src` and a left argument `options` and assigns it to the DataContext property of the Window `win`.

```
[10]   win.DataContext←options(2015⌶)'src'
```

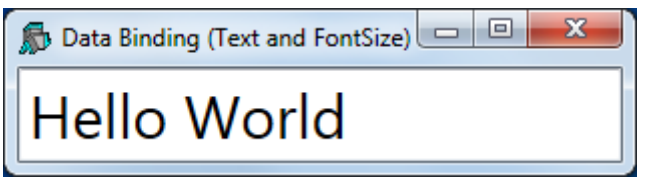
An alternative would be to assign it to the DataContext property of the TextBox object, but this would require one further line of code to identify it. The reason this works is that the DataContext property of a TextBox (and many other controls) is inherited from its parent Window. This feature allows a single Binding Source namespace to be used to specify data bindings between its component variables and any number of properties of any number of controls in the same Window.

As shown before, the left argument of `2015⌶)` is optional. Without it, the namespace would export all its variables using default binding types. In this case, because the binding type of `sizeSource` must be specified as Int32, it is necessary to use a left argument, which means specifying all the variables involved.

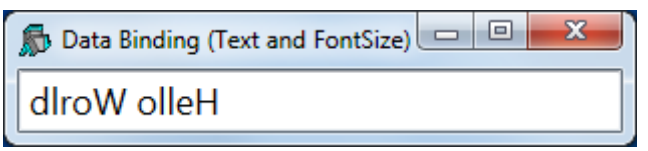
Testing the Data Binding

```
      )LOAD wpfintro
      )CS DataBinding.TextFontSizeXAML
```

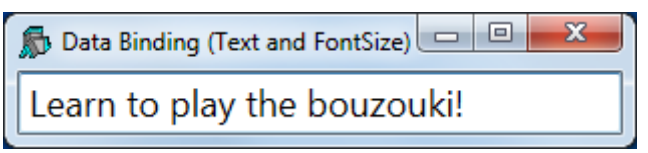
```
      DB_Text_FontSize_XAML'Hello World' 30
```




```
      src.(txtSource sizeSource←(⍎txtSource) 18)
```



As in previous examples, when the user changes the text, the new text appears in `txtSource`.




```
      src.txtSource
Learn to play the bouzouki!
```

4.3.5 Example 5

WPF data binding provides the means to bind controls that display lists of items, such as the ListBox, ListView, and TreeView controls, to collections of data. These controls are all based upon the ItemsControl class. To bind an ItemsControl to a collection object, you use its ItemsSource property.

If the right argument of `2015⌶` names a variable, or a namespace containing a variable, that is a vector other than a simple character vector, it returns a Binding Source object that provides the necessary interfaces to bind the variable as a collection to the ItemSource property of an ItemsControl.

The APL variable will normally contain a vector of character vectors, because most ItemsControl objects deal with collections of strings. However, any APL vector other than a simple character vector will be treated in this way.

This example illustrates binding between a variable containing a vector of character vectors, to the items of a ListBox.

Incidentally, the ItemsSource property overrides the Items collection as a means to specify the content of the ItemsControl. When the ItemsSource property is set, the Items collection becomes read-only and of fixed-size. Note that the ItemsSource property supports OneWay binding by default.

The XAML

The variable `XAML_FilteredList`, shown below, contains XAML to specify a Window containing a StackPanel. The StackPanel control is a WPF layout control that organises child controls in a single line, by default vertically. In this example, the StackPanel contains a TextBox and, below it, a WrapPanel, and below that a TextBlock. The WrapPanel is also a layout control that organises its child controls sequentially from left to right. The WrapPanel contains two ListBox controls.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Filtered List Example"
    SizeToContent="WidthAndHeight"
    Topmost="true">
    <StackPanel>
        <TextBox Name="filter" Margin="5"
         Text="{Binding Filter,Mode=TwoWay,
                UpdateSourceTrigger=PropertyChanged}"/>
        <WrapPanel>
            <ListBox Name="all" Width="135" Height="440"
             Margin="5" ItemsSource="{Binding DyalogNames}"/>
            <ListBox Name="filtered" Width="135" Height="440"
             Margin="5" ItemsSource="{Binding FilteredList}"/>
        </WrapPanel>
        <TextBlock Text="Dyalog WPF Demo" Margin="5"/>
    </StackPanel>
</Window>
```

The Code

```
     ∇ FilteredList;MySource;win;sink
[1]
[2]    MySource←⎕NS''
[3]    MySource.Filter←''
[4]    MySource.FilteredList←0⍴⊂''
[5]    MySource.DyalogNames←DyalogNames
[6]
[7]    win←LoadXAML XAML_FilteredList
[8]    win.DataContext←2015⌶'MySource'
[9]    (win.FindName⊂'filter').onTextChanged←
                              'FilteredList_TextChanged'
[10]   sink←win.ShowDialog
     ∇
```

Like the previous example, this example uses a namespace `MySource` containing the bound variables `Filter`, `FilteredList` and `DyalogNames`.

`FilteredList[8]` creates a Binding Source object and assigns it to the DataContext property of the Window `win`.

```
[8]    win.DataContext←2015⌶'MySource'
```

The DataContext property is inherited by all child controls, so they all share the same Binding Source. Their different Paths to different values in the Binding Source are specified in the XAML as follows.

The Text property of the TextBox named *filter* is bound to the variable `Filter` by the expression `Text="{Binding Filter,...`

```
        <TextBox Name="filter" Margin="5"
         Text="{Binding Filter,Mode=TwoWay,
```

The ItemsSource property of the ListBox named *all* is bound to the variable `DyalogNames` by the expression `ItemsSource="{Binding DyalogNames}"`

```
            <ListBox Name="all" Width="135" Height="440"
             Margin="5" ItemsSource="{Binding DyalogNames}"/>
```

Thirdly, the ItemsSource property of the ListBox named *filtered* is bound to the variable `FilteredList` by the expression `ItemsSource="{Binding FilteredList}"`

```
            <ListBox Name="filtered" Width="135" Height="440"
             Margin="5" ItemsSource="{Binding FilteredList}"/>
```

Testing the Data Binding

```
      FilteredList
```

If the user types a single character, in this case "e", into the TextBox, this fires a TextChanged event which in turn fires the callback function shown below:

```
     ∇ FilteredList_TextChanged a;hits
[1]    hits←(⊂MySource.Filter){⍵/⍨⍺∊⍵}¨DyalogNames
[2]    MySource.FilteredList←hits/DyalogNames
     ∇
```

When the callback runs, the variable `MySource.Filter`, which is bound to the Text property of the TextBox, will contain "e". The function calculates a mask `hits` which identifies which members of the variable `DyalogNames` contain this string. It then assigns that subset to the variable `MySource.FilteredList`. This is bound to the ItemsSource property of the right-hand ListBox, so the result is as follows:

Similarly, typing "er" into the TextBox reduces the number of hits as shown below:

4.3.6 Example 6

This example illustrates data binding using a vector of .NET objects, in this case DateTime objects.

The XAML

The XAML shown below, describes a Window containing a StackPanel, inside which is a ListBox.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="NetObjects (DateTime) Example"
    SizeToContent="WidthAndHeight" >
    <StackPanel>
         <TextBlock Text="Dates of forthcoming Orthodox Easters"
          FontSize="18" Margin="5"/>
         <ListBox Name="EasterDates" Height="100"
          Margin="5" />
    </StackPanel>
</Window>
```

The APL Code

The function `NetObjects` is shown below.

```
     ∇ NetObjects;⎕USING;win;dt
[1]    ⎕USING←'System'
[2]    win←LoadXAML XAML
[3]    win.dates←win.FindName⊂'EasterDates'
[4]    dt←{⎕NEW DateTime ⍵}¨Easter
[5]    win.dates.ItemsSource←2015⌶'dt'
[6]    sink←win.ShowDialog
     ∇
```

`NetObjects[3]` uses FindName to obtain a ref to the ListBox (defined in the XAML) named *EasterDates:*

```
[3]    win.dates←win.FindName⊂'EasterDates'
```

The global variable `Easter` contains a vector of 3-element numeric vectors representing the dates of forthcoming Orthodox Easter Sundays.

```
      ↑Easter
2015 4 12
2016 5  1
2017 4 16
2018 4  8
2019 4 28
2020 4 19
2021 5  2
2022 4 24
2023 4 16
2024 5  5
```

`NetObjects[4]` creates a vector of DateTime objects from the global variable `Easter`.

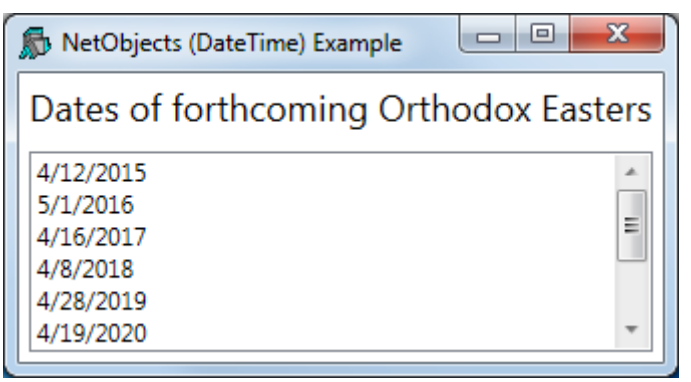
```
[4]     dt←{⎕NEW DateTime ⍵}¨Easter
```

Then, NetObjects[5] creates a binding source object from this array and assigns it to the ItemsSource property of the ListBox.

```
[5]     win.dates.ItemsSource←2015⌶'dt'
```

Testing the Data Binding

```
      )LOAD wpfintro
      DataBinding.NETObjects.NETObjects
```



4.3.7 Example 6a (Casting to DateTime)

This example is similar to Example 6 but illustrates how numeric data in `⎕TS` format can be converted to DateTime type.

The XAML

The XAML shown below describes a Window containing a StackPanel, inside which is a ListBox.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DateTimes using ⎕TS data"
    SizeToContent="WidthAndHeight" >
    <StackPanel>
         <TextBlock Text="Some High Tides at Portsmouth, England"
          FontSize="18" Margin="5"/>
         <ListBox Name="TideTimes" Height="200"
          Margin="5" />
    </StackPanel>
</Window>
```

The APL Code

The function `Tides` is shown below.

```
     ∇ Tides;⎕USING;win;dt;Highs
[1]    ⎕USING←'System'
[2]    win←LoadXAML XAML_Tides
[3]    win.times←win.FindName⊂'TideTimes'
[4]    Highs←(⊂2016 2 18),¨(7 9)(8 44)(19 47)(21 47)
[5]    Highs,←(⊂2016 2 19),¨(8 17)(10 12)(20 51)(22 51)
[6]    dt←7↑¨Highs
[7]    win.times.ItemsSource←DateTime(2015⌶)'dt'
[8]    sink←win.ShowDialog
     ∇
```

`Tides[3]` uses FindName to obtain a ref to the ListBox (defined in the XAML) named *TideTimes*:

```
[3]    win.times←win.FindName⊂'TideTimes'
```

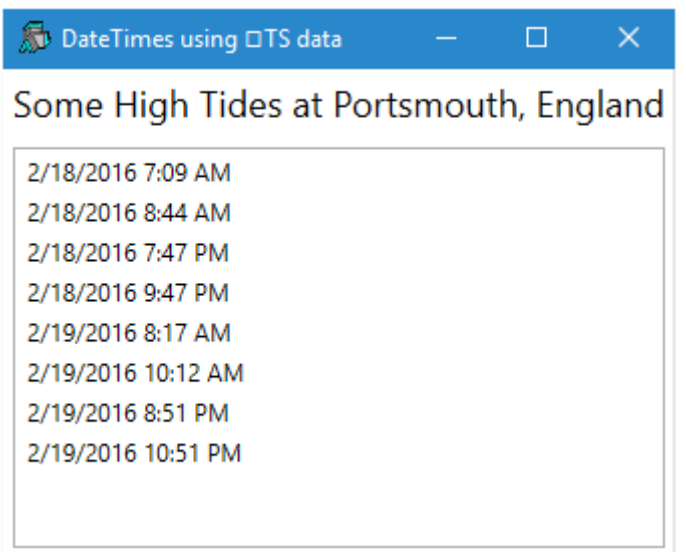
`Tides[4-5]` creates a vector of integer vectors each of which species the time and date of a high tide at Portsmouth. `Tides[6]` extends each to 7-elements, which is required to represent a DateTime object.

Then, `Tides[7]` creates a binding source object from this array and assigns it to the ItemsSource property of the ListBox. Note that the left argument DateTime specifies that the data be cast to that type.

```
[7]    win.times.ItemsSource←DateTime(2015⌶)'dt'
```

Testing the Data Binding

```
      )LOAD wpfintro
      DataBinding.NetObjects.Tides
```



`Tides[3]` uses FindName to obtain a ref to the ListBox (defined in the XAML) named *TideTimes*:

```
[3]    win.times←win.FindName⊂'TideTimes'
```

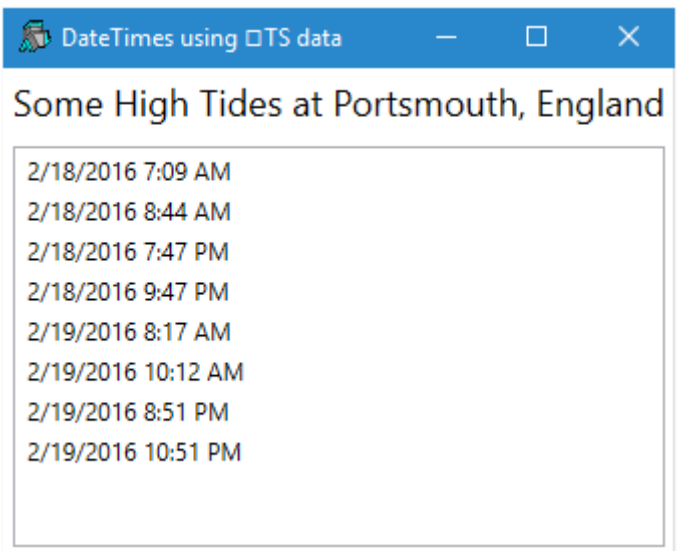
`Tides[4-5]` creates a vector of integer vectors each of which species the time and date of a high tide at Portsmouth. `Tides[6]` extends each to 7-elements, which is required to represent a DateTime object.

Then, `Tides[7]` creates a binding source object from this array and assigns it to the ItemsSource property of the ListBox. Note that the left argument DateTime specifies that the data be cast to that type.

```
[7]    win.times.ItemsSource←DateTime(2015⌶)'dt'
```

Testing the Data Binding

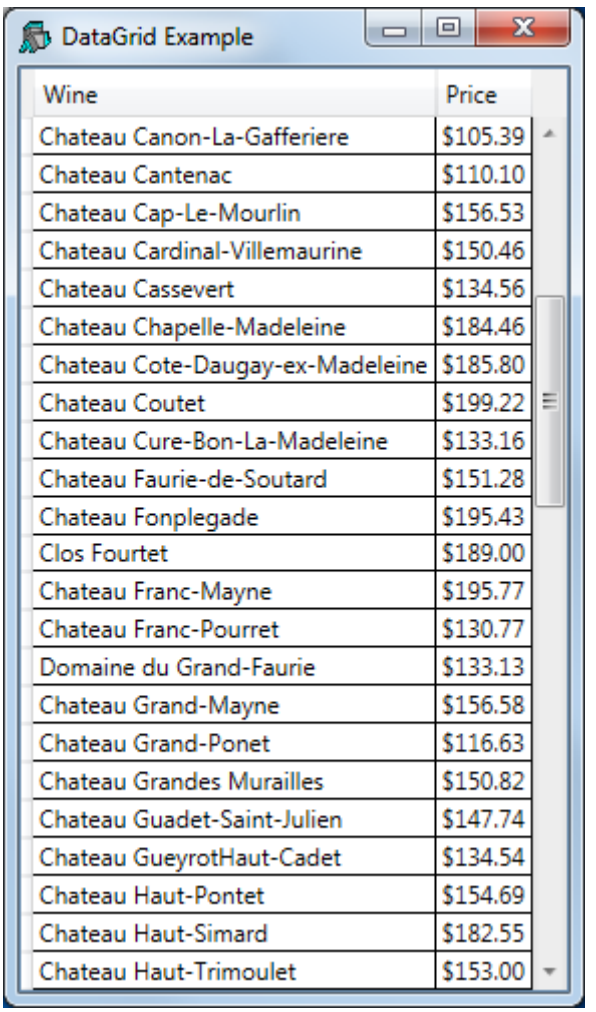
```
      )LOAD wpfintro
      DataBinding.NetObjects.Tides
```



4.3.8 Example 7

This example illustrates data binding using a vector of namespaces.

Each row in the WPF DataGrid control is represented by an object, and each column as a property of that object. Each row in the DataGrid is bound to an object in the data source, and each column in the data grid is bound to a property of the data object.



The XAML

The XAML shown below, describes a Window containing a DockPanel, inside which is a DataGrid.

```
<Window
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="DataGrid Example" Height="500"
   SizeToContent="Width"
   Topmost="true">
   <DockPanel>
       <DataGrid Name="DG1" ItemsSource="{Binding}"
                 AutoGenerateColumns="False" >
           <DataGrid.Columns>
               <DataGridTextColumn Header="Wine"
                Binding="{Binding Name}"/>
               <DataGridTextColumn Header="Price"
                Binding="{Binding Price, StringFormat=C}" />
           </DataGrid.Columns>
       </DataGrid>
   </DockPanel>
</Window>
```

The phrase `ItemsSource="{Binding}"` states that the content of the DataGrid is bound to a data source, which in this case will be inherited from the DataContext property of the parent Window.

`Binding="{Binding Name}"` specifies that the contents of the first column are bound to a Path named *Name* in the data source.

Similarly, `Binding="{Binding Price, StringFormat=C}"` specifies that the Path for the second column is *Price* (`StringFormat=C` merely specifies the default currency format).

The APL Code

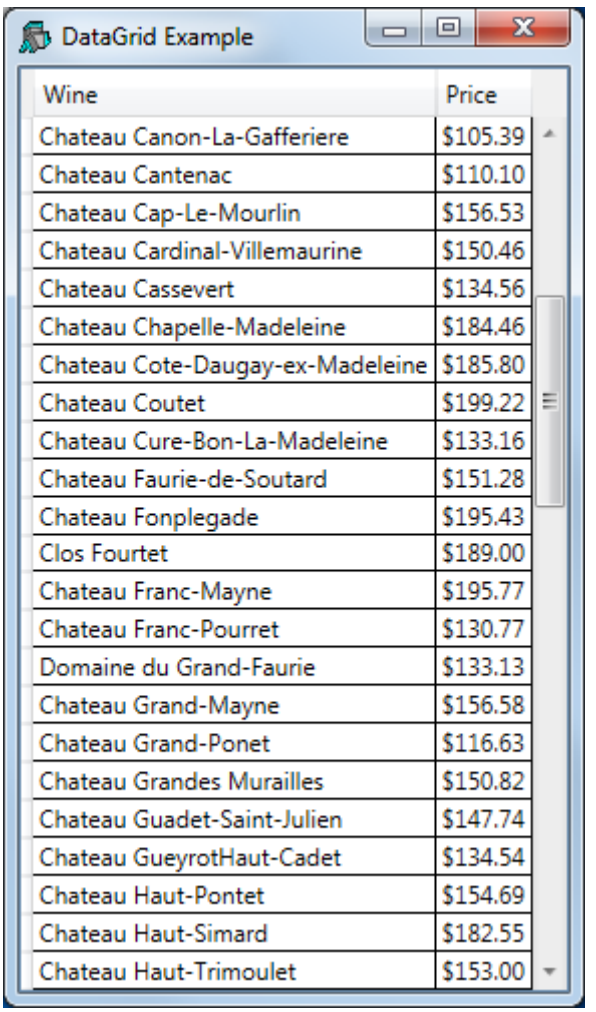
The function `Grid` is shown below.

```
     ∇ Grid;⎕USING;MySource;win
[1]    ⎕USING←'System'
[2]    winelist←⎕NS¨(⍴Wines)⍴⊂''
[3]    winelist.Name←Wines
[4]    winelist.Price←0.01×10000+?(⍴Wines)⍴10000
[5]
[6]    win←LoadXAML XAML
[7]    win.DataContext←2015⌶'winelist'
[8]    win.Show
     ∇
```

The global variable `Wines` contains a vector of character vectors, each of which is the name of a wine. `Grid[2-4]` creates `winelist`, a vector of namespaces, of the same length, each of which contains two variables c `Name` and `Price`.

Testing the Data Binding

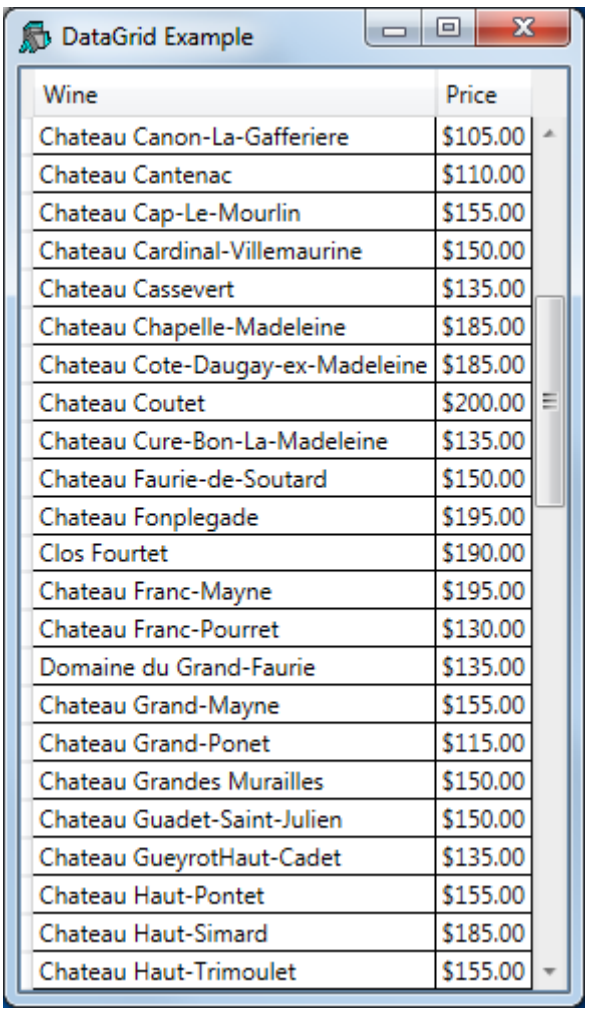
```
      )LOAD wpfintro
      )CS DataBinding.DataGrid
      Grid
```



Wine	Price
Chateau Canon-La-Gafferiere	$105.39
Chateau Cantenac	$110.10
Chateau Cap-Le-Mourlin	$156.53
Chateau Cardinal-Villemaurine	$150.46
Chateau Cassevert	$134.56
Chateau Chapelle-Madeleine	$184.46
Chateau Cote-Daugay-ex-Madeleine	$185.80
Chateau Coutet	$199.22
Chateau Cure-Bon-La-Madeleine	$133.16
Chateau Faurie-de-Soutard	$151.28
Chateau Fonplegade	$195.43
Clos Fourtet	$189.00
Chateau Franc-Mayne	$195.77
Chateau Franc-Pourret	$130.77
Domaine du Grand-Faurie	$133.13
Chateau Grand-Mayne	$156.58
Chateau Grand-Ponet	$116.63
Chateau Grandes Murailles	$150.82
Chateau Guadet-Saint-Julien	$147.74
Chateau GueyrotHaut-Cadet	$134.54
Chateau Haut-Pontet	$154.69
Chateau Haut-Simard	$182.55
Chateau Haut-Trimoulet	$153.00

Let's round the prices to the nearest $5.

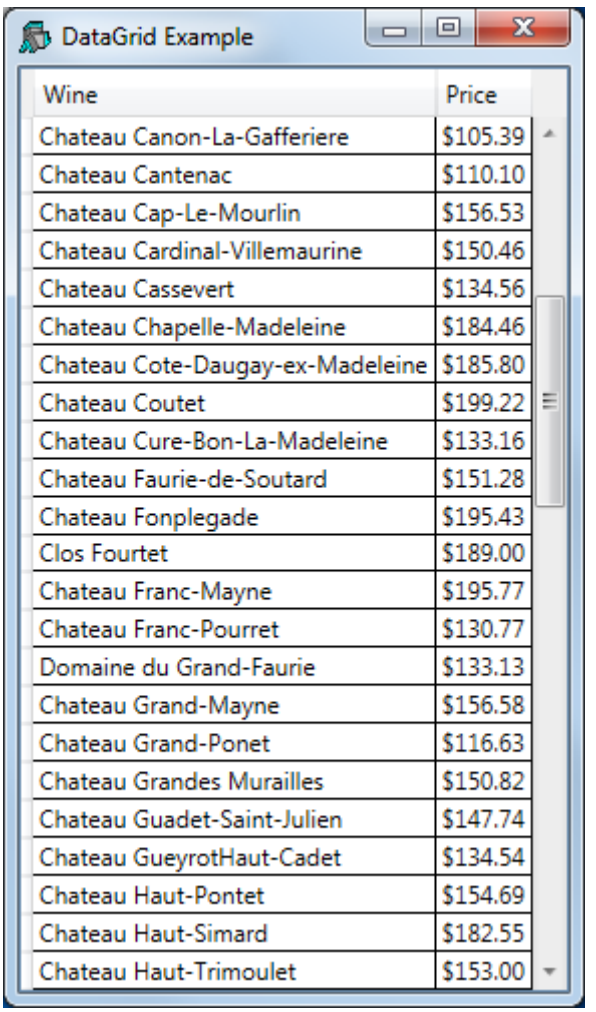
```
winelist.Price←5×⌊0.5+winelist.Price÷5
```



4.3.9 Example 8

This example illustrates data binding using a matrix and is practically identical to Example 7 except that it uses a matrix instead of a vector of namespaces.

Each row in the WPF DataGrid control is represented by an object, and each column as a property of that object. Each row in the DataGrid is bound to an object in the data source, and each column in the data grid is bound to a property of the data object.



The XAML

The XAML shown below, describes a Window containing a DockPanel, inside which is a DataGrid. The XAML is identical to the XAML in Example 7, except for the window caption.

```
<Window
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="DataGrid Matrix Example" Height="500"
   SizeToContent="Width"
   Topmost="true">
   <DockPanel>
       <DataGrid Name="DG1" ItemsSource="{Binding}"
                 AutoGenerateColumns="False" >
           <DataGrid.Columns>
               <DataGridTextColumn Header="Wine"
                Binding="{Binding Name}"/>
               <DataGridTextColumn Header="Price"
                Binding="{Binding Price, StringFormat=C}" />
           </DataGrid.Columns>
       </DataGrid>
   </DockPanel>
</Window>
```

The phrase `ItemsSource="{Binding}"` states that the content of the DataGrid is bound to a data source, which in this case will be inherited from the DataContext property of the parent Window.

`Binding="{Binding Name}"` specifies that the contents of the first column are bound to a Path named *Name* in the data source.

Similarly, `Binding="{Binding Price, StringFormat=C}"` specifies that the Path for the second column is *Price* (the phrase `StringFormat=C` merely specifies the default currency format).

The APL Code

The function `Grid` is shown below.

```
     ∇ Grid;⎕USING;MySource;win;info
[1]    ⎕USING←'System'
[2]    ⎕EX'winelist'
[3]    winelist←Wines,[1.5]0.01×10000+?(⍴Wines)⍴10000
[4]    win←LoadXAML XAML
[5]    info←(⍪'Name' 'Price'),⊂Object
[6]    win.DataContext←info(2015⌶)'winelist'
[7]    win.Show
     ∇
```

As in Example 7, the global variable `Wines` contains a vector of character vectors, each of which is the name of a wine.

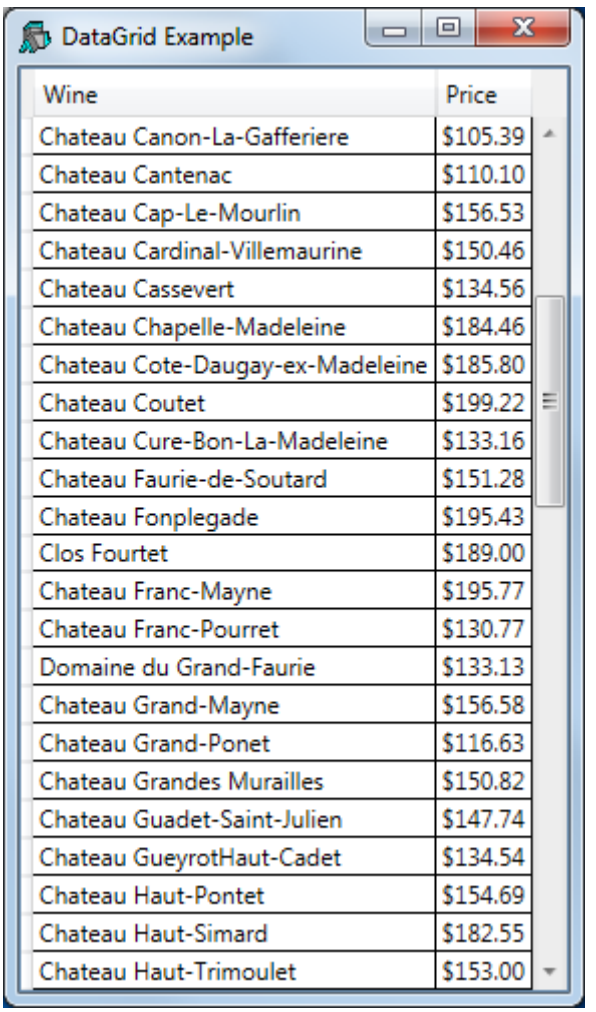
`Grid[2-4]` creates a matrix `winelist`, whose first column contains the names of the wines, and whose second column their (randomly generated) prices. As this is a global variable, the variable is expunged before being used in order to remove any previous data binding information that was associated with it.

`Grid[5]`creates the left argument for `(2015⌶)` which defines the names and data types of the properties which the columns of the matrix `winelist` will be exposed as. In this case, the names of the paths are `Name` and `Price`, and their data types are both `System.Object`. So the first column will be exposed as `Name` and the second as `Price`, matching the path names specified in the XAML:

```
            <DataGridTextColumn Header="Wine"
            Binding="{Binding Name}"/>
            <DataGridTextColumn Header="Price"
            Binding="{Binding Price, StringFormat=C}" />
```

Testing the Data Binding

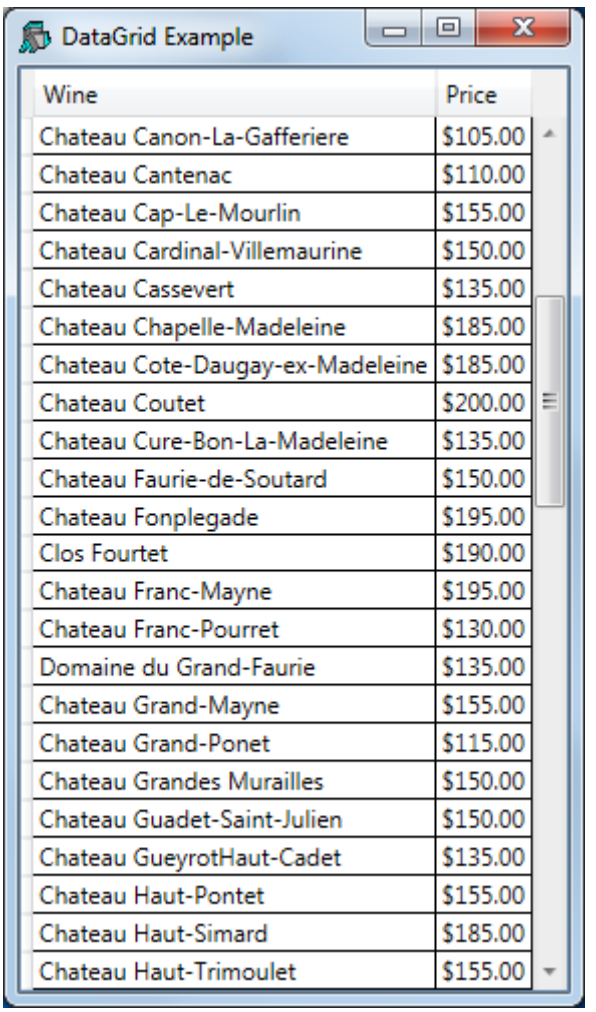
```
      )LOAD wpfintro
      )CS DataBinding.DataGridMatrix
      Grid
```



Wine	Price
Chateau Canon-La-Gafferiere	$105.39
Chateau Cantenac	$110.10
Chateau Cap-Le-Mourlin	$156.53
Chateau Cardinal-Villemaurine	$150.46
Chateau Cassevert	$134.56
Chateau Chapelle-Madeleine	$184.46
Chateau Cote-Daugay-ex-Madeleine	$185.80
Chateau Coutet	$199.22
Chateau Cure-Bon-La-Madeleine	$133.16
Chateau Faurie-de-Soutard	$151.28
Chateau Fonplegade	$195.43
Clos Fourtet	$189.00
Chateau Franc-Mayne	$195.77
Chateau Franc-Pourret	$130.77
Domaine du Grand-Faurie	$133.13
Chateau Grand-Mayne	$156.58
Chateau Grand-Ponet	$116.63
Chateau Grandes Murailles	$150.82
Chateau Guadet-Saint-Julien	$147.74
Chateau GueyrotHaut-Cadet	$134.54
Chateau Haut-Pontet	$154.69
Chateau Haut-Simard	$182.55
Chateau Haut-Trimoulet	$153.00

Let's round the prices to the nearest $5.

```
winelist[;2]←5×⌊0.5+winelist[;2]÷5
```

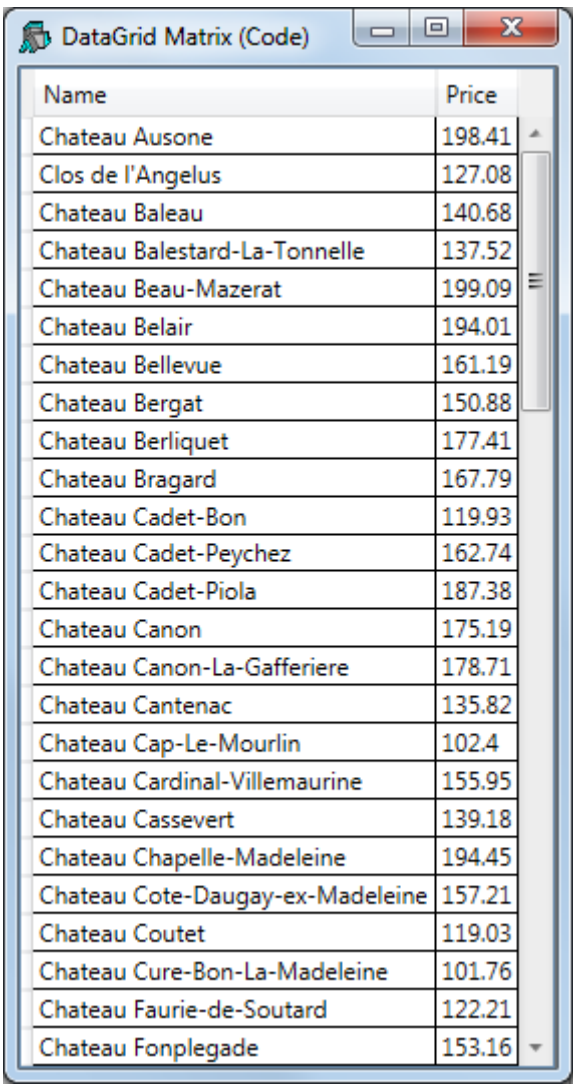


Using Code

The same result can be achieved using code instead of XAML as illustrated by the function `GridCodeNoFmt`. The function is so-named because this code is insufficient to display the second column in currency format.

```
     ∇ GridCodeNoFmt;⎕USING;MySource;win;info;fmt
[1]    ⎕USING←'System'
[2]    ⎕USING,←,⊂'System.Windows.Controls,WPF/PresentationFramework.dll'
[3]    ⎕USING,←⊂'System.Windows.Controls.Primitives,WPF/
PresentationFramework.dll'
[4]    ⎕USING,←⊂'System.Windows,WPF/PresentationFramework.dll'
[5]    ⎕USING,←⊂'System.Windows,WPF/PresentationCore.dll'
[6]
[7]    ⎕EX'winelist'
[8]    winelist←Wines,[1.5]0.01×10000+?(⍴Wines)⍴10000
[9]    win←⎕NEW Window
[10]   win.Title←'DataGrid Matrix (Code)'
[11]   win.grid←⎕NEW DataGrid
[12]   info←(⍪'Name' 'Price'),⊂Object
[13]   win.grid.ItemsSource←info(2015⌶)'winelist'
[14]   win.grid.Height←500
[15]   win.Content←win.grid
[16]   win.SizeToContent←SizeToContent.WidthAndHeight
[17]   win.Show
     ∇
```

This is because by default the DataGrid generates its columns automatically with default formatting.

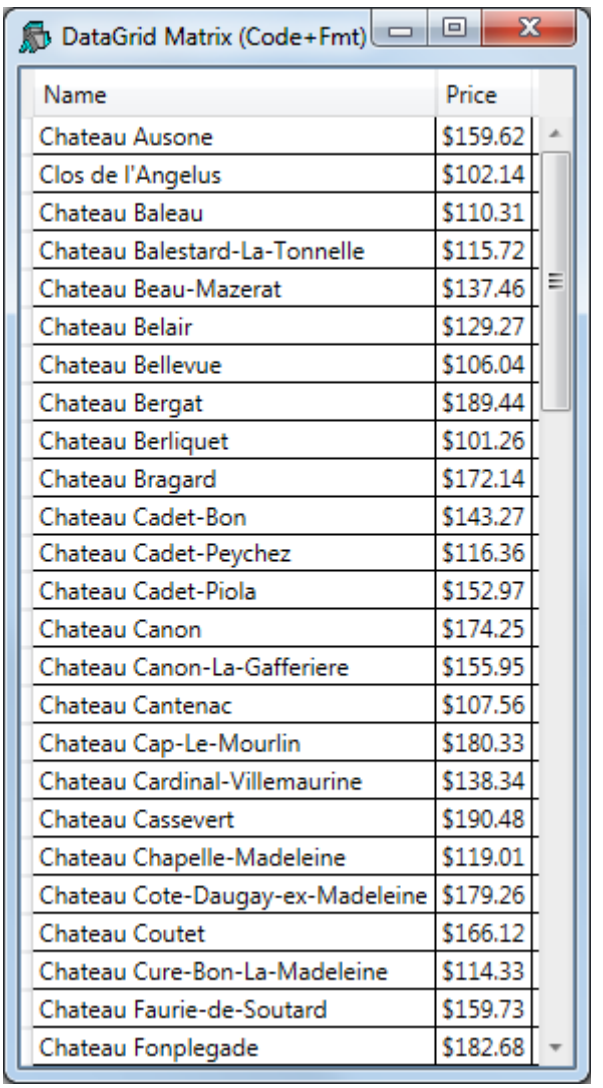


Name	Price
Chateau Ausone	198.41
Clos de l'Angelus	127.08
Chateau Baleau	140.68
Chateau Balestard-La-Tonnelle	137.52
Chateau Beau-Mazerat	199.09
Chateau Belair	194.01
Chateau Bellevue	161.19
Chateau Bergat	150.88
Chateau Berliquet	177.41
Chateau Bragard	167.79
Chateau Cadet-Bon	119.93
Chateau Cadet-Peychez	162.74
Chateau Cadet-Piola	187.38
Chateau Canon	175.19
Chateau Canon-La-Gafferiere	178.71
Chateau Cantenac	135.82
Chateau Cap-Le-Mourlin	102.4
Chateau Cardinal-Villemaurine	155.95
Chateau Cassevert	139.18
Chateau Chapelle-Madeleine	194.45
Chateau Cote-Daugay-ex-Madeleine	157.21
Chateau Coutet	119.03
Chateau Cure-Bon-La-Madeleine	101.76
Chateau Faurie-de-Soutard	122.21
Chateau Fonplegade	153.16

In order to apply special formatting to one or more columns, it is necessary to set the AutoGenerateColumns property to 0, and to generate the columns programmatically as is shown in the second version of the function, `GridCode`.

```
     ∇ GridCode;⎕USING;MySource;win;info;fmt
[1]    ⎕USING←'System'
[2]    ⎕USING,←,⊂'System.Windows.Controls,WPF/PresentationFramework.dll'
[3]    ⎕USING,←⊂'System.Windows.Controls.Primitives,WPF/
PresentationFramework.dll'
[4]    ⎕USING,←⊂'System.Windows,WPF/PresentationFramework.dll'
[5]    ⎕USING,←⊂'System.Windows,WPF/PresentationCore.dll'
[6]
[7]    ⎕EX'winelist'
[8]    winelist←Wines,[1.5]0.01×10000+?(⍴Wines)⍴10000
[9]    win←⎕NEW Window
[10]   win.Title←'DataGrid Matrix (Code with Formatting)'
[11]   win.grid←⎕NEW DataGrid
[12]   info←(⍪'Name' 'Price'),⊂Object
[13]   win.grid.ItemsSource←info(2015⌶)'winelist'
[14]   win.grid.Height←500
[15]   win.grid.AutoGenerateColumns←0
[16]   win.Content←win.grid
[17]   win.SizeToContent←SizeToContent.WidthAndHeight
[18]   ⍝ Add columns and set format
[19]   win.grid.Columns.Add¨''' 'C'{
[20]       col←⎕NEW DataGridTextColumn
[21]       col.Header←⍵
[22]       col.Binding←⎕NEW Data.Binding(⊂⍵)
[23]       col.Binding.StringFormat←,⍺
[24]       col
[25]   }¨'Name' 'Price'
[26]
[27]   win.Show
     ∇
```

In this version of the function, lines `[19-25]` create the two columns `Name` and `Price`, applying currency format to the `Price` column.


DataGrid Matrix (Code+Fmt)
Name
Price
Chateau Ausone
$159.62
Clos de l'Angelus
$102.14
Chateau Baleau
$110.31
Chateau Balestard-La-Tonnelle
$115.72
Chateau Beau-Mazerat
$137.46
Chateau Belair
$129.27
Chateau Bellevue
$106.04
Chateau Bergat
$189.44
Chateau Berliquet
$101.26
Chateau Bragard
$172.14
Chateau Cadet-Bon
$143.27
Chateau Cadet-Peychez
$116.36
Chateau Cadet-Piola
$152.97
Chateau Canon
$174.25
Chateau Canon-La-Gafferiere
$155.95
Chateau Cantenac
$107.56
Chateau Cap-Le-Mourlin
$180.33
Chateau Cardinal-Villemaurine
$138.34
Chateau Cassevert
$190.48
Chateau Chapelle-Madeleine
$119.01
Chateau Cote-Daugay-ex-Madeleine
$179.26
Chateau Coutet
$166.12
Chateau Cure-Bon-La-Madeleine
$114.33
Chateau Faurie-de-Soutard
$159.73
Chateau Fonplegade
$182.68

4.4 Syncfusion Libraries

Dyalog does not include the Syncfusion library of WPF controls. A separate licence is required from Syncfusion to use these in application development and to distribute them with run-time applications.

> **Note**
>
> From Dyalog v20.0 onwards, you must not use the Syncfusion libraries that were distributed with previous versions of Dyalog.

Requirements

To use the Syncfusion libraries you must be using Microsoft .NET Version 4.6.

In addition, to use the controls contained in these assemblies it is necessary to perform one or both of the following steps.

Using XAML

If using XAML, the XAML must include the appropriate `xmlns` statements that specify where the Syncfusion controls are to be found. For example:

```
xmlns:syncfusion="clr-namespace:Syncfusion.Windows.Gauge;
                  assembly=Syncfusion.Gauge.WPF"
```

The above statement defines the prefix `syncfusion` to mean the specified Syncfusion namespace and assembly that contains the various Gauge controls. When the prefix `syncfusion` is subsequently used in front of a control in the XAML, the system knows where to find it. For example:

```
<syncfusion:CircularGauge Name="fahrenheit" Margin="10">
```

□USING

In common with all .NET types, when a Syncfusion control is loaded using XAML or using `□NEW` it is essential that the current value of `□USING` identifies the .NET namespace and assembly in which the control will be found. For example:

```
      ⎕USING,←⊂'Syncfusion.Windows.Gauge,YOUR_INSTALL_PATH/
Syncfusion.Gauge.WPF.dll'
```

This statement tells APL to search the .NET namespace named *Syncfusion.Windows.Gauge*, which is located in the assembly file whose path depends on your specific Syncfusion installation.

4.5 Syncfusion Circular Gauge Example

Dyalog does not include the Syncfusion library of WPF controls. A separate licence is required from Syncfusion to use these in application development and to distribute them with run-time applications.

> **Note**
>
> From Dyalog v20.0 onwards, you must not use the Syncfusion libraries that were distributed with previous versions of Dyalog.


SyncFusion CircularGauge
112
132
92
152
72
172
52
192
Fahrenheit
32
212
50
40
60
30
70
20
80
10
90
Centigrade
0
100

The XAML

Like most Syncfusion controls, the CircularGauge is made up of a complex structure of objects, and the XAML (see variable `XAML_SF`) is too extensive to describe in detail herein. It was created from the sample XAML from the Syncfusion documentation for this control entitled *Essential Gauge for WPF*, which may be downloaded from https://help.syncfusion.com/wpf/gauge.

The key statements in the XAML are as follows:

```
xmlns:syncfusion="clr-
namespace:Syncfusion.Windows.Gauge;assembly=Syncfusion.Gauge.WPF"
```

The above statement defines the prefix `syncfusion` to mean the specified Syncfusion namespace and assembly. When the prefix `syncfusion` is subsequently used in front of a control in the XAML, the system knows where to find it.

The next two statements define CircularPointer controls (the needles on the gauges); one for the Fahrenheit gauge (named *f_pointer*) and one for the Centigrade gauge (named *c_pointer*).

```
<syncfusion:CircularPointer Name="f_pointer" BorderWidth="0.3"
 PointerLength="100" PointerPlacement="Inside" PointerWidth="20"
 Value="32"/>
```

```
<syncfusion:CircularPointer Name="c_pointer" BorderWidth="0.3"
 PointerLength="100" PointerPlacement="Inside" PointerWidth="20"
 Value="0"/
```

The APL Code

The following functions were used to produce the example illustrated above. The main function is `SF_TC_XAML`.

```
     ∇ SF_TC_XAML;⎕USING;win;f_pointer;c_pointer;sink
[1]
[2]    win←LoadXAML XAML_SF
[3]
[4]    f_pointer←win.FindName⊂'f_pointer'
[5]    c_pointer←win.FindName⊂'c_pointer'
[6]
[7]    f_pointer.onMouseEnter←'MouseEnter'
[8]    c_pointer.onMouseEnter←'MouseEnter'
[9]
[10]   sink←win.ShowDialog
     ∇
```

After creating the Window from the text in `XAML_SF`, the function `SF_TC_XAML` obtains refs to the two CircularPointer controls named *f_pointer* (in the Fahrenheit gauge) and *c_pointer* (in the Centigrade gauge). It then attaches the `MouseEnter` callback to each of these objects.

```
     ∇ MouseEnter(this ev);ptrs
[1]    ptrs←f_pointer c_pointer
[2]    ptrs.onValueChanged←(ptrs⍳this)⌽0 'TempChanged'
     ∇
```

In this example, the user grabs one of the gauge needles and moves it around the face of the gauge. When the user moves the mouse into one of these needles, the `MouseEnter` callback fires. The function `MouseEnter` receives the CircularPointer object that generated the event `this` as the first item in its argument.

The code simply attaches the callback function `TempChanged` to `this`, and disables any callback on the other CircularPointer object.

Note that if both CircularPointer objects had callbacks on `TempChanged` at the same time, the system would enter a callback loop.

```
     ∇ TempChanged(obj ev)
[1]    :Select obj
[2]    :Case f_pointer
[3]        c_pointer.Value←(obj.Value-32)×5÷9
[4]    :Case c_pointer
[5]        f_pointer.Value←32+obj.Value÷5÷9
[6]    :EndSelect
     ∇
```

The `LoadXAML` function used in this example is subtly different from previous examples.

```
     ∇ win←LoadXAML xaml;⎕USING;str;xml
[1]    ⎕USING←'System.IO'
[2]    ⎕USING,←⊂'System.Windows.Markup'
[3]    ⎕USING,←⊂'System.Xml,system.xml.dll'
[4]    ⎕USING,←⊂'System.Windows.Controls,WPF/PresentationFramework.dll'
[5]    ⎕USING,←⊂'Syncfusion.Windows.Gauge,YOUR_INSTALL_DIR/
Syncfusion.Gauge.WPF.dll'
[6]    str←⎕NEW StringReader(⊂xaml)
[7]    xml←⎕NEW XmlTextReader str
[8]    win←XamlReader.Load xml
     ∇
```

In particular, it contains the all-important statement:

```
[5]    ⎕USING,←⊂'Syncfusion.Windows.Gauge,
                 YOUR_INSTALL_DIR/Syncfusion.Gauge.WPF.dll'
```

This statement tells APL to search the .NET namespace named *Syncfusion.Windows.Gauge*, which is located in the assembly file whose path depends on your exact Syncfusion installation.

5 APLScript

5.1 Introduction

`APLScript` is a Dyalog scripting language. It was originally designed specifically to program ASP.NET Web Pages and Web Services, but it has been extended to be of more general use outside the Microsoft .NET environment.

`APLScript` is not workspace oriented (although you can call workspaces from it) but is simply a character file containing function bodies and expressions.

`APLScript` files may be viewed and edited using any character-based editor which supports Unicode text files, such as `Notepad`. `APLScript` files may also be edited using Microsoft Word, although they must be saved as text files without any Word formatting.

`APLScript` files employ Unicode encoding so you need a Unicode font with APL symbols, such as APL385 Unicode, to view them. In order to type Dyalog symbols into an `APLScript` file, you also need the Dyalog Input Method Editor (IME), or other APL compatible keyboard.

If you choose to use the Dyalog IME it can be configured from the Dyalog *Configuration* dialog. You may change the associated `.DIN` file and various other options. See *Unicode Input Tab (Unicode Edition Only)*.

There are basically three types of APLScript files that may be identified by three different file extensions. APLScript files with the extension `.aspx` and `.asmx` specify .NET classes that represent ASP.NET Web Pages and Web Services respectively. APLScript files with the extension `.apl` may specify .NET classes or may simply represent an APL application in a script format as opposed to a workspace format. Such applications do not necessarily require the Microsoft .NET Framework.

5.2 The APLScript Compiler

APLScript files are *compiled* into executable code by the APLScript compiler whose name is given in the table below.

	Unicode Edition	Classic Edition
32-Bit	`dyalogc_unicode.exe`	`dyalogc.exe`
64-Bit	`dyalogc64_unicode.exe`	`dyalogc64.exe`

This program is called automatically by `ASP.NET` when a client application requests a Web Page (.aspx) or Web Service (.asmx) and in these circumstances always generates the corresponding .NET class. However, the Script Compiler may also be used to:

- Compile an APLScript into a workspace (`.dws`) that you may subsequently run using `dyalog.exe` or `dyalogrt.exe` in the traditional manner.
- Compile an APLScript into a .NET class (`.dll`) which may subsequently be used by any other .NET compatible host language such as C# or Visual Basic.
- Compile an APLScript into a native Windows executable program (`.exe`), which may be run as a stand-alone executable. This program may be distributed, along with the Dyalog runtime DLL, as a packaged application, and does not require any of the additional support files and registry entries that are typically needed by the Dyalog run-time `dyalogrt.exe`. Note too that the Dyalog dynamic link library does not use MAXWS but instead allocates workspace dynamically as required. See the *Dyalog for Microsoft Windows Installation and Configuration Guide: Run-Time Applications and Components* for further details.
- Compile a Dyalog workspace (.dws) into a native Microsoft Windows executable program, with the same characteristics and advantages described above.

The Script is designed to be run from a command prompt. If in the 64-bit Unicode Edition change to the appropriate directory and type `dyalogc64_unicode /?` (to query its usage) the following output is displayed:

```
c:\Program Files\Dyalog\Dyalog APL-64 18.0 Unicode>dyalogc64_unicode /?
Dyalog APLScript compiler 64 bit. Unicode Mode. Version 18.0.38524.0
Copyright Dyalog Ltd 2000-2020

dyalogc.exe command line options:

/?                Usage
/r:file           Add reference to assembly
/o[ut]:file       Output file name
/res:file         Add resource to output file
/icon:file        File containing main program icon
/q                Operate quietly
/v                Verbose
/s                Treat warnings as errors
/nonet            Creates a binary that does not use Microsoft .Net
/runtime          Build a non-debuggable binary
/lx:expression    Specify entry point (Latent Expression)
/t:library        Build .Net library (.dll)
/t:nativeexe      Build native executable (.exe). Default
/t:workspace      Build dyalog workspace (.dws)
/nomessages       Process does not use windows messages. Use when
creating
                  a process to run under IIS
/console          Creates a console application
/c                Creates a console application
/multihost        Support multi-hosted interpreters
/unicode          Creates an application that runs in a Unicode
intepreter
/wx:[0|1|3]       Sets ⎕WX for default code
/a:file           Specifies a JSON file containing attributes to be
attached
       to the binary
/i:Process        Set the isolation mode of a .NET Assembly
/i:Assembly       Set the isolation mode of a .NET Assembly
/i:AppDomain      Set the isolation mode of a .NET Assembly
/i:Local          Set the isolation mode of a .NET Assembly
```

Note that the isolation mode specified by the `/i` option overrides the setting in `web.config`. See *Section 12.7*.

The `/a` option is used to specify the name of a JSON file that contains assembly info. For example:

```
dyalogc64_unicode.exe /t:library j:/ws/attributetest.dws /a:c:/tmp/
atts.json
```

where `c:/tmp/atts.json` contains:

```
{
   "AssemblyVersion": "1.2.2.2",
   "AssemblyFileVersion": "2.1.1.4",
   "AssemblyProduct": "My Application",
   "AssemblyCompany": "My Company",
   "AssemblyCopyright": "Copyright 2020",
   "AssemblyDescription": "Provides a text description for an
assembly.",
   "AssemblyTitle": "My Assembly Title",
   "AssemblyTrademark": "Your Legal Trademarks"
}
```

5.3 Creating an APLScript File

Conceptually, the simplest way to create an `APLScript` file is with Notepad, although you may use many other tools including Microsoft Visual Studio as described in the next Chapter.

1. Start Notepad
2. *Choose Format/Font* from the Menu Bar and select an appropriate Unicode font that contains APL symbols, such as *APL 385 Unicode*.
3. Select an APL keyboard by clicking on your keyboard selector in the System Tray. Note that this keyboard setting (and button) is associated only with the current instance of Notepad. If you start another instance of Notepad, or another editor, you will have to select the APL keyboard for it separately and there will be two floating toolbars on your display.
4. Now type in your APL code. If you use a Ctrl keyboard, you will discover that Ctrl+ keystrokes generate APL symbols For example, Ctrl+n generates `⍳`.
5. Choose *File/Save*. When the *Save As* dialog appears, ensure that *Encoding* is set to *Unicode* and *Save as type*: is set to *All Files*. Enter the name of the file, adding the extension .asmx or .aspx, and then click *Save*. Note that you have to save the .asmx file somewhere in an IIS Virtual Directory structure.

5.4 Copying code from the Dyalog Session

You may find it easier to write APL code using the Dyalog APL function or class editor that is provided by the Dyalog APL Session. Or you may already have code in a workspace that you want to copy into an `APLScript` file.

If so, you can transfer code from the Session into your `APLScript` editor (for example, Notepad) using the clipboard. Notice that because APLScript requires Unicode encoding (for APL symbols), you must ensure that character data is written to the clipboard in Unicode.

In the Unicode interpreter this is always done. In the Classic interpreter this is controlled by a parameter called UnicodeToClipboard that specifies whether or not data is transferred to and from the Windows clipboard as Unicode. This parameter may be changed using the Trace/Edit page of the Configure dialog box.

If set (the default), APL text pasted to the clipboard from the Session is written as Unicode and APL requests Unicode data back from the clipboard when it is required. This makes it easy to transfer APL code between the Session and an APLScript editor.

In the Classic interpreter when pasting code *into* the Dyalog editor, there are two menu items under the Edit menu, which allow you to explicitly select whether the Unicode mapping should be used, or the old mapping which corresponds to the Dyalog Std TT or Dyalog Alt TT fonts. You should use "Paste non-Unicode" when transferring text from the on line help, or text copied from earlier versions of Dyalog APL without the Unicode option.

Unless you explicitly want to have line numbers in your `APLScript`, the simplest way to paste APL code from the Session into an APLScript text editor is as follows:

1. open the function in the function editor
2. select all the lines of code, or just the lines you want to copy
3. select *Edit/Copy* or press Ctrl+Ins
4. switch to your `APLScript` editor and select *Edit/Paste* or press Shift+Ins.
5. insert Del (∇) symbols at the beginning and end of the function.

If you want to preserve line numbers (this is allowed, but not recommended in APLScript files), you may use the following technique:

1. in the Session window, type a del (∇) symbol followed by the name of the function, followed by another del (∇) and then press Enter. This causes the function to be displayed, with line numbers, in the Session window.

2. select the function lines, including the surrounding Dels (∇) and choose *Edit/Copy* or press Ctrl+Insert.
3. switch to your `APLScript` editor and select *Edit/Paste* or press Shift+Ins.

5.5 General principles of APLScript

The layout of an `APLScript` file differs according to whether the script defines a Web Page, a Web Service, a .NET class, or an APL application that may have nothing to do with the .NET Framework. However, within the `APLScript`, the code layout rules are basically the same.

An APLScript file contains a sequence of function bodies and executable statements that assign values to variables. In addition, the file typically contains statements that are directives to the APLScript compiler. If the script is a Web Page or Web Service, it may also contain directives to ASP.NET. The former all start with a colon symbol (:) in the manner of control structures. For example, the `:Namespace` statement tells the APLScript compiler to create, and change into, a new namespace. The `:EndNamespace` statement terminates the definition of the contents of a namespace and changes back from whence it came.

Assignment statements are used to set up system variables, such as `⎕ML`, `⎕IO`, `⎕USING` and arbitrary APL variables. For example:

```
⎕ML←2
⎕IO←0
⎕USING∪←⊂'System.Data'

A←88
B←'Hello World'

⎕CY'MYWS'
```

These statements are extracted from the `APLScript` and executed by the compiler in the order that they appear. It is important to recognise that they are executed at compile time, and not at run-time, and may therefore only be used for initialisation.

Notice that it **is** acceptable to execute `⎕CY` to bring in functions and variables from a workspace that are to be incorporated into the code. This is especially useful to import a set of utilities. Note also that it is possible to export these functions as methods of .NET classes if the functions contain the appropriate colon statements.

The `APLScript` compiler will in fact execute any valid APL expression that you include. However, the results may not be useful and may indeed simply terminate the compiler. For example, it is not sensible to execute statements such as `⎕LOAD`, or `⎕OFF`.

Function bodies are defined between opening and closing del (`∇`) symbols. These are fixed by the `APLScript` compiler using `⎕FX`. Line numbers and white space formatting are ignored.

5.6 Creating Programs (.exe) with APLScript

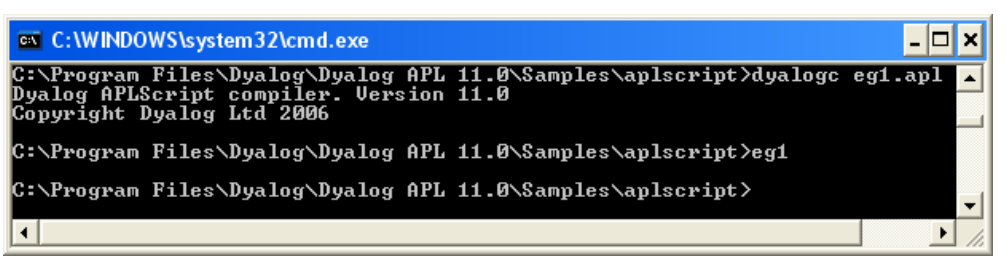
The following examples, which illustrate how you can create an executable program (`.exe`) direct from an APLScript file, may be found in the directory `samples\aplscript`.

A simple GUI example

The following APLScript illustrates the simplest possible GUI application that displays a message box containing the string "Hello World".

```
:Namespace N
⎕LX←'N.RUN'
∇RUN;M
'M'⎕WC'MsgBox' 'A GUI exe' 'Hello World'
⎕DQ'M'
∇
:EndNamespace
```

This example, which is saved in the file `eg1.apl`, is compiled to a Windows executable (`.exe`) using `dyalogc.exe` and run from the same command window as shown below. Notice that it is essential to surround the code with `:Namespace / :EndNamespace` statements and to define a `⎕LX` either in the APLScript itself, or as a parameter to the `dyalogc` command.



You can associate the `.exe` with a desktop icon, and it will run stand-alone, without a Command Prompt window. Furthermore, any default APL output that would normally be displayed in the session window will simply be ignored.

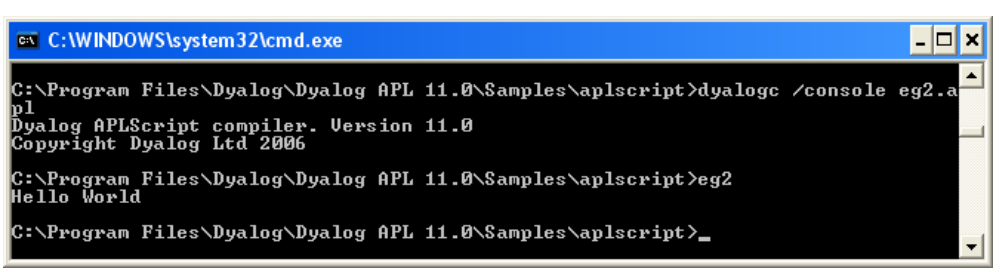
A simple console example

The following APLScript illustrates the simplest possible application that displays the text "Hello World".

This example, which is saved in the file `eg2.apl`, is compiled to a Windows executable (`.exe`) and run from a command window as shown below. Notice that the `/console` flag is used to tell the APLScript compiler to create a *console* application that runs from a command prompt. In this case, default APL output that would normally be displayed in the session window turns up in the command window from which the program was run.

```
:Namespace N
⎕LX←'N.RUN'
∇RUN
'Hello World'
∇
:EndNamespace
```

Once more, it is essential to surround the code with `:Namespace/:EndNamespace` statements and to define a `⎕LX` either in the APLScript itself, or as a parameter to the `dyalogc` command.



Defining Namespaces

Namespaces are specified in an APLScript using the `:Namespace` and `:EndNamespace` statements. Although you may use `⎕NS` and `⎕CS` within functions inside an APLScript, you should not use these system functions outside function bodies. Note that such use is not *prevented*, but that the results will be unpredictable.

```
:Namespace Name
```

introduces a new namespace called `Name` relative to the current space.

```
:EndNamespace
```

terminates the definition of the current namespace. Subsequent statements and function bodies are processed in the context of the original space.

It is imperative that at least ONE namespace be specified.

All functions specified between the `:Namespace` and `:EndNamespace` statements are fixed in that namespace. Similarly, all assignments define variables inside that namespace.

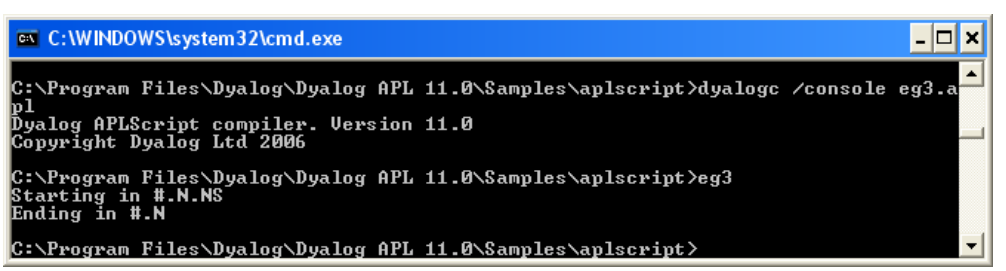
The following example illustrates how APL namespace usage is handled in APLScript. The program, contained in the file `eg3.apl`, is as follows:

```
:Namespace N
⎕LX←'N.RUN'

∇RUN
⎕PATH←'↑'
NS.START
END
∇
∇R←CURSPACE
R←⊃⎕NSI
∇
∇END
'Ending in ',CURSPACE
∇

:NameSpace NS
∇START
'Starting in ',CURSPACE
∇
:EndNameSpace
:EndNameSpace
```

This somewhat contrived example illustrates how a namespace is defined inside another namespace using `:NameSpace` and `:EndNamespace` statements. The namespace `NS` contains a single function called `START`, which is called from the main function `RUN`.

Notice that ⎕PATH is defined *dynamically* in function RUN. If it were defined outside a function in a static statement in the script (say, after the statement that sets ⎕LX), it would not be honoured when the application was run.

This program is shown, compiled and run as a console application, below.



5.7 Creating .NET Classes with APLScript

It is possible to define and use new .NET classes within an `APLScript`.

A class is defined by `:Class` and `:EndClass` statements. The methods provided by the class are defined as function bodies enclosed within these statements. Please see the Language Reference for a complete discussion of writing classes in Dyalog. This chapter will only provide a brief introduction to the subject, aimed specifically at APLScript.

You may also define sub-classes or nested classes using nested `:Class` and `:EndClass` statements.

```
      :Class Name: Type
```

Declares a new class called `Name`, which is based upon the Base Class `Type`, which may be any valid .NET Class.

```
      :EndClass
```

Terminates a class definition block

A class specified in this way will automatically support the methods, properties and events that it inherits from its Base Class, together with any new public methods that you care to specify.

However, the new class only inherits a default constructor (which is called with no parameters) and does not inherit all of the other private constructors from its Base Class. You can define a method to be a constructor using the `:Implements Constructor` declarative comment. Constructor overloading is supported and you may define any number of different constructor functions in this way, but they must have unique parameter sets for the system to distinguish between them.

You can create and use instances of a class by using the `⎕NEW` system function in statements elsewhere in the `APLScript`.

Exporting Functions as Web Methods

Within a `:Class` definition block, you may define private functions and public functions. A public function is one that is exposed as a method and may be called by a client that creates an instance of your class. Public functions must have a section of *declaration* statements. Other functions are purely internal to the class and are not directly accessible by a client application.

The declaration statements for public functions perform the same task for an `APLScript` that is performed using the .NET Properties dialog box, or by executing SetMethodInfo in the Dyalog Session, prior to creating a .NET assembly. The following declaration statements may be used.

```
      :Access Public
```

Specifies that the function is callable. This statement applies only to a .NET class or to a Web Page and is not applicable to a Web Service.

```
      :Access WebMethod
```

Specifies that the function is callable as a Web Method. This statement applies only to a Web Service (.asmx). From version 11.0, the statement is equivalent to:

```
      :Access Public
      :Attribute System.Web.Services.WebMethodAttribute
      :Implements Constructor
```

Specifies that the function is a constructor for a new .NET class. This function must appear between `:Class` and `:EndClass` statements and this applies only to a Web Page (.aspx). See *Defining Classes in APLScript* for further details. A constructor is called when you execute the `New` method in the class.

```
      :Signature result←fn type1 Name1, type2 Name2,..
```

Declares the result of the method to have a given data type, if any. It also declares parameters to the method to have given data types and names. `Namex` is optional and may be any well-formed name that identifies the parameter. This name will appear in the metadata and is made available to a client application as information. It is therefore sensible to choose meaningful names. The names you allocate to parameters have no other meaning and are not associated with the names of local variables that you may choose to receive them. However, it is not a bad idea to use the same local names as the public names of your parameters.

A .NET Class example

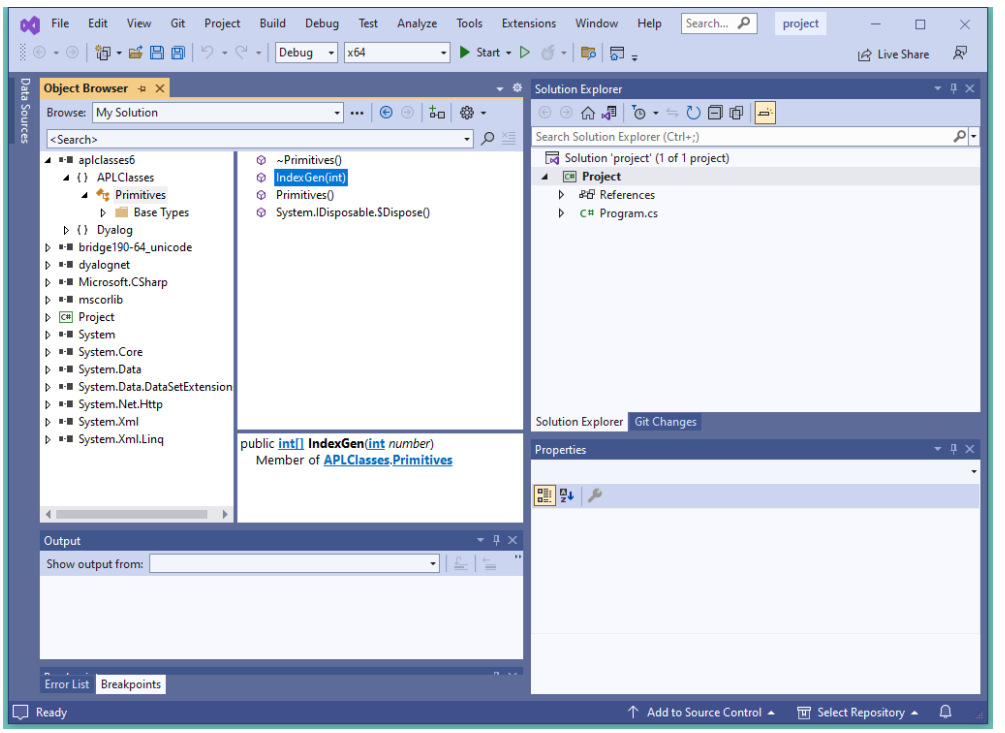
The following APLScript illustrates how you may create a .NET Class using APLScript. The example class is the same as *Example 1* in Chapter 5. The APLScript code, saved in the file `samples\aplclasses\aplclasses6.apl`, is as follows:

```
:Namespace APLClasses

    :Class Primitives: Object
        ⎕USING←,⊂'System'
    :Access public

        ∇ R←IndexGen N
          :Access Public
          :Signature Int32[]←IndexGen Int32 number
          R←⍳N
        ∇
    :EndClass

:EndNamespace
```

This APLScript code defines a namespace called `APLClasses`. This simply acts as a container and is there to establish a .NET namespace of the same name within the resulting .NET assembly. Within `APLClasses` is defined a .NET class called `Primitives` whose base class is `System.Object`. This class has a single public method named `IndexGen`, which takes a parameter called `number` whose data type is `Int32`, and returns an array of `Int32` as its result.

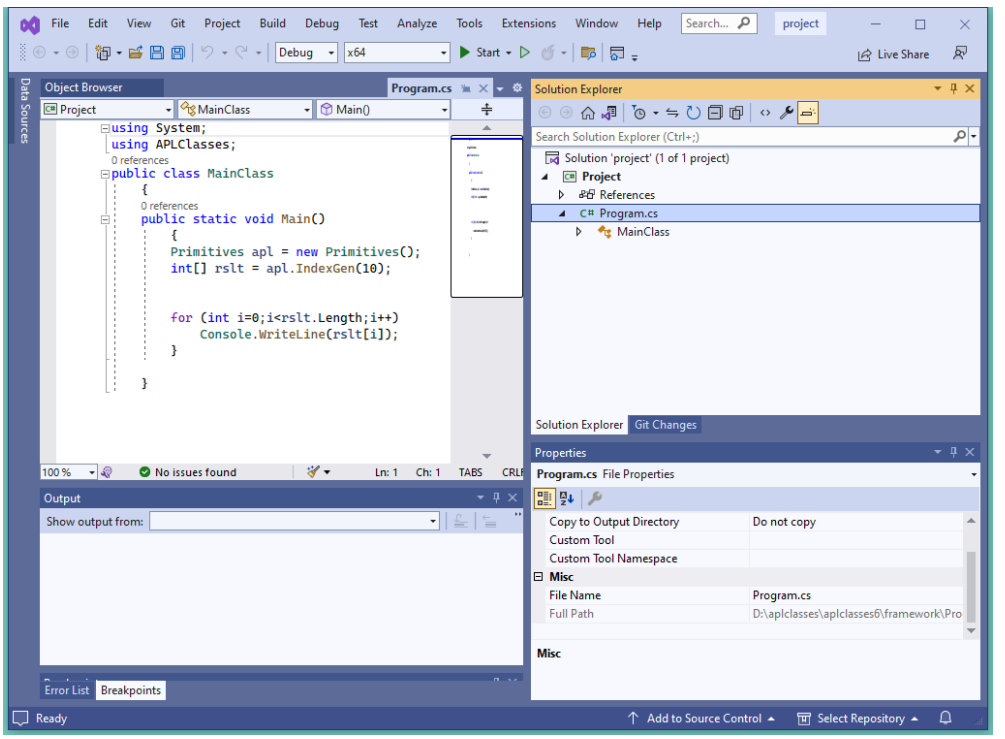
`aplclasses6.apl` is compiled to a .NET Assembly using the APLScript compiler with the `/t:library` flag. For details, see the file `aplclasses6\framework\build.bat`.

Using VS, open the solution file `d:\aplclasses\aplclasses6\Framework\project.sln` and view `aplclasses6.dll`.

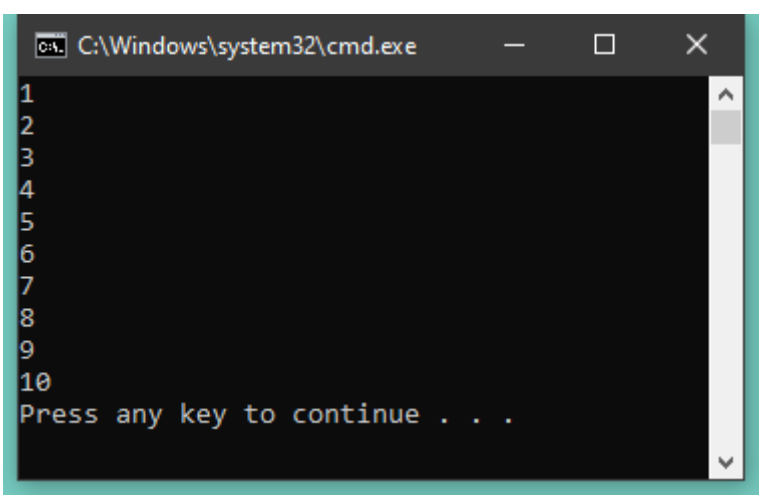


This shows that `aplclasses6.dll` contains a .NET namespace called APLClasses, which in turn contains the Primitives class. Primitives has a single method named IndexGen() which takes a number (an int).

Next, Display the c# program `program.cs`. This is the same program as in Example1.



Now click *Debug/Start Without debugging* (or press Ctrl+F5) to run the program. The results are shown in a console window.



This .NET Class can also be called from APL just like any other. For example:

```
      )CLEAR
clear ws

       'APLClasses, D:
\aplclasses\aplclasses6\framework\bin\aplclasses6.dll'

      APL←⎕NEW Primitives
      APL.IndexGen 10
1 2 3 4 5 6 7 8 9 10
```

Defining Properties

Properties are defined by `:Property` and `:EndProperty` statements. A property pertains to the class in which it is defined.

```
:Property Name
     ∇ C←get
[1]    :Access public
[2]    :Signature Double←get
[3]     C←...
     ∇
```

Declares a new property called `Name` whose data type is `System.Double`. The latter may be any valid .NET type which can be located through `⎕USING`.

```
:EndProperty
```

Terminates a property definition block

Within a `:Property` block, you must define the *accessors* of the property. The accessors specify the code that is associated with referencing and assigning the value of the property. No other function definitions or statements are allowed inside a `:Property` block.

The accessor used to reference the value of the property is represented by a function named `get` that is defined within the `:Property` block. The accessor used to assign a value to the property is represented by a function named `set` that is defined within the `:Property` block.

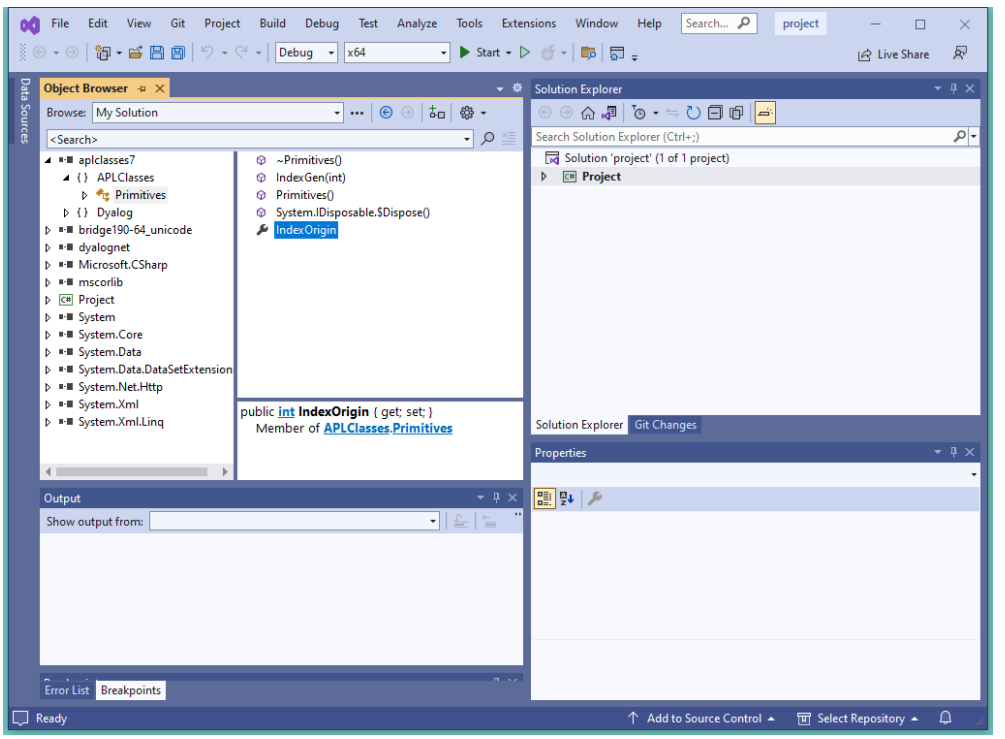
The `get` function is used to retrieve the value of the property and must be a niladic result returning function. The data type of its result determines the `Type` of the property. The `set` function is used to change the value of the property and must be a monadic function with no result. The argument to the function will have a data type

`Type` specified by the `:Signature` statement. A property that contains a `get` function but no `set` function is effectively a read-only property.

The following APLScript, saved in the file `samples\aplclasses\aplclasses7.apl`, shows how a property called `IndexOrigin` can be added to the previous example. Within the `:Property` block there are two functions defined called `get` and `set` which are used to reference and assign a new value respectively. These functions have the fixed names and syntax specified for *property get* and *property set* functions as described above.

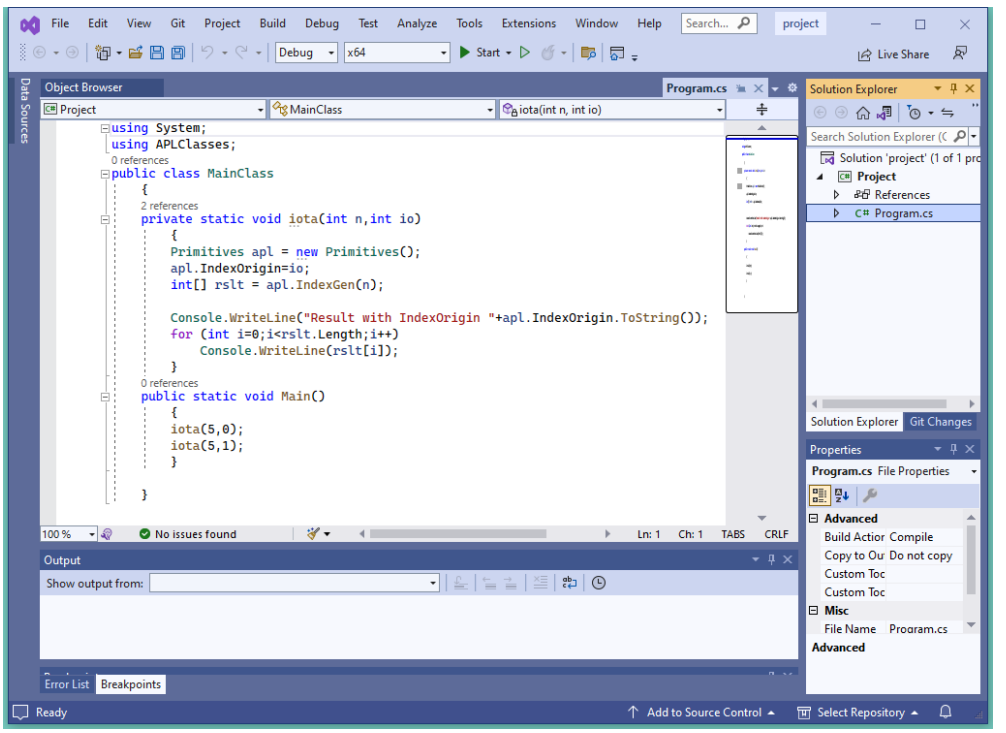
```
:Namespace APLClasses

    :Class Primitives: Object
        ⎕USING←,⊂'System'
    :Access public

        ∇ R←IndexGen N
          :Access Public
          :Signature Int32[]←IndexGen Int32 number
          R←⍳N
        ∇

        :Property IndexOrigin
            ∇ io←get
              :Signature Int32←get Int32 number
              io←⎕IO
            ∇

            ∇ set io
              :Signature set Int32 number
              :If io∊0 1
                  ⎕IO←io
              :EndIf
            ∇
        :EndProperty

    :EndClass

:EndNamespace
```

Using VS, open the solution file `d:\aplclasses\aplclasses7\Framework\project.sln` and view `aplclasses7.dll`.

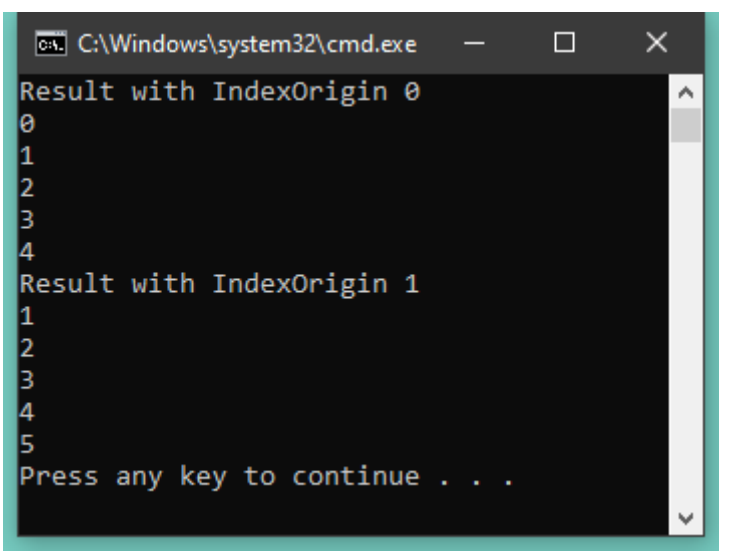


This shows that `aplclasses7.dll` contains a .NET namespace called APLClasses, which in turn contains the Primitives class. Primitives has a single method named IndexGen() which takes a number (an int) and a property named IndexOrigin.

Next, Display the c# program `program.cs`. Notice that the main program calls a subroutine iota twice to calculate `⍳5` in origin 0 and 1.



Now click *Debug/Start Without debugging* (or press Ctrl+F5) to run the program. The results are shown in a console window.



This .NET Class can also be called from APL just like any other. For example:

```
      )CLEAR
clear ws

      'APLClasses, D:
\aplclasses\aplclasses7\framework\bin\aplclasses7.dll'

      APL←⎕NEW Primitives
      APL.IndexGen 10
1 2 3 4 5 6 7 8 9 10
      APL.IndexOrigin
1
      APL.IndexOrigin←0
      APL.IndexGen 10
0 1 2 3 4 5 6 7 8 9
```

Indexers

An *indexer* is a property of a class that enables an instance of that class (an object) to be indexed in the same way as an array, if the host language supports this feature. Languages that support object indexing include C# and Visual Basic. Dyalog does also allow indexing to be used on objects. This means that you can define an APL class that exports an indexer and you can use the indexer from C#, Visual Basic, or Dyalog.

Indexers are defined in the same way as properties, between `:Property Default` and `:EndProperty` statements. There may be only one indexer defined for a class.

Note: the `:Property Default` statement in Dyalog is closely modelled on the indexer feature in C# and employs similar syntax. If you use `ILDASM` to browse a .NET class containing an indexer, you will see the indexer as the *default property* of that class, which is how it is actually implemented.

5.8 Creating ASP.NET Classes with APLScript

As mentioned previously, the original purpose of APLScript was to provide the ability to write ASP.NET Web Pages and Web Services in Dyalog. Both these applications are based upon script files.

Web Page Layout

An ASP.NET Web Page typically consists of a mixture of HTML and code written in a scripting language. The script code is separated from the HTML by being embedded within tags and normally appears in the section of the page. Only one block of script is

allowed in a page. The script block normally consists of a collection of functions, which are invoked by some event on the page, or on an element of the page.

`APLScript` code starts with a statement:

```
<script language="Dyalog" runat=server>
```

and finishes with:

```
</script>
```

Typically, the APLScript code consists of callback functions that are attached to server-side events on the page.

For further information, see *Section 12.7*.

Web Service Layout

The first line in a Web Service script must be a declaration statement such as:

```
<%@ WebService Language="Dyalog" Class="ServiceName" %>
```

where `ServiceName` is an arbitrary name that identifies your Web Service.

The next statement must be a `:Class` statement that declares the name of the Web Service and its Base Class from which it inherits. The base class will normally be `System.Web.Services.WebService`. For example:

```
:Class ServiceName: System.Web.Services.WebService
```

The last line in the script must be:

```
:EndClass
```

Although it may appear awkward to have to specify the name of your Web Service twice, this is necessary because the two statements are being processed quite separately by different software components. The first statement is processed by ASP.NET. When it sees `Language="Dyalog"`, it then calls the Dyalog `APLScript` compiler, passing it the remainder of the script file. The `:Class` statement tells the `APLScript` compiler the name of the Web Service and its base class. `:Class` and `:EndClass` statements are private directives to the `APLScript` compiler and are not relevant to ASP.NET.

How APLScript is processed by ASP.NET

Like any other Web Page or Web Service, an `APLScript` file is processed by ASP.NET.

The first time ASP.NET processes a script file, it first performs a compilation process whose output is a .NET assembly. ASP.NET then calls the code in this assembly to generate the HTML (for a Web Page) or to run a method (for a Web Service).

ASP.NET associates the compiled assembly with the script file, and only recompiles it if/when it has changed.

ASP.NET does not itself compile a script; it delegates this task to a specialised compiler that is associated with the language declared in the script. This association is made either in the application's web.config file or in the global `machine.config` file. Dyalog Installs a default web.config file which includes these settings in the `samples\asp.net` folder.

The `APLScript` compiler is itself written in Dyalog.

Although the compilation process takes some time, it is typically only performed once, so the performance of an `APLScript` Web Service or Web Page is not compromised. Once it has been compiled, ASP.NET redirects all subsequent requests for an `APLScript` to its compiled assembly.

Please note that the use of the word *compile* in this process does not imply that your APL code is actually compiled into Microsoft Intermediate Language (MSIL). Although the process does in fact generate *some* MSIL, your APL code will still be interpreted by the Dyalog DLL engine at run-time. The word *compile* is used only to be consistent with the messages displayed by ASP.NET when it first processes the script.

6 Writing .Net Classes

6.1 Introduction

Dyalog allows you to build new .NET Classes, components and controls. .NET classes created by Dyalog may be hosted by any application or programming language that supports .NET.

A component is a class with emphasis on cleanup and containment and implements specific interfaces.

A control is a component with user interface capabilities.

With one exception, every .NET Class inherits from exactly one base class. This means that it begins with all of the behaviour of the base class, in terms of the base class properties, methods and events. You add functionality by defining new properties, methods and events on top of those inherited from the base class or by overriding base class methods with those of your own.

6.2 Assemblies, Namespaces and Classes

To create a .NET class in Dyalog, you simply create a standard APL Class and export the workspace as a *Microsoft .NET Assembly (*.dll)*.

.NET Classes are organised in .NET Namespaces. If you wrap your Class (or Classes) within an APL namespace, the name of that namespace will be used to identify the name of the corresponding .NET Namespace in your Assembly.

If a Class is to be based upon a specific .NET Class, the name of that .NET Class must be specified as the Base Class in the `:Class` statement, and the `:Using` statements must correctly locate the base class. If not, the Class is assumed to be based upon System.Object. If you use any .NET Types within your Class, you must ensure that these too are located by :Using.

Once you have defined the functionality of your .NET classes, you are ready to save them in an assembly. This is simply achieved by selecting *Export* from the Session *File* menu.

You will be prompted to specify the directory and name of the assembly (DLL) and it will then be created and saved. Your .NET class is now ready for use by any .NET development environment, including APL itself.

When a Dyalog .NET class is invoked by a host application, it automatically loads the Dyalog DLL, the developer/debug or run-time dynamic link library version of Dyalog. You decide which of these DLLs is to be used according to the setting of the *Runtime application* checkbox in the *Create bound file* dialog box. Note however that the Dyalog .NET class, and all the Dyalog DLLs on which it depends, reside in the same directory as the host program.

Note that if you wish to include a Dyalog .NET class in a Visual Studio application it is recommended that you add the Bridge DLL as a reference in a Visual Studio .NET project.

If you want to repeat the most recent export after making changes to the class, you can click on the icon to the right of the save icon on the WS button bar at the top of the session. Note that the workspace itself is not saved when you do an export, so if you want the export options to be remembered you must `)SAVE` the workspace *after* you have exported it.

6.3 Getting Started

This tutorial, as provided, supports the 64-bit Unicode version of Dyalog only.

The tutorial described in this Chapter was originally designed (for Dyalog Version 10) to be exercised in a console window, with the user invoking the C# compiler directly using a command-line interface. It was originally envisaged to be run *in-situ* in the `samples\aplclasses` sub-directory.

Today, the `samples\aplclasses` sub-directory is read-only, and direct access to the C# compiler via a command-line interface is problematical. Another consideration is the change in requirement for dependent Dyalog DLLs, which must now reside in the same directory as the host program.

The tutorial has therefore been re-factored to use Microsoft Visual Studio. The `samples\aplclasses` sub-directory has been expanded to support .NET Core (now renamed simply .NET) which is cross-platform as well as the original .NET Framework which is Windows only.

All the examples are to be executed as simple console applications written in C# in the framework of *Microsoft Visual Studio Community 2022* (hereafter referred to as VS). To run the examples as described herein, you should install VS, taking care to include all

the components needed to create a C# console application. suitable VS project files are included in the `samples\aplclasses` sub-directory.

Initialisation

The first step is to copy the `samples\aplclasses` sub-directory into a directory to which you have write access. For example, into `d:\aplclasses`.

Each of the sub-directories contained in `aplclasses`, namely `aplclasses1` - `aplclasses7`, represents a separate example application. Within each one the file structure is as follows:

`aplclasses[n].dws`	APL workspace
`Framework`	Files for the .NET Framework
`Framework\program.cs`	C# program
`Framework\project.sln`	VS solution file
`Framework\project.csproj.`	C# project file
`Framework\bin`	Directory containing the C# program and DLLs

When the application is executed by VS it will be run in the `bin` sub-diredctory.

It is mandatory that the Dyalog .NET class, and all the Dyalog DLLs on which it depends, reside in the same directory as the host program.

Therefore, copies of the requisite Dyalog DLLs are provided in the `bin`sub-directory. These DLLs are:

- Development DLL and/or Run-Time DLL (this tutorial uses the Development DLL)
- Bridge DLL
- DyalogNet DLL

Running the Tutorial

Each example consists of two parts. First you will `)LOAD` a workspace, examine the code, and then export it as a DLL. The second (optional) part is to run the VS solution that hosts the DLL and view the results.

Each workspace contains a .NET Namespace called `APLClasses` which itself contains a single .NET Class called `Primitives` that exports a single method called `IndexGen`.

6.4 Example 1

Load the workspace `aplclasses1.dws`, then view the `Primitives` class:

```
      )LOAD D:\aplclasses\aplclasses1\aplclasses1.dws
D:\aplclasses\aplclasses1\aplclasses1.dws ⍝ saved ...

      )ED ⍥APLClasses.Primitives
```

> **Note**
>
> The character before the name `APLClasses.Primitives`, `⍥`, is typically obtained with `Ctrl-O`. It is used to tell the editor to edit a class.

```
:Class Primitives
:Using System
     ∇ r←IndexGen n
      :Access public
      :Signature Int32[]←IndexGen Int32 n
      r←⍳n
     ∇
:EndClass
```

`Primitives` contains one public method/function named `IndexGen`.

The public characteristics for the exported method are included in the definition of the class and its functions. Those are specified in the `:Signature` statement.

Its syntax is:

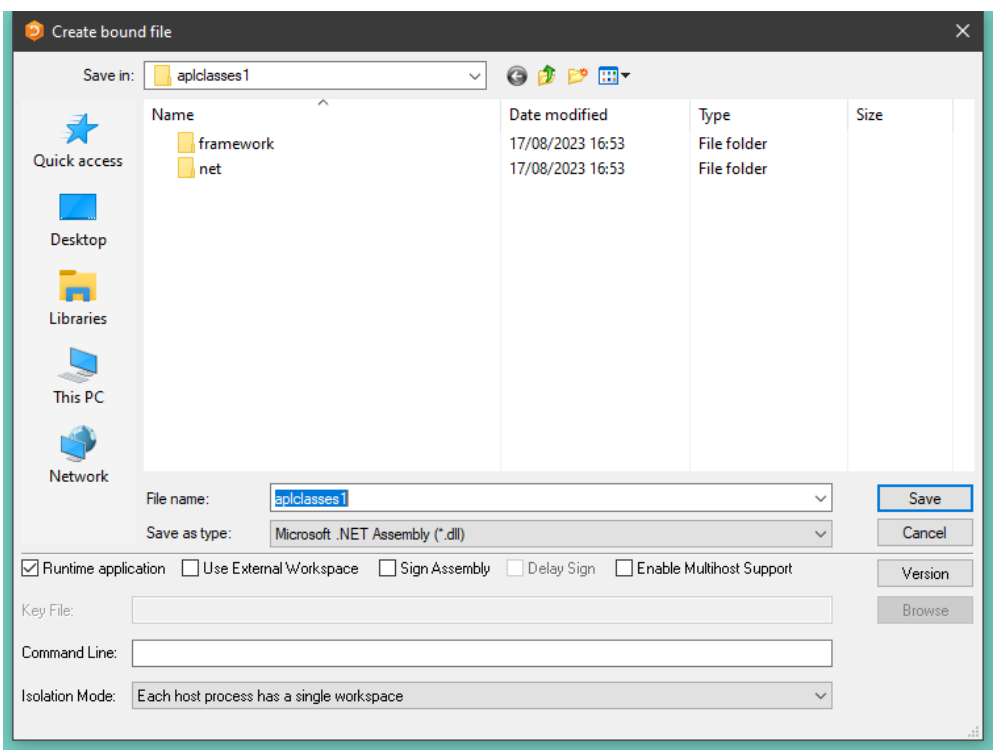
```
:Signature [return type←] fnname [arg1type [arg1name]
                                 [,argNtype [argNname]]*]
```

that is: The type of the result returned by the function - followed by arrow - if any, the exported name (it can be different from the APL function name but it must be provided), and, if any arguments are to be supplied, their types and optional names, each type-name pair separated from the next by a comma. In the example above the function returns an array of 32-bit integers and takes a single integer as its argument. For further details, see *Programming: Signature*.

Note that, when the class is fixed, APL will try to find the .NET data types you have specified for the result and for the parameters. If one or more of the data types are not recognised as available .NET Types, you will be informed in the status window and APL will refuse to fix the class. If you see such a warning you have either entered an incorrect data type name, or you have not set `:Using` correctly, or some other syntax problem has been detected (for example the function is missing a terminating ∇. In the previous example, the only data type used is `System.Int32`. Since we have set `:Using System`, the name `Int32` is found in the right place and all is well.

It should be noted that in the previous release of Dyalog the statements `:Returns` and `:ParameterList` were used instead of `:Signature`. They are still accepted for backwards compatibility but are considered deprecated. Their syntax will not be documented here but a list can be found in Appendix A.

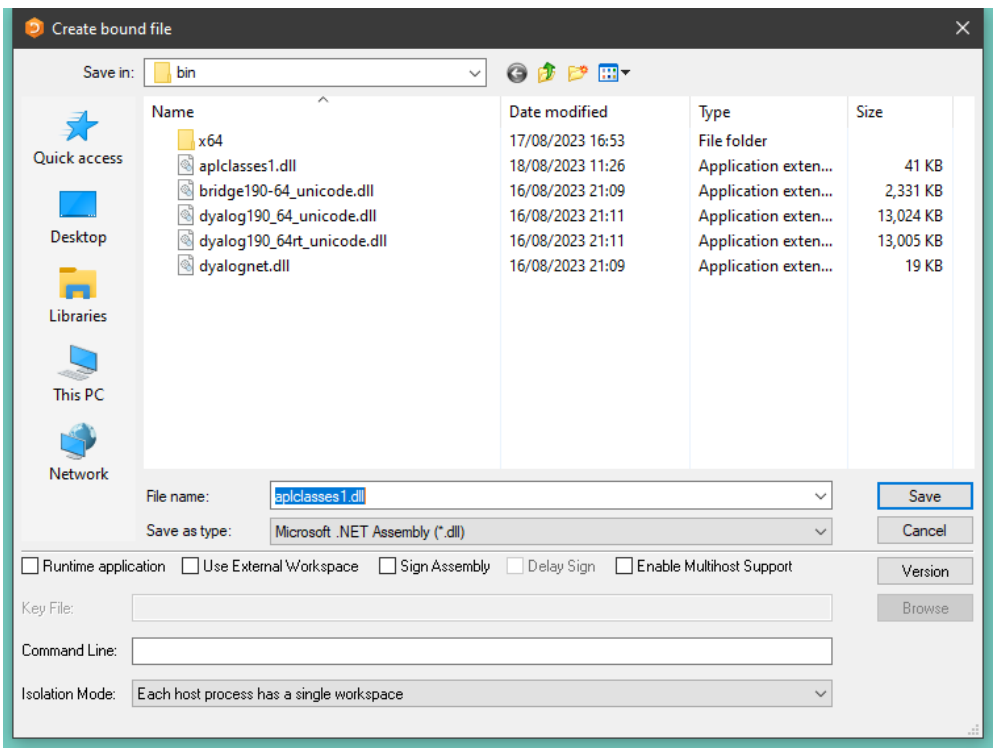
Now you are ready to create the assembly. This is done by selecting *Export...* from the Session *File* menu. This displays the following dialog box.



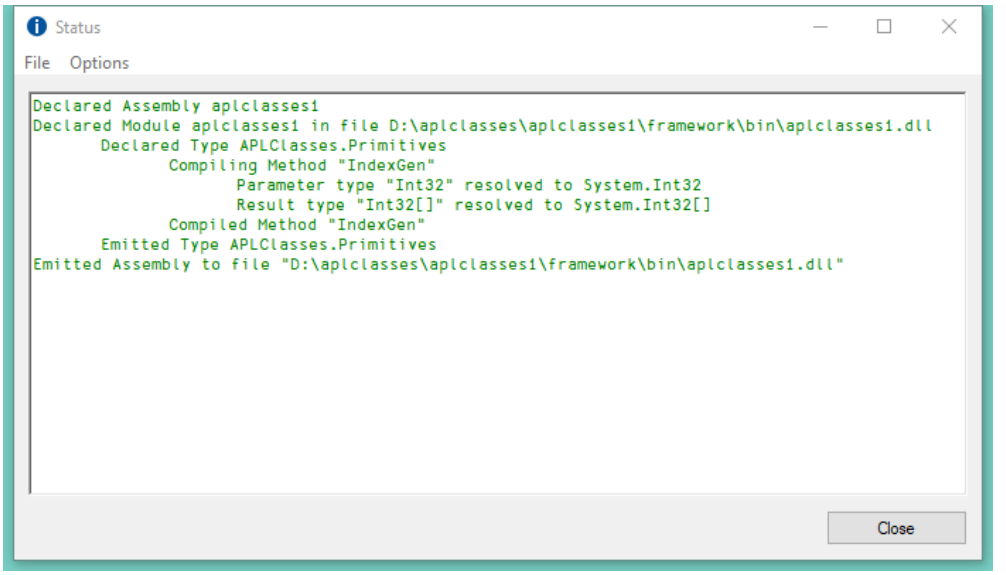
This gives you the opportunity to change the name or path of the assembly. The *Runtime application* checkbox allows you to choose to which if the two versions of the Dyalog dynamic link library the assembly will be bound. In this tutorial we will use the

Development version. The *Isolation Mode* Combo box allows you to choose which Isolation Mode you require.

- Browse to the `Framework\bin` sub-diectory.
- Clear the *Runtime application* checkbox



Finally, click *Save*. APL now makes the assembly and, as it does so, displays information in the Status window as shown below. If any errors occur during this process, the Status window will inform you.



program.cs

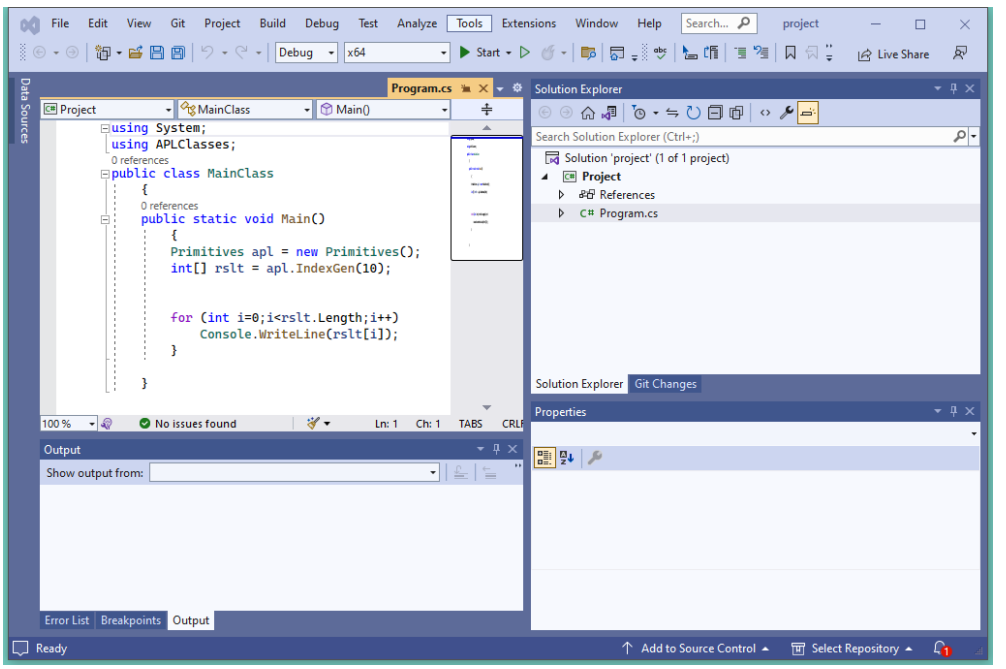
The following C# source, called `aplclasses1Framework\program.cs`, will be used to call our Dyalog.NET Class.

The `using` statements specify the names of .NET namespaces to be searched for unqualified class names.

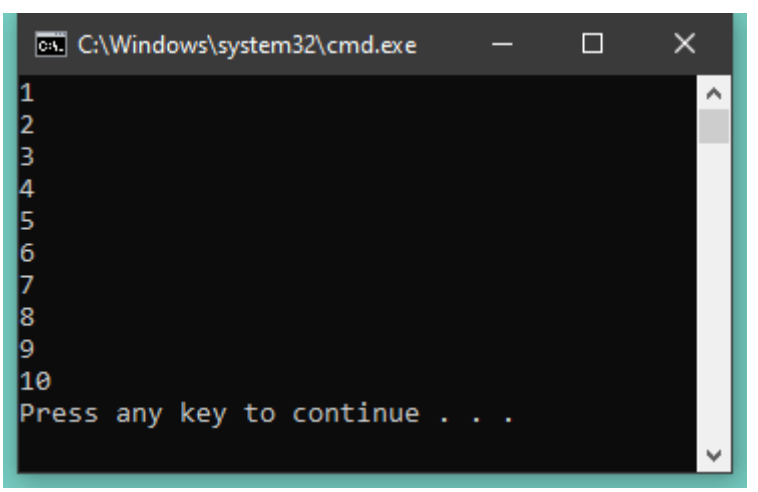
The program creates an object named `apl` of type `Primitives` by calling the `new` operator on that class. Then it calls the `IndexGen` method with a parameter of 10.

```
using System;
using APLClasses;
public class MainClass
    {
    public static void Main()
        {
            Primitives apl = new Primitives();
            int[] rslt = apl.IndexGen(10);
            for (int i=0;i<rslt.Length;i++)
            Console.WriteLine(rslt[i]);
        }
    }
```

Using VS, open the solution file `d:\aplclasses\aplclasses1\Framework\project.sln` and view `program.cs`.



Now click *Debug/Start Without debugging* (or press `Ctrl+F5`) to run the program. The results are shown in a console window.



6.5 Example 2

In Example 1, we said nothing about a constructor used to create an instance of the `Primitives` class. In Example 2, we will show how this is done.

In fact, in Example 1, APL supplied a default constructor, which is inherited from the base class (`System.Object`) and is called without arguments.

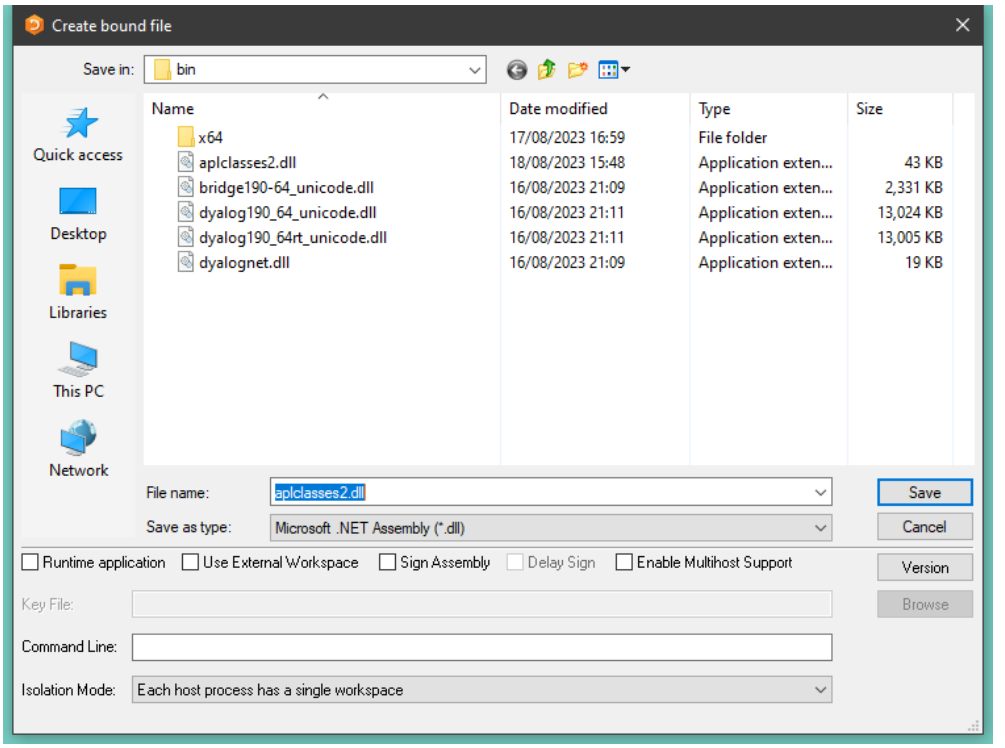
Example 2 will extend Example 1 by adding a constructor that specifies the value of `⎕IO`.

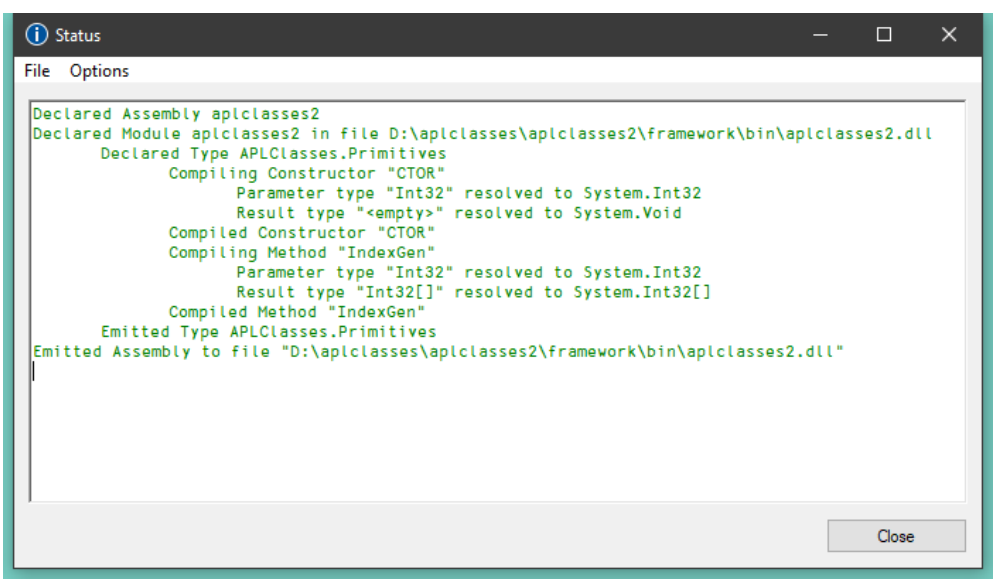
Load the workspace `aplclasses2.dws` from `aplclasses2`, then display the `Primitives` class:

```
      ↑⎕SRC APLClasses.Primitives
:Class Primitives
:Using System

    ∇ CTOR IO
      :Implements constructor
      :Access public
      :Signature CTOR Int32 IO
      ⎕IO←IO
    ∇

    ∇ R←IndexGen N
      :Access public
      :Signature Int32[]←IndexGen Int32
      R←⍳N
    ∇

:EndClass ⍝ Primitives
```

This version of `Primitives` contains a *constructor* function called `CTOR` that simply sets `⎕IO` to the value of its argument. The name of this function is purely arbitrary.

Using this version, build a new .NET Assembly using *File/Export…* as before. Remember that the *Build runtime assembly* checkbox is unchecked.


Create bound file
Save in:
bin
Quick access
Desktop
Libraries
This PC
Network
Name
Date modified
Type
Size
x64
17/08/2023 16:59
File folder
aplclasses2.dll
18/08/2023 15:48
Application exten...
43 KB
bridge190-64_unicode.dll
16/08/2023 21:09
Application exten...
2,331 KB
dyalog190_64_unicode.dll
16/08/2023 21:11
Application exten...
13,024 KB
dyalog190_64rt_unicode.dll
16/08/2023 21:11
Application exten...
13,005 KB
dyalognet.dll
16/08/2023 21:09
Application exten...
19 KB
File name:
aplclasses2.dll
Save as type:
Microsoft .NET Assembly (*.dll)
Save
Cancel
Runtime application
Use External Workspace
Sign Assembly
Delay Sign
Enable Multihost Support
Version
Key File:
Browse
Command Line:
Isolation Mode:
Each host process has a single workspace


Status
File
Options
Declared Assembly aplclasses2
Declared Module aplclasses2 in file D:\aplclasses\aplclasses2\framework\bin\aplclasses2.dll
Declared Type APLClasses.Primitives
Compiling Constructor "CTOR"
Parameter type "Int32" resolved to System.Int32
Result type "<empty>" resolved to System.Void
Compiled Constructor "CTOR"
Compiling Method "IndexGen"
Parameter type "Int32" resolved to System.Int32
Result type "Int32[]" resolved to System.Int32[]
Compiled Method "IndexGen"
Emitted Type APLClasses.Primitives
Emitted Assembly to file "D:\aplclasses\aplclasses2\framework\bin\aplclasses2.dll"
Close

program.cs

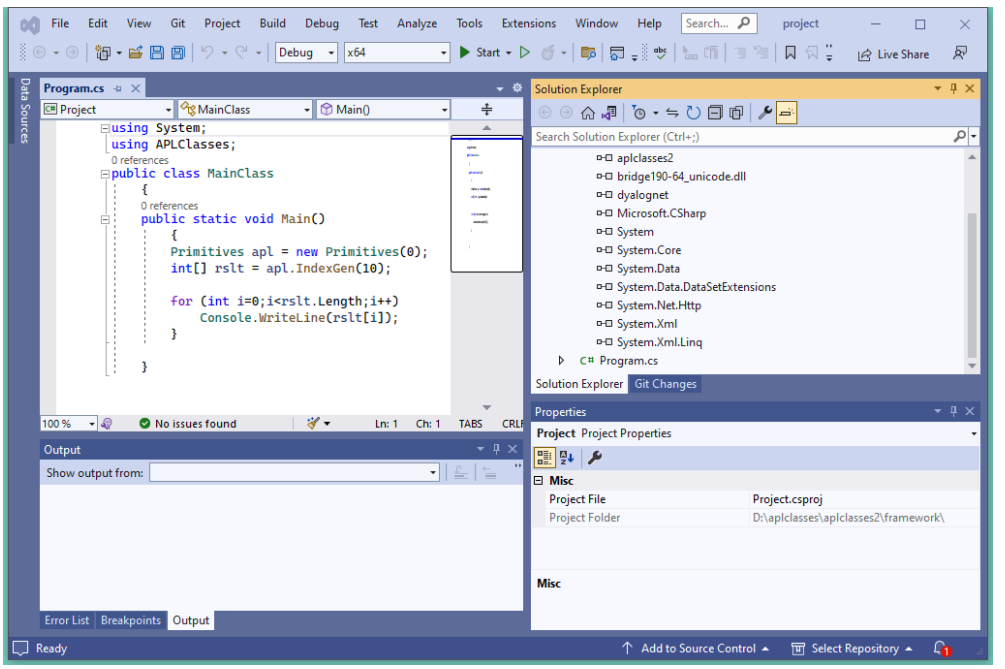
The following C# source, called `aplclasses2\Framework\program.cs`, will be used to call the new version of our Dyalog.NET class.

```
      using System;
      using APLClasses;
      public class MainClass
            {
            public static void Main()
                  {
                  Primitives apl = new Primitives(0);
                  int[] rslt = apl.IndexGen(10);

                  for (int i=0;i<rslt.Length;i++)
                  Console.WriteLine(rslt[i]);
                  }
            }
```

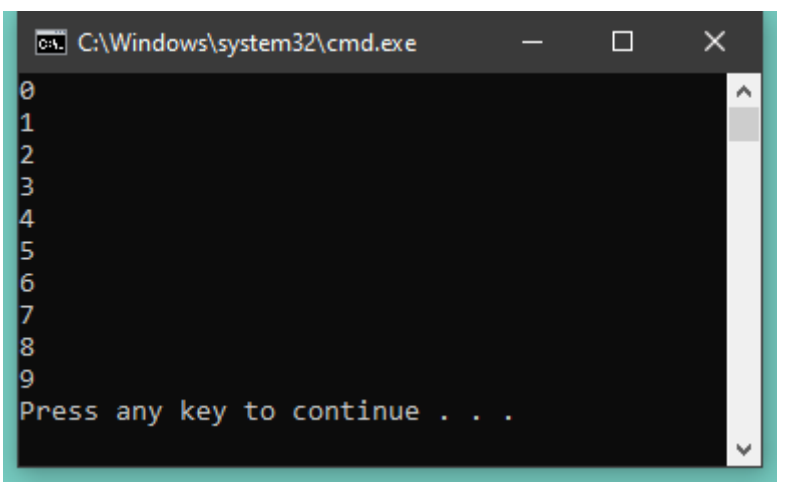
The program is the same as in the previous example, except that the code that creates an instance of the `Primitives` class is simply changed to specify an argument; in this case 0.

```
Primitives apl = new Primitives(0);
```

Using VS, open the solution file `d:\aplclasses\aplclasses2\Framework\project.sln` and view `program.cs`.



Then click *Debug/Start Without debugging* (or press `Ctrl+F5`) to run the program. The results are shown in a console window.



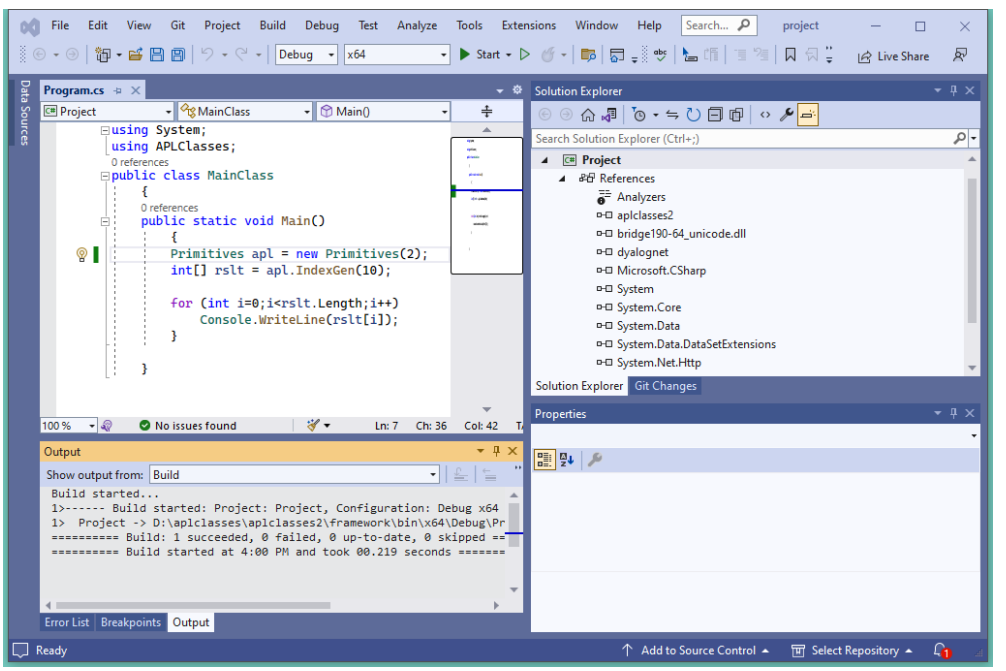
6.6 Example 2a

In Example 2, the argument to `CTOR`, the constructor for the `Primitives` class, was defined to be `Int32`. This means that the .NET Framework will allow a client to specify *any* integer when it creates an instance of the `Primitives` class. What happens if the

client uses a parameter of 2? Clearly this is going to cause an APL `DOMAIN ERROR` when used to set `⎕IO`.

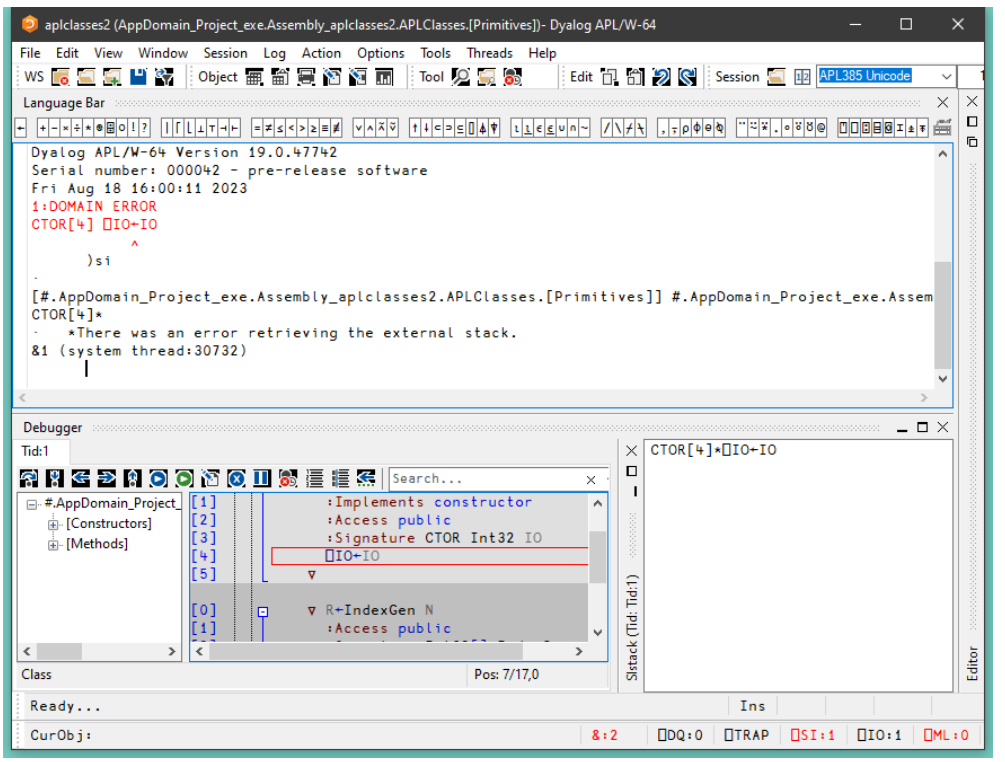
To investigate this case, change the line of code in `program.cs` that creates an instance of the Primitives class, passing the argument 2, like this:

```
Primitives apl = new Primitives(2);
```



Then click *Debug/Start Without debugging* (or press `Ctrl+F5`) to run the program.

… as we have built the Dyalog .NET class to use the *Development DLL*, the APL Session appears, and the Tracer can be used to debug the problem. You can see that the constructor `CTOR` has stopped with a `DOMAIN ERROR`. Meanwhile, the C# program is still waiting for the call (to create an instance of `Primitives`) to finish.

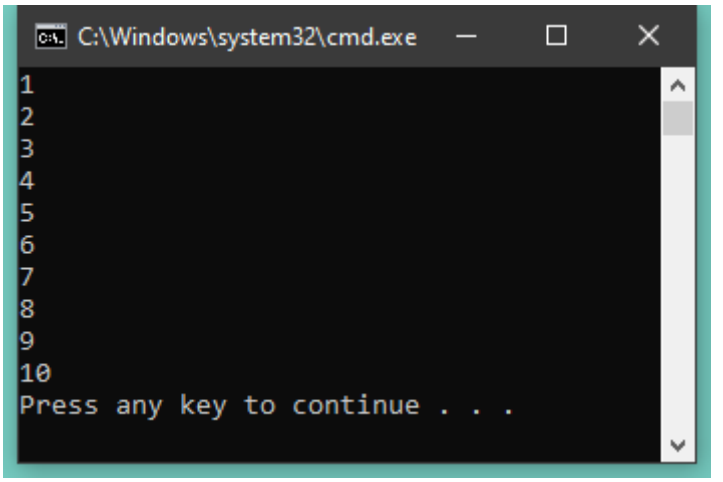


Notice that in Dyalog, the `)SI` System Command provides information about the entire calling stack, including the .NET function calls that are involved. Notice too that the `CTOR` function, the constructor for this APL .NET class, is running here in APL thread 1, which is associated with the system thread 30732.

In this case, debugging is simple, and you can simply type:

```
      IO←1
      →⎕LC
```

Now, the `CTOR` function completes, the program continues and the output is displayed.



6.7 Example 3

The correct .NET behaviour when an APL function fails with an error is to *throw an exception*, and this example shows how to do it.

In the .NET Framework, exceptions are implemented as .NET Classes. The base exception is implemented by the `System.Exception` class, but there are a number of super classes, such as `System.ArgumentException` and `System.ArithmeticException` that inherit from it.

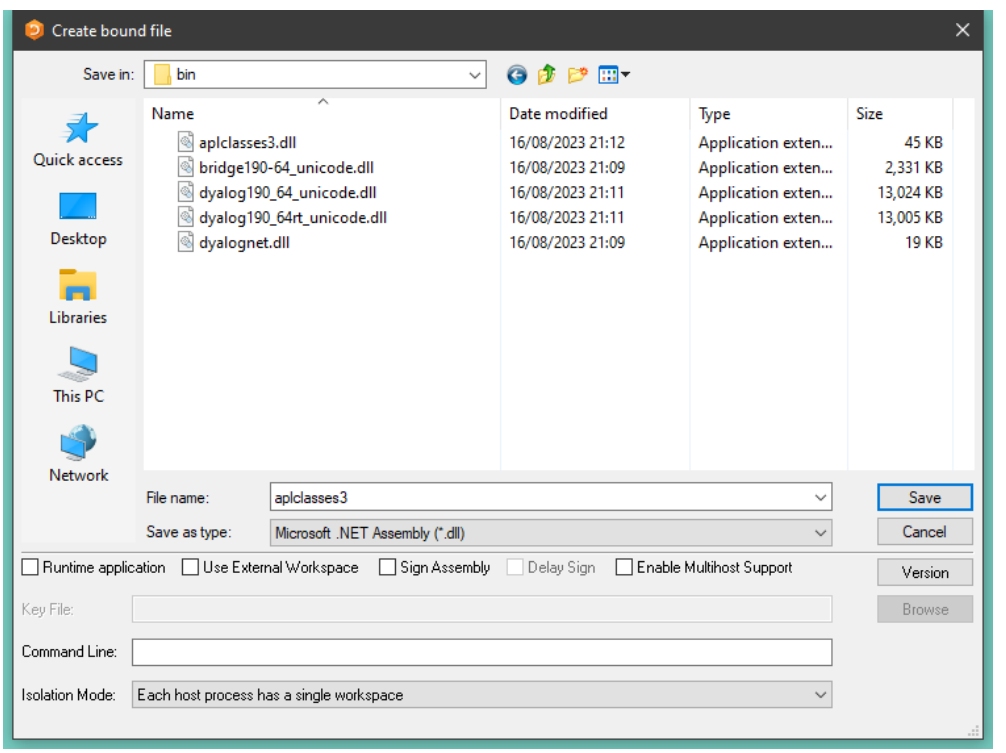
`⎕SIGNAL` may be used to *throw an exception*. To do so, its right argument should be 90 and its left argument should be an object of type `System.Exception` or an object that inherits from `System.Exception`.

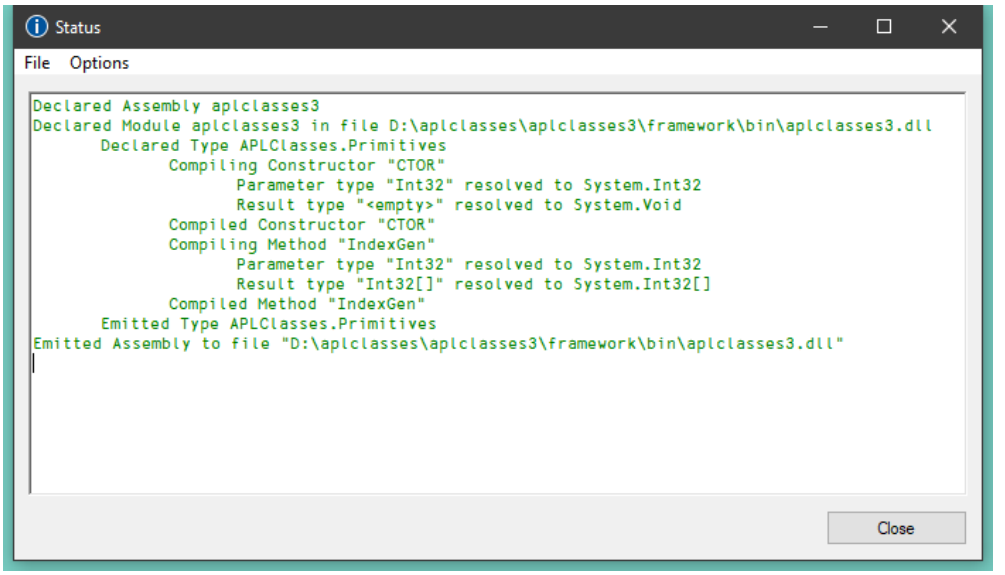
When you create the instance of the `Exception` class, you may specify a string (which will turn up in its `Message` property) containing information about the error.

`aplclasses3.dws` contains an improved version of the `CTOR` constructor function.

```
     ∇ CTOR IO;EX
[1]   :Access public
[2]   :Signature CTOR Int32 IO
[3]   :Implements constructor
[4]   :If IO∈0 1
[5]      ⎕IO←IO
[6]   :Else
[7]      EX←⎕NEW ArgumentException,⊂⊂'IndexOrigin must be
                                     0 or 1'
[8]      EX ⎕SIGNAL 90
[9]   :EndIf
     ∇
```

Load `aplclasses3.dws` and export `aplclasses3.dll` as before.



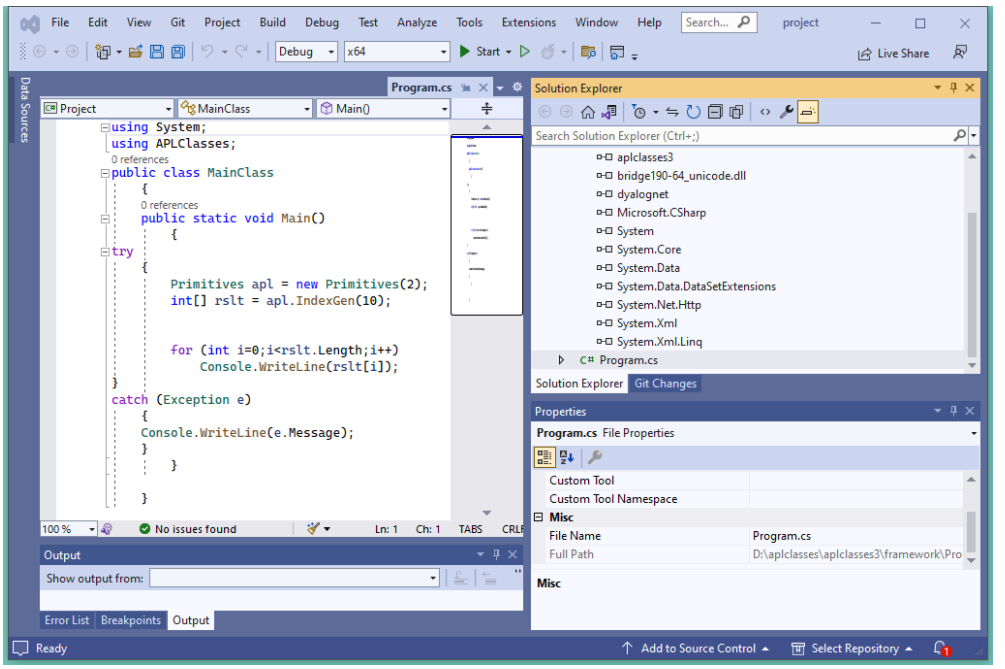


program.cs

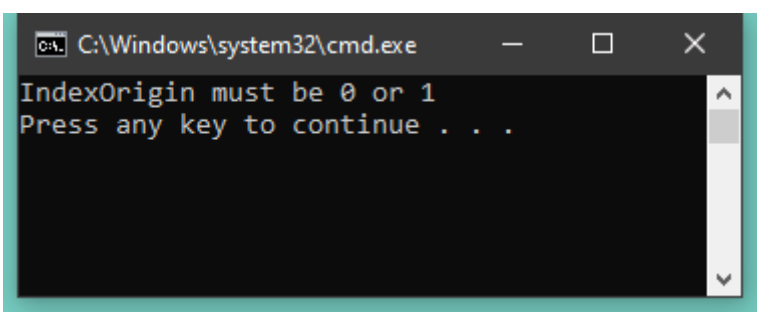
The following C# source, called `aplclasses2\Framework\program.cs`, contains code to catch the exception and to display the exception message.

```
using System;
using APLClasses;
public class MainClass
    {
    public static void Main()
        {
try
    {
        Primitives apl = new Primitives(2);
        int[] rslt = apl.IndexGen(10);

        for (int i=0;i<rslt.Length;i++)
            Console.WriteLine(rslt[i]);
}
catch (Exception e)
    {
    Console.WriteLine(e.Message);
    }
        }

    }
```

Using VS, open the solution file `d:\aplclasses\aplclasses3\Framework\project.sln` and view `program.cs`.



Click *Debug/Start Without debugging* (or press `Ctrl+F5`) to run the program. The results are shown in a console window.



6.8 Example 4

This example builds on Example 3 and illustrates how you can implement *constructor overloading*, by establishing several different constructor functions.

By way of an example, when a client application creates an instance of the `Primitives` class, we want to allow it to specify the value of `⎕IO` or the values of both `⎕IO` and `⎕ML`.

The simplest way to implement this is to have two public constructor functions `CTOR1` and `CTOR2`, which call a private constructor function `CTOR`.

`aplclasses4.dws` contains a new version of the `Primitives` class with these additions:

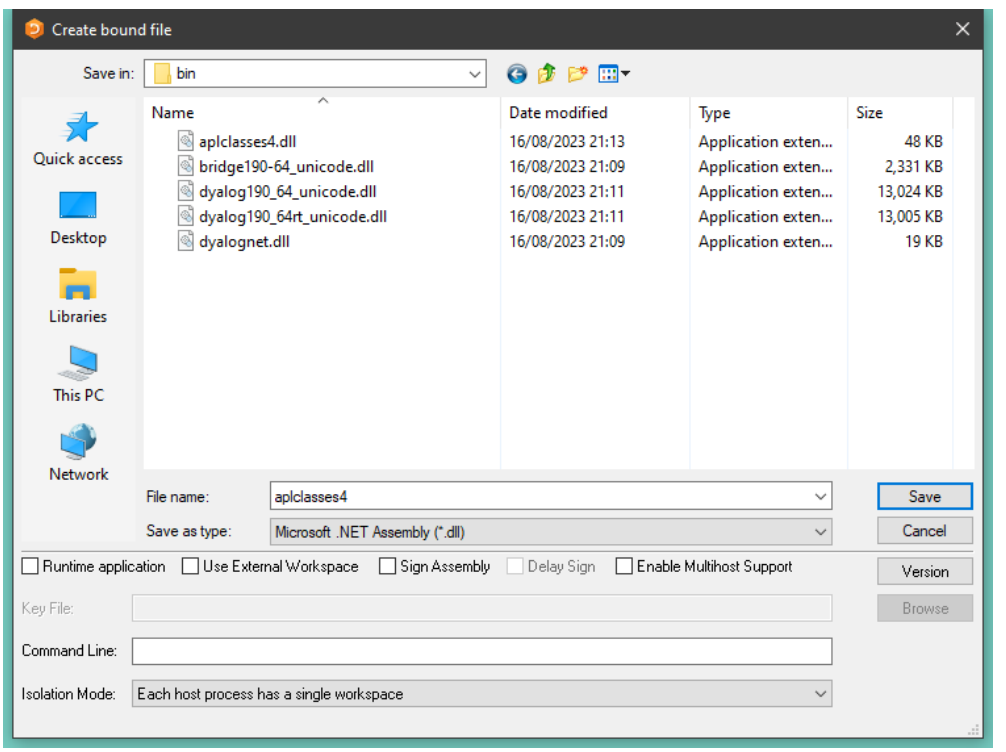
```
      ∇ CTOR1 IO
[1]     :Implements constructor
[2]     :Access public
[3]     :Signature CTOR1 Int32 IO
[4]     CTOR IO 0
      ∇

      ∇ CTOR2 IOML
[1]     :Implements constructor
[2]     :Access public
[3]     :Signature CTOR2 Int32 IO,Int32 ML
[4]     CTOR IOML
      ∇

      ∇ CTOR IOML;EX
[1]     IO ML←IOML
[2]     :If ~IO∊0 1
[3]         EX←⎕NEW ArgumentException,⊂⊂'IndexOrigin must
                                              be 0 or 1'
[4]         EX ⎕SIGNAL 90
[5]     :EndIf
[6]     :If ~ML∊0 1 2 3
[7]         EX←⎕NEW ArgumentException,⊂⊂'MigrationLevel
                                    must be 0, 1, 2 or 3'
[8]         EX ⎕SIGNAL 90
[9]     :EndIf
[10]    ⎕IO ⎕ML←IO ML
      ∇
```

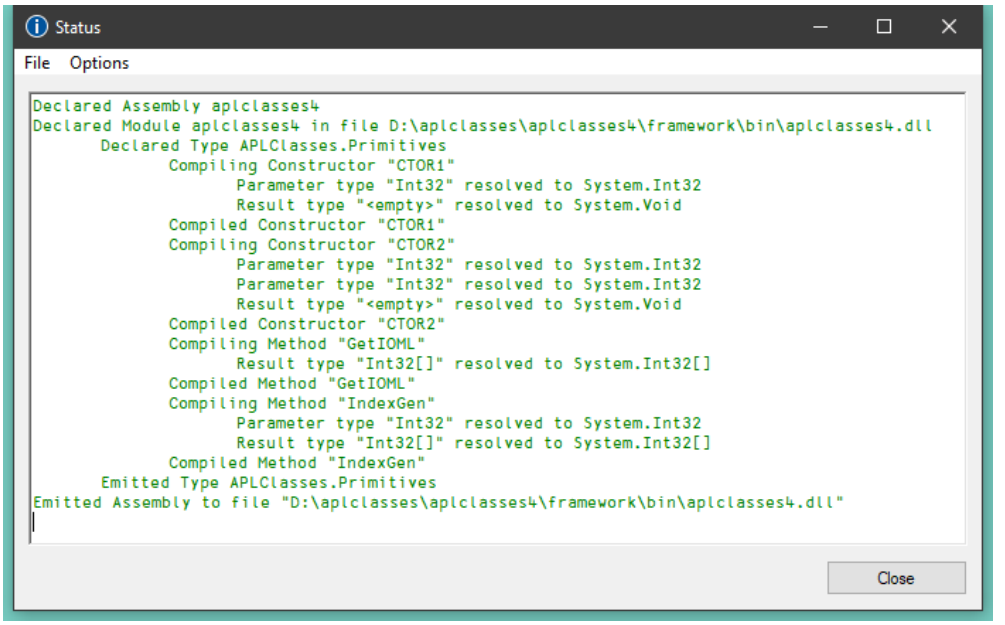
The `:Signature` statements for these three functions show that `CTOR1` is defined as a constructor that takes a single `Int32` parameter, `CTOR2` is defined as a constructor that takes two `Int32` parameters, and `CTOR` has no .NET Properties defined at all. Note that in .NET terms, `CTOR` is not a *Private Constructor*; it is simply an internal function that is invisible to the outside world.

Next, a function called `GetIOML` is defined and exported as a Public Method. It simply returns the current values of `⎕IO` and `⎕ML`.

```
     ∇ R←GetIOML
[1]   :Access public
[2]   :Signature Int32[]←GetIOML
[3]    R←⎕IO ⎕ML
      ∇
```

Load `aplclasses4.dws` and export a new version of `aplclasses.dll` as before.





program.cs

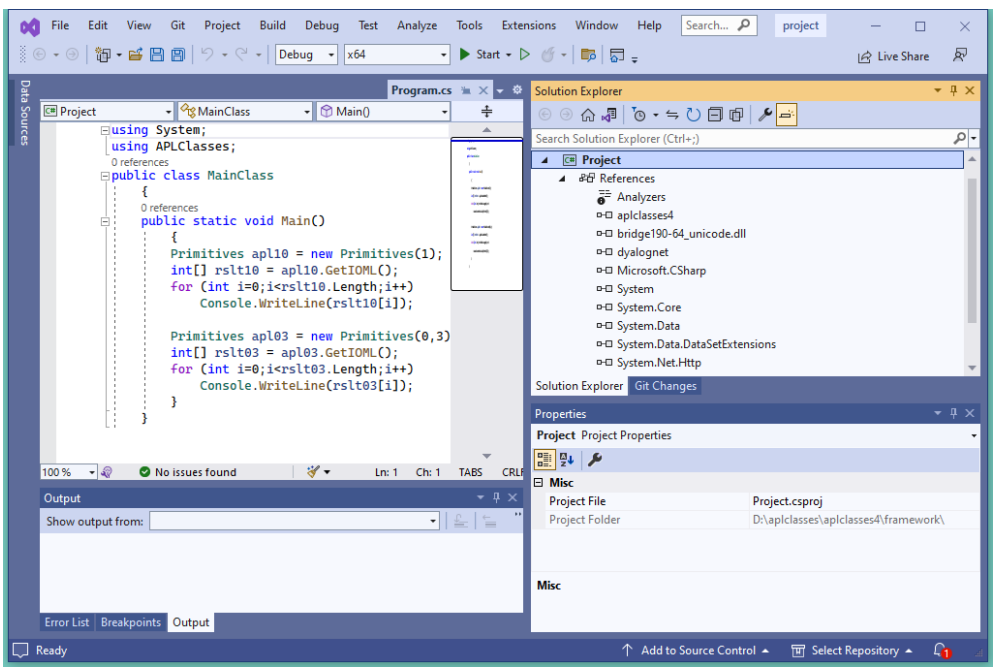
`aplclasses4\Framework\program.cs` contains code to invoke the two different constructor functions `CTOR1` and `CTOR2` :

```
using System;
using APLClasses;
public class MainClass
    {
    public static void Main()
        {
        Primitives apl10 = new Primitives(1);
        int[] rslt10 = apl10.GetIOML();
        for (int i=0;i<rslt10.Length;i++)
            Console.WriteLine(rslt10[i]);

        Primitives apl03 = new Primitives(0,3);
        int[] rslt03 = apl03.GetIOML();
        for (int i=0;i<rslt03.Length;i++)
            Console.WriteLine(rslt03[i]);
        }
    }
```

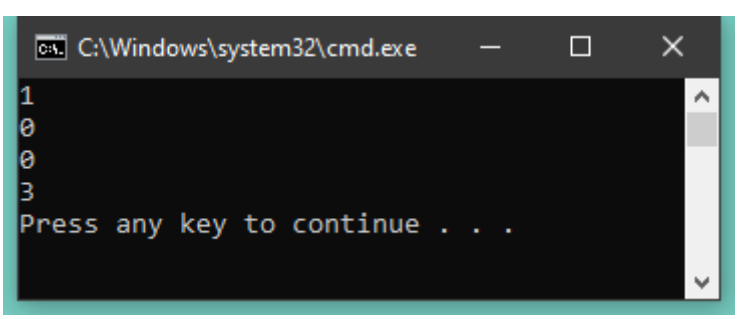
Here the code creates two instances of the Primitives class named `apl10` and `apl03`. The first is created with a constructor parameter of `(1)`; the second with a constructor parameter of `(0,3)`.

The C# compiler matches the first call with `CTOR1`, because `CTOR1` is defined to accept a single `Int32` parameter. The second call is matched to `CTOR2` because `CTOR2` is defined to accept two `Int32` parameters.

Using VS, open the solution file `d:\aplclasses\aplclasses4\Framework\project.sln` and view `program.cs`.



Click *Debug/Start Without debugging* (or press `Ctrl+F5`) to run the program. The results are shown in a console window.



6.9 Example 5

This example takes things a stage further and illustrates how you can implement *method overloading*.

In this example, the requirement is to export three different versions of the `IndexGen` method; one that takes a single number as an argument, one that takes two numbers, and a third that takes any number of numbers. These are represented by three functions named `IndexGen1`, `IndexGen2` and `IndexGen3` respectively. Because monadic `⍳` performs all of these operations, the three APL functions are in fact identical. However, their public interfaces, as defined in their `:Signature` statement, are all different.

The overloading is achieved by entering the same name for the exported method (`IndexGen`) in the box provided, for each of the three APL functions.

`aplclasses5.dws` contains a new version of the `Primitives` class with three different versions of `IndexGen` as shown below:

```
     ∇ R←IndexGen1 N
[1]   :Access public
[2]   :Signature Int32[]←IndexGen Int32 N
[3]    R←⍳N
     ∇
```

This is the version we have seen before. The method is defined to take a single argument of type `Int32`, and to return a 1-dimensional array (vector) of type `Int32`.

```
     ∇ R←IndexGen2 N
[1]   :Access public
[2]   :Signature Int32[][,]←IndexGen Int32 N1, Int32 N2
[3]    R←⍳N
     ∇
```

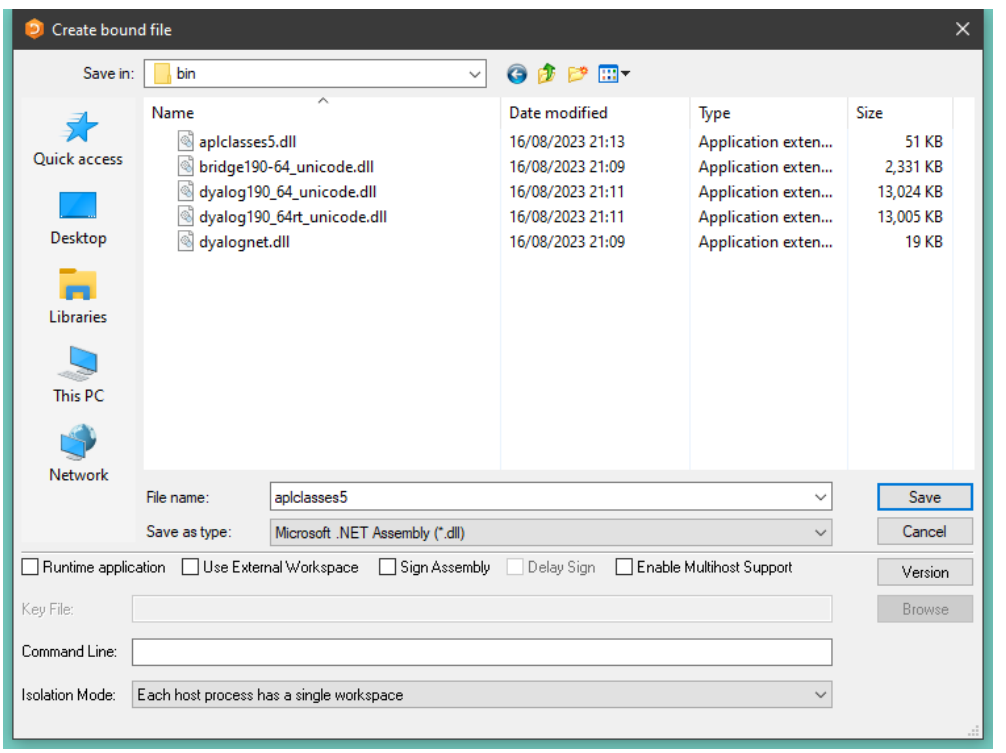
This version is defined to take two arguments of type `Int32`, and to return a 2-dimensional array, each of whose elements is a 1-dimensional array (vector) of type `Int32`.

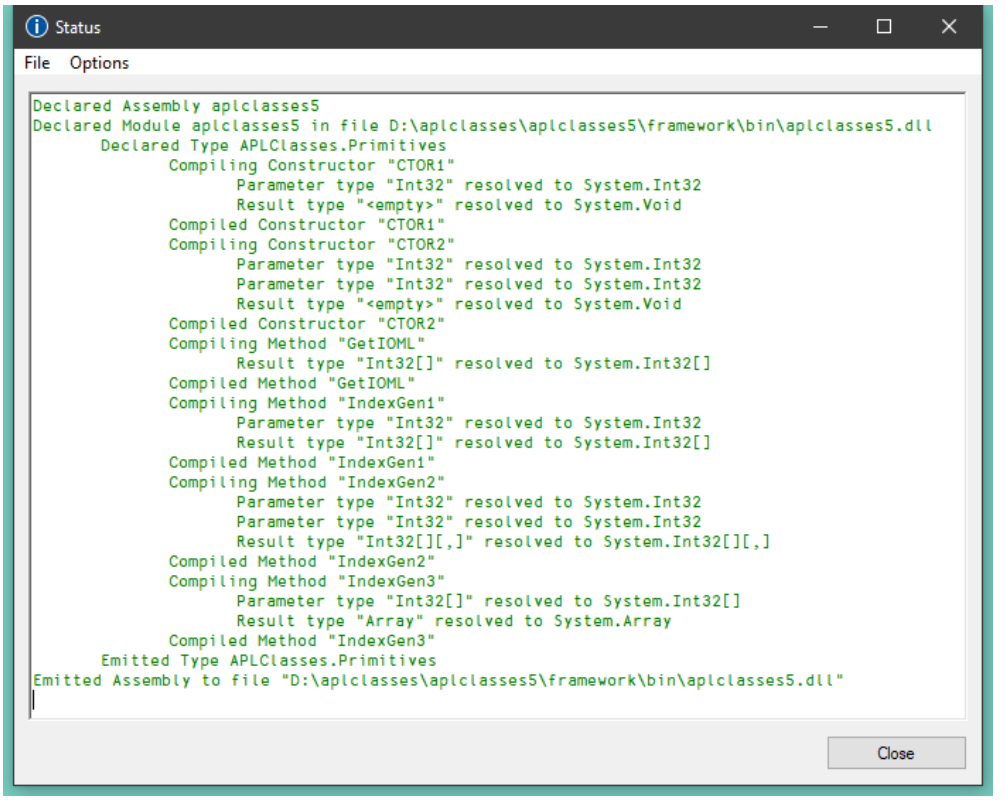
```
     ∇ R←IndexGen3 N
[1]   :Access public
[2]   :Signature Array←IndexGen Int32[] N
[3]    R←⍳N
     ∇
```

In principle, we could define 7 more different versions of the method, taking 3, 4, 5 etc. numeric parameters. Instead, this method is defined more generally, to take a single parameter that is a 1-dimemsional array (vector) of numbers, and to return a result of type `Array`. In practice we might use this version alone, but for a C# programmer, this is harder to use than the two other specific cases.

Notice also that all function use the same descriptive name, `<IndexGen>`.

Load `aplclasses5.dws` and export `aplclasses5.dll` as before.



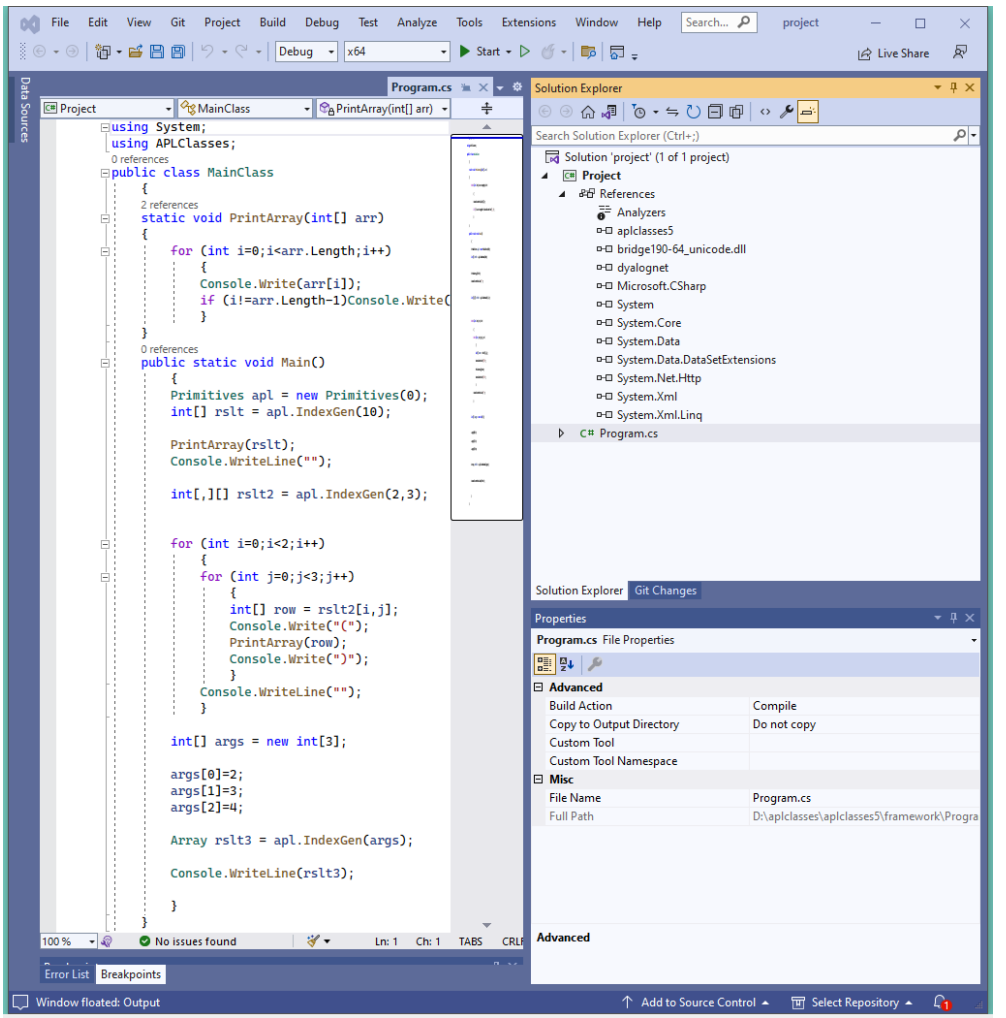


program.cs

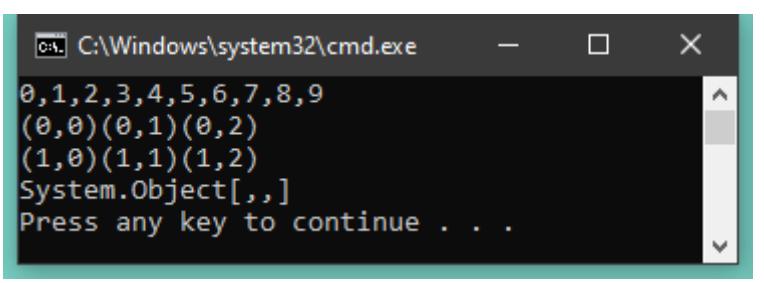
`samples\APLClasses\aplfns5.cs`contains code to invoke the three different variants of `IndexGen`, in the new `aplclasses.dll`. Notice that it uses a local sub-routine `PrintArray()`.

```
using System;
using APLClasses;
public class MainClass
      {
      static void PrintArray(int[] arr)
      {
            for (int i=0;i<arr.Length;i++)
                {
                Console.Write(arr[i]);
                if (i!=arr.Length-1)
                   Console.Write(",");
                }
      }

      public static void Main()
            {
            Primitives apl = new Primitives(0);
            int[] rslt = apl.IndexGen(10);
            PrintArray(rslt);
            Console.WriteLine("");

            int[,][] rslt2 = apl.IndexGen(2,3);
            for (int i=0;i<2;i++)
                     {
                     for (int j=0;j<3;j++)
                              {
                              int[] row = rslt2[i,j];
                              Console.Write("(");
                              PrintArray(row);
                              Console.Write(")");
                              }
            Console.WriteLine("");
                     }

            int[] args = new int[3];
            args[0]=2;
            args[1]=3;
            args[2]=4;
            Array rslt3 = apl.IndexGen(args);
            Console.WriteLine(rslt3);

      }
```

Using VS, open the solution file `d:\aplclasses\aplclasses2\Framework\project.sln` and view `program.cs`.



Click *Debug/Start Without debugging* (or press `Ctrl`+F5) to run the program. The results are shown in a console window.



It is possible for a function to have several `:Signature` statements. Given that our three functions perform exactly the same operation, it might have made more sense to use a single function:

```
     ∇ R←IndexGen1 N
[1]   :Access public
[2]   :Signature Int32[]←IndexGen Int32 N
[3]   :Signature Int32[][,]←IndexGen Int32 N1, Int32 N2
[4]   :Signature Array←IndexGen Int32[] N
[5]    R←⍳N
     ∇
```

6.10 Interfaces

Interfaces define additional sets of functionality that classes can implement; however, interfaces contain no implementation, except for static methods and static fields. An interface specifies a contract that a class implementing the interface must follow. Interfaces can contain shared (known as "static" in many compiled languages) or instance methods, shared fields, properties, and events. All interface members must be public. Interfaces cannot define constructors. The .NET runtime allows an interface to require that any class that implements it must also implement one or more other interfaces.

When you define a class, you list the interfaces which it supports following a colon after the class name. The value of `⎕USING` (possibly set by `:Using`) is used to locate Interface names.

If you specify that your class implements a certain `Interface`, you must provide all of the members (methods, properties, and so forth) defined for that `Interface`. However, some Interfaces are only marker Interfaces and do not actually specify any members.

An example is the `TemperatureControlCtl2` custom control described in Chapter 11, which derives from `System.Web.UI.Control`. The first line of this class definition reads:

```
:Class TemperatureConverterCtl2: System.Web.UI.Control,
    System.Web.UI.IPostBackDataHandler,
    System.Web.UI.IPostBackEventHandler
```

Following the colon, the first name is the base class. Following the (optional) base class name is the list of interfaces which are implemented. The `TemperatureControlCtl2` custom control implements two interfaces named `IPostBackDataHandler` and `IPostBackEventHandler`. These interfaces are required for a custom control that

intends to render the HTML for its own form elements in a Web page. These interfaces define certain methods that get called at the appropriate time by the page framework when a Web page is constructed for the browser. It is therefore essential that the class implements all the methods specified by the interface, even if they do nothing.

The base class, `System.Web.UI.Control`, defines an optional Interface called `INamingContainer`. A class based on `Control` that implements `INamingContainer` specifies that its child controls are to be assigned unique ID attributes within an entire application. This is a marker interface with no methods or properties defined for it.

See these examples in Chapter 11 for further details.

7 Dyalog APL and IIS

7.1 Introduction

Microsoft Internet Information Services (IIS) is a comprehensive Web Server software package that allows you to publish information on your Intranet, or on the World Wide Web. IIS is included with Professional and Server versions of all recent Windows operating systems; all you need add is a network connection to run your own Web site.

IIS includes Active Server Page (ASP) technology. The basic idea of ASP is to permit web pages to be created dynamically by the web server. An ASP file is a character file that contains a mixture of HTML and scripts. When IIS receives a request for an ASP file, it executes the server-side scripts contained in the file to build the Web page that is to be sent to the browser. In addition to server-side scripts, ASP files can contain HTML (including related client-side scripts) as well as calls to components that can perform a variety of tasks such as database lookup, calculations, and business logic.

Basically, each script inside an ASP page generates a stream of HTML. The server runs the scripts and assembles the resulting HTML into a single stream (Web page) that is sent to the browser.

ASP.NET is a new version of ASP and is based upon the Microsoft .NET Framework technology. It offers significantly better performance and a host of new features including support for *Web Services*.

7.2 IIS Installation Dependency

During installation, Dyalog registers itself with ASP.NET as an ASP.NET programming language. Among other things, this allows ASP.NET web pages to be written in Dyalog. The Dyalog installation program also registers the Dyalog asp.net sample applications as IIS *Virtual Directories*.

It is not practical for the Dyalog `setup.exe` to perform these tasks unless IIS and ASP.NET are already installed. Furthermore, unless IIS and ASP.NET are already installed and activated on the system, the Dyalog sub-directory `Samples/asp.net` will not even be copied onto the system, because the samples it contains would be inoperable.

If IIS is installed after Dyalog, it is necessary to de-install and then re-install Dyalog to enable the registration of Dyalog as an ASP.NET Programming language to occur, and for the `Samples/asp.net` sub-directory to be copied onto the system and the samples registered as IIS Virtual Directories.

7.3 IIS Applications, Virtual Directories, Application Pools

IIS supports the concept of an *Application*. An application is a logically separate service or web site. IIS can run any number of Applications concurrently. The files associated with an application are stored in a physical directory on disk, which is linked to an IIS *Virtual Directory*. The name of the Virtual Directory is the name of the Application or Web Site.

The Dyalog APL distribution contains a directory named `Dyalog\Samples\asp.net` and a set of sub-directories each of which contains a sample application.

During the installation of Dyalog APL, these are automatically registered as IIS Virtual Directories, under a common root. The name of the root begins `dyalog.net` followed by the Dyalog Version number, the edition (Unicode or classic), and the architecture (32-bit or 64-bit). For example, `dyalog.net.15.0.unicode.64` [7]. The name of the root application is referred to henceforth as `dyalog.net`.

IIS applications run in application pools. An application pool is a group of one or more URLs that are served by the same worker process or set of worker processes which are separate from the worker process that services another application pool. This mechanism isolates applications from one another, providing resilience should any one application fail.

Each `dyalog.net` application is associated with an application pool named Dyalog APL xx (.NET v4.0 Classic[8]), where xx is 32 or 64) which is created if required during installation.

When you want to run the Web Services and Web Page examples, you do so by specifying the URL `http://localhost/dyalog.net.xxxx/`

These samples can be easily found by selecting the *Documentation Centre* menu item from the Help menu on the Dyalog session, and scrolling down to the Tutorials section.

---

7 Versions of Dyalog APL prior to Version 11.0 created Virtual Directories under apl.net .

8 The term NET v4.0 Classic refers to the name of a standard application pool on which it is based, and has nothing to do with the Classic variant of Dyalog.

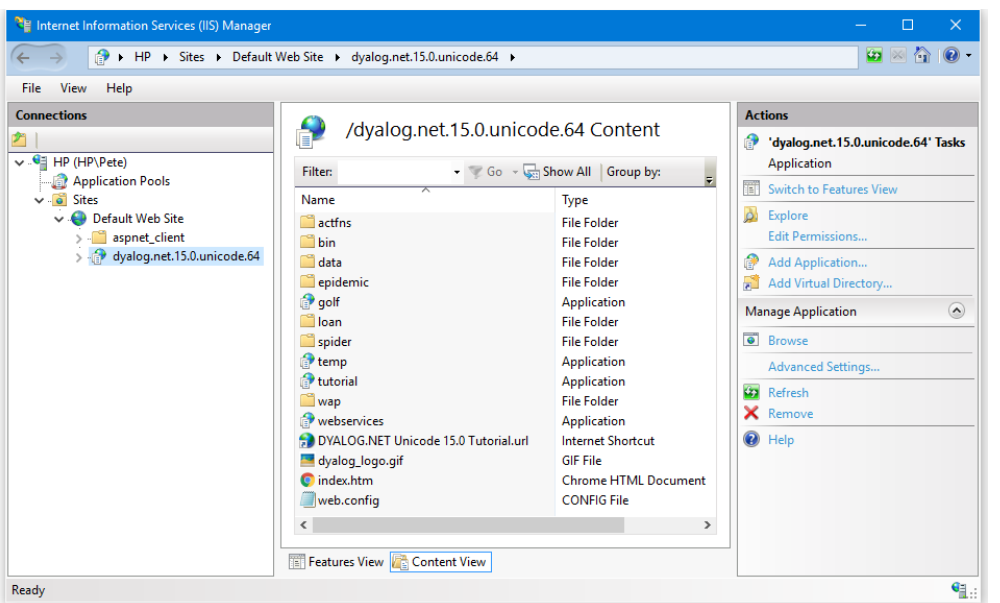
7.4 Internet Services Manager

As its name suggests, Internet Services Manager is a tool for managing IIS. If you are developing Web Pages and/or Web Services, you will be using this tool a lot, and it makes sense to add it as a shortcut on your desktop.

To do this, open *Control Panel*, then open *Administrative Tools*, right-click *Internet Information Services (IIS) Manager*, and select *Send To Desktop (create shortcut)*.

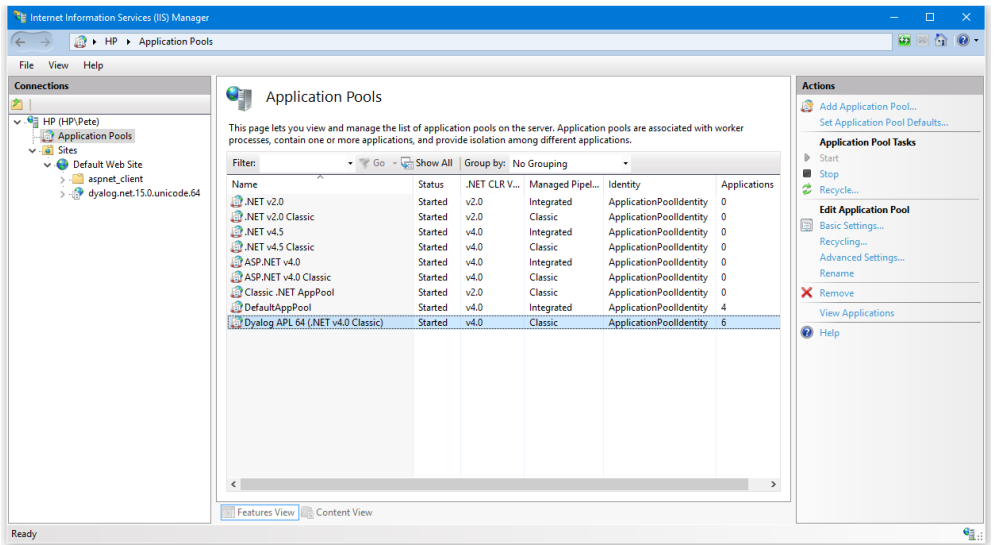
The *dyalog.net* Application

Following a successful installation of Dyalog APL, the `dyalog.net` Application should appear in Internet Services Manager as shown below.

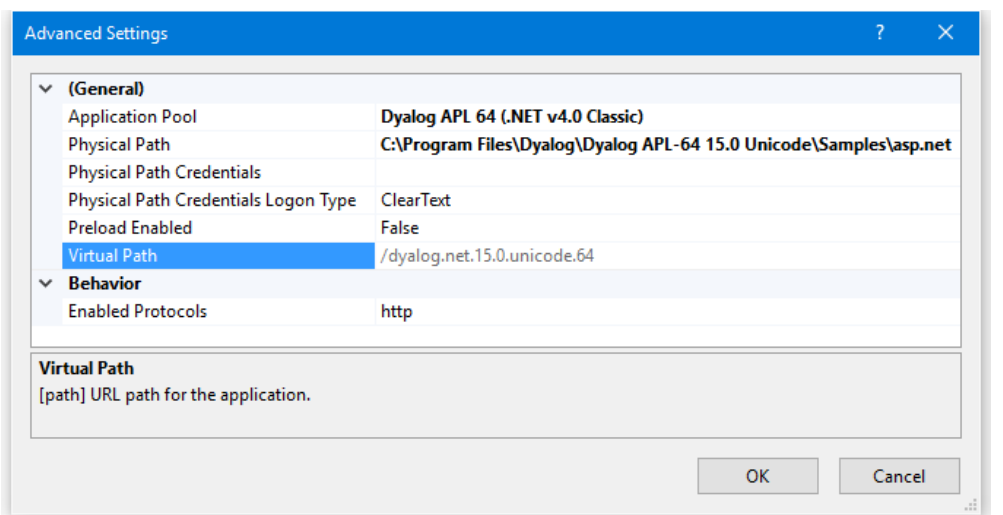


Note that the `golf`, `temp` and `webservices` sub-directories in the `dyalog.net` application represent separate IIS applications.

The dyalog Application Pool will appear in the list of Applications Pools as shown below.



The Advanced Settings of the `dyalog.net` Application are shown below.



8 Writing Web Services

8.1 Introduction

A Web Service can be thought of as a Remote Procedure Call. However, it is a remote procedure call that can be made over the Internet using character-based messages.

Web Services are implemented using Simple Object Access Protocol (SOAP), Extensible Mark-up Language (XML) and Hypertext Transfer Protocol (HTTP). Web Services do not require proprietary network protocols or software. Web Service calls and responses can successfully be transmitted over the Internet without the need to specially configure firewalls.

A Web Service is a class that may be called by any program running on the computer, any program running on a computer on the same LAN, or any program running on any computer on the internet.

Web Services are hosted (that is, executed) by ASP.NET running under Microsoft IIS. Any one Web Service sits on a single server computer and runs there under ASP.NET/IIS. The messages that invoke the Web Service, pass its arguments, and return its results, utilise standard HTTP/SOAP/XML protocols.

A Web Service consists of a single text script file, with the extension `.asmx`, in an IIS Virtual Directory on the server computer.

A Web Service may expose a number of Methods and Properties. Methods may be called *synchronously* (the calling process waits for the result) or *asynchronously* (the calling process invokes the method, continues for a bit, and then subsequently checks for the result of the previous call).

8.2 Web Service (.asmx) Scripts

Web Services may be written in a variety of languages, including `APLScript`, the scripting version of Dyalog APL. See *Section 5.1*.

The first statement in the script file declares the language and the name of the service. For example, the following statement declares a Dyalog APL Web Service named `GolfService`.

```
<%@ WebService Language="Dyalog" Class="GolfService" %>
```

Note that `Language="Dyalog"` is specifically connected to the Dyalog APL script compiler through the application's web.config file or through the global ASP.NET system file `Machine.config`. Note that versions of Dyalog prior to 11.0 used `Language="APL"`.

The syntax of this first line is common to all Web Services, regardless of the language in which they are written.

A Dyalog APL Web Service script starts with a `:Class` statement and ends with an `:EndClass` statement. These statements are directives used by the Dyalog APL script compiler and are specific to Dyalog APL.

The `:Class` statement declares the name of the Class (which must be the same as the name declared in the `WebService` statement) and the *Base Class* from which it inherits, which is normally `System.Web.Services.WebService`.

```
   :Class GolfService: System.Web.Services.WebService
```

Following the `:Class` statement, there may appear any number of APL expressions and function bodies. Following these there must be a `:EndClass` statement. Internal sub-classes (nested classes) may also be defined within the main `:Class ... :EndClass` block.

Because the functions usually take arguments and return results whose types must be known, the statement

```
   :Using System
```

must almost always appear **immediately** after the `:Class` statement to locate them.

8.3 Compilation

When the Web Service, specified by the `.asmx` file, is called **for the first time**, ASP.NET invokes the appropriate language compiler (in this case, the Dyalog Script compiler) whose job is to produce an Assembly that defines and describes a class. When the Web Service is used subsequently, the request is satisfied by creating and using an instance of the class. However, ASP.NET detects if the `.asmx` script has been modified, and recompiles it in this case.

The Dyalog Script compiler creates a `DLL` containing a workspace, which itself contains the Web Service class. The class contains all the functions, which are defined within the

script, together with any variables that were established by expressions in the script. A single function comprises all the statements enclosed within a pair of del (∇) symbols.

For example, the following script would define a class, instances of which would run using ⎕ML←2, containing a single function FOO and a variable X.

```
:Class MyClass
   ⎕ML←2
   X←10
   ∇ Z←FOO Y
     Z←Y+X
   ∇
:EndClass
```

Note that all expressions in the class script are executed by the script compiler when it creates the assembly. They are not executed when the Web Service is invoked.

If your script contains a ⎕CY statement, it will be executed by the compiler when establishing the class. This may be used to import functions from other workspaces and obviate the need to include them in the .asmx file.

8.4 Exporting Methods

Your Web Service will be of no use unless it exports at least one method. To export a function as a method, you must include declaration statements. Such declarations may be supplied anywhere within the function body, but it is recommended that they appear together as the first block of statements in your code. All declaration statements begin with the colon (:) character and the following declaration statements are supported:

```
:Access WebMethod
```

This statement causes the function to be exported as a method and **must** be present.

```
:Signature type ← fnname type name1, type name2, ...
```

This statement declares the data type of the result and the arguments of the method where type may specify any valid .NET type that is supported by Web Services. Note that the assignment arrow (←) is necessary if the function returns a result.

The declaration of each parameter of the method is separated from the next by a comma. Each name may be any ASCII character string. Note that names are optional.

Add1

```
∇ R←Add1 args
 :Access WebMethod
 :Signature Int32←Add Int32 arg1,Int32 arg2
  R←+/args
∇
```

The `Add1` function defined above is exported as a method named `Add`, that takes exactly (and only) two parameters of type `Int32` and returns a result of type `Int32`. Armed with this definition, which is recorded in the metadata associated with the class, the .NET Framework guarantees that the method will only be called in this way.

Add2

```
∇ R←Add2 arg
 :Access WebMethod
 :Signature Double←Add Double[] arg1
  R←+/arg
∇
```

The `Add2` function defined above is exported as a method that takes an array of `Double` and returns a result of type `Double`. Depending on the type of the arguments provided when the method is invoked, .NET and Dyalog will call `Add1` or `Add2` - or generate an exception if the argument does not match any of the signatures.

8.5 Web Service Data Types

In principle, Web Services are designed to support most, if not all, of the data types supported by the .NET Framework, and to support any new .NET classes that you choose to define.

In practice, the current set of data types supported by Web Services is somewhat restricted; in particular:

- Multi-dimensional arrays are not supported; only vectors.
- Arbitrary nested arrays are not supported.

However, despite these restrictions, it is possible to build effective Web Services, as you will see in the following examples.

8.6 Execution

When your Web Service (or Page) is invoked, ASP.NET requests an instance of the corresponding Class from the Assembly (DLL) that was created when it was compiled. The first time this happens for any Dyalog Web Service or Web Page, the Dyalog dynamic link library is loaded into the ASP.NET host process and the namespace corresponding to your Web Service class is `)COPYed` from the Assembly. The Dyalog dynamic link library then delivers an instance of this namespace to the client (calling) process. See *Section 12.1* for further details.

In general, every call on a method in a Web Service causes a new instance of the Web Server class to be created. If you need to maintain/update variables between calls, you need to write them to permanent storage.

If a client invokes a different Dyalog Web Service or Web Page, its class is `)COPY`ed from its Assembly into the workspace managed by the Dyalog dynamic link library. When you export a class, you can select one of three Isolation Modes:

1. Each host process has a single workspace
2. Each AppDomain has its own workspace
3. Each Assembly has its own workspace

In this context, "workspace" is synonymous with "Dyalog process": Each workspace is managed by a separate process running dyalog.dll. Under option 1, all Dyalog APL Web Services (and Web Pages) hosted by the IIS host process share the same workspace when they are invoked.

The isolation mode selected has implications for the way that you access and manage global resources such as component files. Finer isolation modes may be implemented in future versions of Dyalog.

8.7 Global.asax, Application and Session Objects

When a Web Service runs, it has access to the Application and Session objects. These are objects provided by ASP.NET through which you can manage the execution of the Web Service. ASP.NET creates an Application object when it first starts the Application, that is, when any client requests any Web Service or Web Page stored in the same IIS Virtual Directory. It also creates a Session object for each client process.

When the first request comes in for an ASP.NET application, ASP.NET checks for an optional file named `global.asax`, and if it is there it compiles it. The application's `global.asax` instance is then used to apply application events.

`global.asax` typically defines callback functions to be executed on the various `Application` and `Session` events, such as `Application_Start`, `Application_End`, `Session_Start`, `Session_End` and so forth.

Dyalog allows you to use APL functions in the `global.asax` script. This allows you to initialise your APL application when it is first invoked, and to close it down cleanly when it is terminated.

For example, you can use `global.asax` to tie a component file on start-up, and untie it on termination.

8.8 Sample Web Service: EG1

The first APLExample sample is supplied in `samples\asp.net\webservices\eg1.asmx` which is mapped via an IIS Virtual Directory to the URL:

```
http://localhost/dyalog.net.15.0.unicode.32/webservices/eg1.asmx
```

```
<%@ WebService Language="Dyalog" Class="APLExample" %>

:Class APLEXample: System.Web.Services.WebService
:Using System

 ∇ R←Add args
   :Access WebMethod
   :Signature Int32←Add Int32 arg1,Int32 arg2
   R←+/args
 ∇

:EndClass
```

The `Add` function defined above is exported as a method that takes exactly (and only) two parameters of type `Int32` and returns a result of type `Int32`.

Line [3] could in fact be coded as:

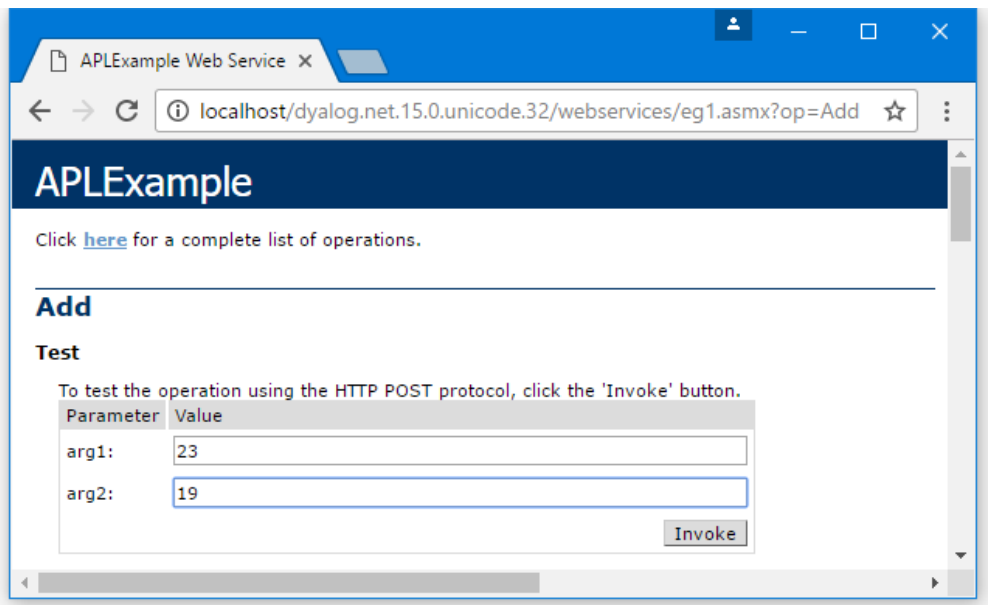
```
   R←args[1]+args[2]
```

because .NET guarantees that a client can only call the method by providing two 32-bit integers as parameters.

Testing APLExample from a Browser

If you connect to a URL that represents a Web Service, the browser displays a page that provides information about the service and the methods that it contains. In certain cases, but by no means all, the page also contains form fields that let you invoke a method from the browser.

The screen shot below shows the page displayed by Google Chrome when it is pointed at `eg1.asmx`. It shows that the Web Service is called `APLExample`, and that it exports a single method called `Add`. Furthermore, the `Add` method takes two parameters of type `int`, named *arg1* and *arg2*.



The following screen shot shows the result of entering the values 23 and 19 into the form fields and then pressing the Invoke button.

In this case, the method returns an `int` value 42.

It is important to understand what is happening here.

Accessed in this way from a browser, a Web Service appears to be behaving like a Web Server; this is not the case.

It is simply that the browser detects that the target URL is a Web Service, and invokes an ASP+ page named `DefaultSdlHelpGenerator.aspx` that inspects the compiled class and returns an HTML view of the Web service.

8.9 Sample Web Service: LoanService

The `LoanService` sample is supplied in `Dyalog\Samples\asp.net\Loan\Loan.asmx`, which is mapped via an IIS Virtual Directory to the URL:

```
http://localhost/dyalog.net.15.0.unicode.32/Loan/Loan.asmx
```

This `APLScript` sample defines a class named `LoanService` that is based upon `System.Web.Services.WebService`. The `LoanService` class defines a sub-class called `LoanResult` and a method called `CalcPayments`.

```
<%@ WebService Language="Dyalog" Class="LoanService" %>
:Class LoanService: System.Web.Services.WebService
:Using System
    :Class LoanResult
    :Access public
        :Field Public Int32[] Periods
        :Field Public Double[] InterestRates
        :Field Public Double[] Payments
    :EndClass

    ∇ R←CalcPayments X;LoanAmt;LenMax;LenMin;IntrMax;
                       IntrMin;PERIODS;INTEREST;NI;NM
[1]   :Access WebMethod
[2]   :Signature LoanResult←CalcPayments Int32 LoanAmt,
                         Int32 LenMax,Int32 LenMin,
                         Int32 IntrMax,Int32 IntrMin
[3]
[4]   ⍝ Calculates loan repayments
[5]   ⍝ Argument X specifies:
[6]   ⍝   LoanAmt     Loan amount
[7]   ⍝   LenMax      Maximum loan period
[8]   ⍝   LenMin      Minimum loan period
[9]   ⍝   IntrMax     Maximum interest rate
[10]  ⍝   IntrMin     Minimum interest rate
[11]
[12]  LoanAmt LenMax LenMin IntrMax IntrMin←X
[13]  R←⎕NEW LoanResult
[14]  R.Periods←¯1+LenMin+⍳1+LenMax-LenMin
[15]  R.InterestRates←0.5×¯1+(2×IntrMin)+⍳1+2×
                    IntrMax-IntrMin
[16]  NI←⍴INTEREST←R.InterestRates÷100×12
[17]  NM←⍴PERIODS←R.Periods×12
[18]  R.Payments←,(LoanAmt)×((NI,NM)⍴NM/INTEREST)÷
                    1-1÷(1+INTEREST)∘.*PERIODS
    ∇
:EndClass
```

`CalcPayments` takes five integer parameters (see comments for their descriptions) and returns an object of type `LoanResult`.

Note that the block of `APLScript` that defines the sub-class `LoanResult` must reside between the `:Class` and `:EndClass` statements of the main class, `LoanService`. You may define any number of internal classes in this way.

The `LoanResult` class is made up only of Fields and it does not export any methods or properties. Furthermore, there are no constructor methods defined and it relies solely on its default constructor that is inherited from its base class, `System.Object`. The default constructor is called without any parameters and in fact does nothing except to create an instance of the class. In particular, the fields it contains initialised to zero. In this case, that is sufficient, as all the fields will be filled in explicitly later.

```
    :Class LoanResult
    :Access public
        :Field Public  Int32[] Periods
        :Field Public Double[] InterestRates
        :Field Public Double[] Payments
    :EndClass
```

The `:Class` statement starts the definition of a new class and specifies its name. The `:EndClass` statement terminates it definition.

The three `:Field` declaration statements specify the names and data types of three public fields. The `Public` attributes are necessary to make the fields visible to methods within the `LoanService` class as a whole, as well as to external clients.

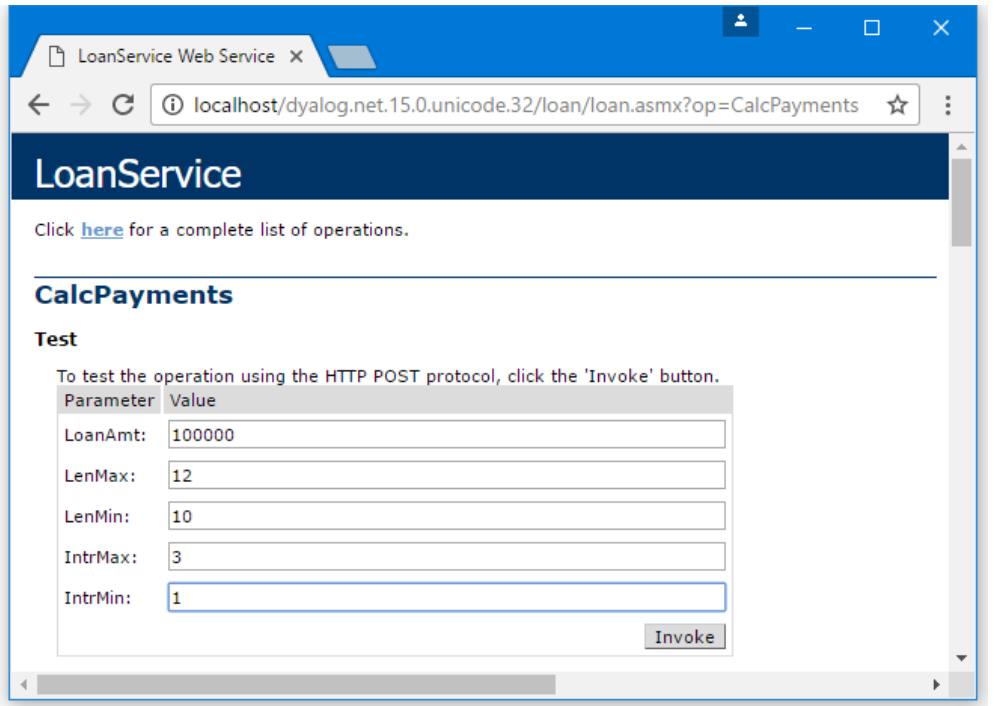
The `Periods` field is defined to be an array of integers; the `InterestRates` field an array of `Double`. Both these arrays are 1-dimensional, that is, vectors. These will contain the numbers of years, and the different interest rates, to which the repayments matrix applies.

Notice however that `Payments` is also defined to be 1-dimensional when in fact it is, more naturally, a 2-dimesional matrix. The reason for this is that, currently, Web Services do not support multi-dimensional arrays. This is a .NET restriction and not a Dyalog restriction.

`CalcPayments[13]` gets a new instance of the `LoanResult` class by doing `⎕NEW LoanResult`. It then assigns values to each of the three fields in lines `[14]`, `[15]` and `[18]`.

Testing LoanService from a Browser

Like the methods exported by the `APLEXample` Web Services described above, the `CalcPayments` method exported by `LoanService` is callable from a browser and the page that is displayed when you point a browser at it is shown below.



To test the `CalcPayments` method, you can enter numbers into the form fields in this page, as shown in the screen shot above, and then press the *Invoke* button. The result of the method is then displayed in a separate window as illustrated below.

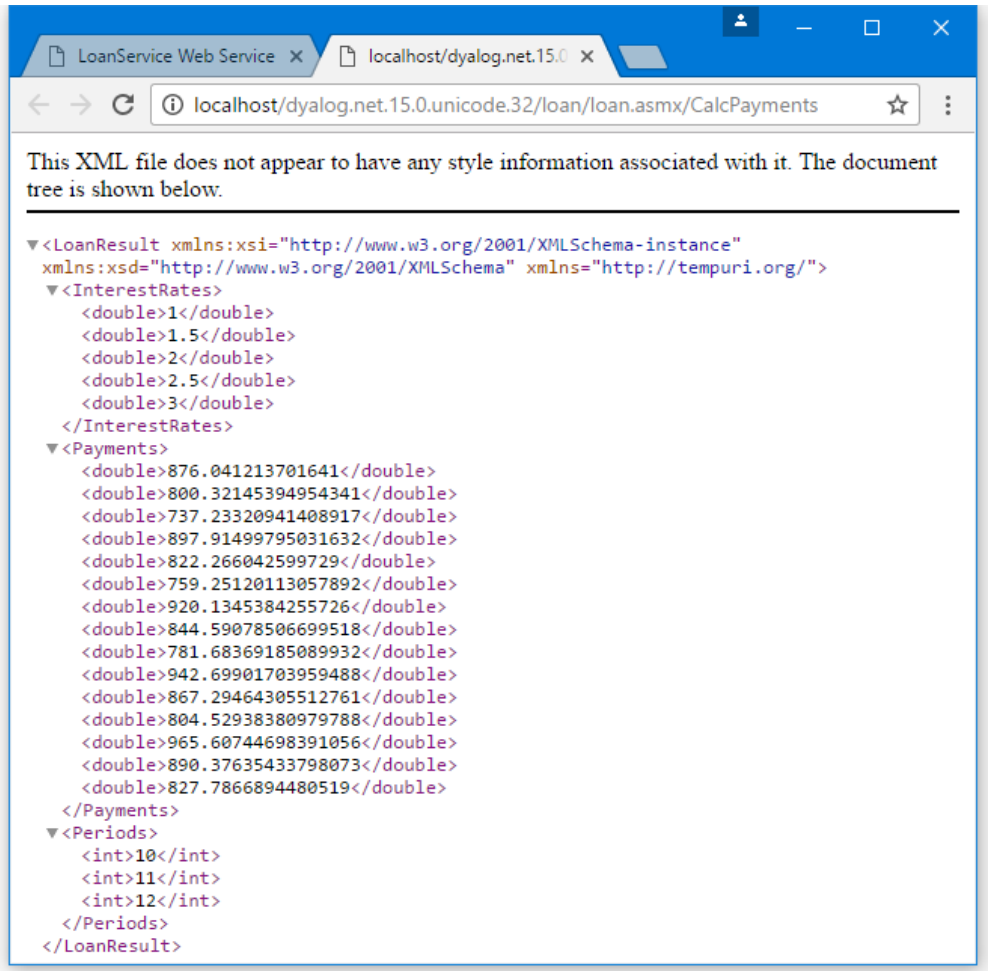
Notice that the result is described using XML, which is in fact the very language used to invoke a Web Service and return its result.

You can see that the result is of type `LoanResult`, and it contains 3 fields named `Payments`, `InterestRates` and `Periods`. This information was derived by our definition of the `LoanResult` class in the `APLScript` file.

As you can see, the `InterestRates` field shows that it contains a vector of floating-point values (`double`) from the minimum rate to the maximum rate that we specified on the input form. This time, the increment is 0.5.

Similarly, the `Payments` field contains the calculated repayment values.

Finally the `Periods` field, contains a vector of integers from the minimum period to the maximum period that we specified on the input form, in increments of 1.



8.10 Sample Web Service: GolfService

`GolfService` is an example Web Service that resides in the directory `samples\asp.net\Golf` and is associated with the IIS Virtual Directory `dyalog.net/Golf`. This example makes extensive use of internal classes to define data structures that are appropriate for a client application, such as C# or VB.

The directory contains a `global.asax` script, which is used to initialise the application.

The Golf Web Service example manages the reservation of tee-times at golf courses. All the data is held in a component file called `GolfData.dcf`. This file may be initialised using the function `Golf.INITFILE` in the workspace

`samples\asp.net\webservices\webservices.dws`. You may need to alter the file path first.

Each golf course managed by the application has a unique code (integer) and a name (string). This is handled by defining a class (structure) called `GolfCourse` with two fields, `Code` and `Name`.

`GolfService` provides 3 methods:

```
GetCourses()
```

Returns a list of Golf Courses (CourseCode and CourseName). The result of this method is an array of `GolfCourse` objects.

```
GetStartingSheet(CourseCode,Date)
```

Returns the starting sheet for a specified golf course on a given day. A starting sheet is a list of starting times with a list of the golfers booked to start their round at that time. The result of this method is a `StartingSheet` object.

```
MakeBooking(CourseCode,TeeTime,GimmeNearest, Name1,Name2,Name3,Name4))
```

Requests a tee reservation at the course specified by `CourseCode`. `TeeTime` is a `DateTime` object that specifies the requested date and time. `GimmeNearest` is `Boolean`. If 1, requests the nearest tee-time to that specified; if 0, requests only the specified tee-time. Name1-4 are strings specifying up to 4 players. Note that all parameters are required. The result of this method is a `Booking` object.

GolfService: Global.asax

```
<script language="Dyalog" runat=server>

∇ Application_Start;GOLFID
  :Access Public
  GOLFID←'c:\Dyalog\samples\asp.net\golf\GolfData' ⎕FTIE 0[^1]
  Application[⊂'GOLFID']←GOLFID
∇

∇ Application_End;GOLFID
  :Access Public
  :Trap 6
      GOLFID←Application[⊂'GOLFID']
      ⎕FUNTIE GOLFID
  :EndTrap
∇
</script>
```

The `Application_Start` function is called when the `GolfService` Web Service is invoked for the first time. It ties the `GolfData` component file then stores the tie number in a new Item called `GOLFID` in the Application object. This item is then subsequently available to methods in the `GolfService` for the duration of the application.

The `Application_End` function is invoked when the `GolfService` Web Service terminates. It unties the `GolfData` component file.

This example may be considered slightly weak in that the location of the data file is hard-coded in the application's `Global.asax` file. An alternative is to store this information in the `<appsettings>` section of the appropriate `web.config` file or in the global `machine.config` file. This is preferable if the resource (in this case a file name) is to be accessed from more than one script. For further information on ASP.NET *config* files, see the documentation for the .NET Framework SDK.

Note that the `GolfData` file may be initialised using the function `Golf.INITFILE` in the `samples\asp.net\webservices\webservices.dws` workspace. The function will prompt you for the path of the file, initialize it and update the `Global.asax` file accordingly.

GolfService: GolfCourse class

The `GolfCourse` class is effectively a structure with two fields named `Code` and `Name`. `Code` is an integer code that provides a shorthand way to refer to a specific golf course; `Name` is a `String` containing its full name.

```
:Class GolfCourse
    :Access Public
    :Field Public Int32 Code
    :Field Public String Name

    ∇ ctor args
     :Implements Constructor
     :Access public
     :Signature fn Int32, String
      Code Name←args
    ∇
    ∇ ctor_def
     :Implements Constructor
     :Access public
      ctor ¯1 ''
    ∇
:EndClass
```

The `GolfCourse` class provides two constructors. The first, named `ctor_def`, takes no arguments and therefore overrides the default constructor that is inherited from `System.Object`. `ctor_def` calls `ctor` to initialise the instance with a Code of `¯1` and an empty Name.

The constructor named `ctor` accepts two parameters named `CourseCode` (an integer) and `CourseName` (a string), and simply assigns these values into the corresponding fields.

Therefore, valid ways to create an instance of a `GolfCourse` are:

```
      GC←⎕NEW GolfCourse
      GC.(Code Name)←1 'St Andrews'
```

Or, more simply

```
      GC←⎕NEW GolfCourse (1 'St Andrews')
```

Note that the names of the constructor functions are not visible outside the class. Constructors are identified by their signatures (basically, the `:Implements Constructor` statement) and not by their names.

GolfService: Slot class

The `Slot` class is effectively a structure with two fields named `Time` and `Players`. `Time` is a `DateTime` object that represents a time that can be reserved on the first tee. `Players` is an array of (up to 4) strings that contains the names of the golfers who have reserved to start their round of golf at that time.

```
:Class Slot
    :Access Public
    :Field Public DateTime Time
    :Field Public String[] Players

    ∇ ctor1 arg
     :Implements Constructor
     :Access public
     :Signature fn DateTime
      Time←arg
      Players← 0⍴⊂''
    ∇
    ∇ ctor2 args
     :Implements Constructor
     :Access public
     :Signature fn DateTime, String[]
      Time Players←args
    ∇
    ∇ ctor_def
     :Implements Constructor
     :Access public
    ∇
:EndClass
```

This class provides two constructor functions named `ctor1` and `ctor2`. However, for internal reasons, if a class defines any constructor functions, it is currently necessary to provide a dummy default constructor (the form of the constructor that takes no parameters); hence `ctor_def`.

The constructor `ctor1` accepts a single `DateTime` parameter, which it assigns to the Time, field, and initialises the `Players` field to an empty array.

The constructor `ctor2` accepts two arguments, a specified tee time, and an array of strings that contains golfers' names. It assigns these parameters to `Time` and `Players` respectively.

GolfService: Booking class

The `Booking` class represents the result of the `MakeBooking` method. It contains 4 fields named `OK`, `Course`, `TeeTime` and `Message`.

`OK` is `Boolean` and indicates whether or not the attempt to make a reservation was successful. If `OK` is false (0), the `Message` field (a string) indicates the reason for failure.

If `OK` is true (1) the `Course` field contains an instance of a `GolfCourse` object, and the `TeeTime` field contains an instance of a `Slot` object. Together, these objects identify the reserved golf course and starting slot. The latter specifies both the starting time, and the names of all the golfers who have been allocated that starting time and who will therefore play together.

```
:Class Booking
    :Access Public
    :Field Public Boolean OK
    :Field Public GolfCourse Course
    :Field Public Slot TeeTime
    :Field Public String Message

    ∇ ctor args
     :Implements Constructor
     :Access public
     :Signature fn Boolean, GolfCourse, Slot, String
      OK Course TeeTime Message←args
    ∇
    ∇ ctor_def
     :Access public
     :Implements Constructor
    ∇
:EndClass
```

This class provides a single constructor method, which must be called with values for all four fields.

GolfService: StartingSheet class

The `StartingSheet` class represents the result of the `GetStartingSheet` method. It contains 5 fields named `OK`, `Course`, `Date`, `Slots` and `Message`. `OK` is `Boolean` and indicates whether or not a starting sheet is available for the specified course and date.

If `OK` is false (0), the `Message` field (a string) indicates the reason for failure.

If `OK` is true (1) the `Course` field contains an instance of a `GolfCourse` object, the `Date` field contains the date in question, and the `Slots` field contains an array of `Slot` objects. Each `Slot` object specifies a starting time and the names of golfers who are booked to play at that time.

```
:Class StartingSheet
    :Access Public
    :Field Public Boolean OK
    :Field Public GolfCourse Course
    :Field Public DateTime Date
    :Field Public Slot[] Slots
    :Field Public String Message

    ∇ ctor args
     :Implements Constructor
     :Access public
     :Signature fn Boolean, GolfCourse, DateTime
      OK Course Date←args
    ∇

    ∇ ctor_def
     :Implements Constructor
     :Access public
    ∇
:EndClass
```

Like the `Booking` class, the `StartingSheet` class provides a single constructor method. In this case, the constructor is called with values for just 3 of the fields; the values of the other fields are expected to be assigned later.

GolfService: GetCourses function

```
     ∇ R←GetCourses;COURSECODES;COURSES;INDEX;GOLFID
[1]   ⍝
[2]   :Access WebMethod
[3]   :Signature GolfCourse[]←fn
[4]
[5]    GOLFID←Application['GOLFID']
[6]    COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
[7]    R←⎕NEW¨GolfCourse,¨⊂¨↓⍉↑COURSECODES COURSES
     ∇
```

The `GetCourses` function retrieves the tie number of the `GolfData` component file from the `Application` object and reads its first component.

The function then creates a `GolfCourse` object for each of the courses recorded on the file, and returns the array of `GolfCourse` objects as its result.

GolfService: GetStartingSheet function

The `GetStartingSheet` function retrieves the tie number of the `GolfData` component file from the `Application` object and reads its first component. Line [10] creates an instance of a `StartingSheet` object and uses it to initialise the result `R`. The value of the `OK` field is set to zero to indicate failure.

It then validates the requested `CourseCode`. If invalid, it simply sets the `Message` field in the result and returns it. Similarly, it checks to see if there is a starting sheet on file for the requested date. If not, it sets the `Message` field to indicate this, and returns.

Note that line [15] extracts the `Year`, `Month` and `Day` properties from the requested tee time, a `DateTime` object, and converts them to an IDN. This is used to index the component containing the starting sheet for that day.

```
     ∇ R←GetStartingSheet ARGS;CODE;COURSE;DATE;GOLFID;
       COURSECODES;COURSES;INDEX;COURSEI;IDN;DATES;COMPS;
       IDATE;TEETIMES;GOLFERS;I;T
[1]    ⍝
[2]    :Access WebMethod
[3]    :Signature StartingSheet←fn Int32 CCode,
                                   DateTime Date
[4]
[5]    CODE DATE←ARGS
[6]    GOLFID←Application[⊂'GOLFID']
[7]    COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
[8]    COURSEI←COURSECODES⍳CODE
[9]    COURSE←⎕NEW GolfCourse (CODE(COURSEI⊃COURSES,⊂''))
[10]   R←⎕NEW StartingSheet (0 COURSE DATE)
[11]   :If COURSEI>⍴COURSECODES
[12]       R.Message←'Invalid course code'
[13]       :Return
[14]   :EndIf
[15]   IDN←2 ⎕NQ'.' 'DateToIDN',DATE.(Year Month Day)
[16]   DATES COMPS←⎕FREAD GOLFID,COURSEI⊃INDEX
[17]   IDATE←DATES⍳IDN
[18]   :If IDATE>⍴DATES
[19]       R.Message←'No Starting Sheet available'
[20]       :Return
[21]   :EndIf
[22]   TEETIMES GOLFERS←⎕FREAD GOLFID,IDATE⊃COMPS
[23]   R.OK←1
[24]   T←⎕NEW¨'DateTime',¨⊂¨(⊂DATE.(Year Month Day)),¨
                 3↑¨↓[1]24 60⊤TEETIMES
[25]   R.Slots←⎕NEW¨'Slot',¨⊂¨T,∘⊂¨↓GOLFERS
     ∇
```

Line[23] sets the `OK` field of the result to 1 (success).

Line[24] converts the stored tee times (in minutes) to `DateTime` objects.

Line[25] combines the tee times and golfers into a vector of 2-element arrays, and creates a `Slot` object for each of them. The result is assigned to the `Slots` field of the result `R`.

GolfService: MakeBooking function

The `MakeBooking` function checks that the requested tee-time is available, for the specified number of players and updates the starting sheet accordingly. The result of the function is a `Booking` object.

`MakeBooking` first retrieves the tie number of the `GolfData` component file from the Application object and reads its first component.

Lines[13 14] create instances of `GolfCourse` and `Slot` objects, which at this stage are not validated. Line[15] then initialises the result `R`, a `Booking` object, which includes these instances. At this stage, `R.OK` is 0 indicating failure.

Line[16] validates the requested `CourseCode`, and, if invalid, simply sets `R.Message` and returns.

Similarly, lines [20 23] check that the requested tee time is within the next 30 days from now. If not, the function assigns the appropriate error message to `R.Message` and returns. Note that these two statements employ the APL primitive function `>` (rather that the `op_GreaterThan` method) to compare the requested tee time (a `DateTime` object) with a new `DateTime` object that represents *now* and *now+30 days* respectively.

Notice that line[24] uses the `AddDays` method to create a new `DateTime` object that represents *now + 30 days*. An alternative expression, to get *now+30* days is:

```
      TEETIME.Now+⍬NEW TimeSpan (30 0 0 0)
```

Lines[28-47] are concerned with retrieving the appropriate component from the file, initialising it or re-using an old one, if it is not present. Each component represents the starting sheet for a particular course on a particular day.

Lines[48-63] check whether or not the requested slot is available (for the specified number of golfers). If not it returns an error message as before or, if `GimmeNearest` is 1 (true), it attempts to allocate the slot closest to the requested time.

If an appropriate slot is found, Lines[72 73] update the `Slot` object with the assigned time and names of the golfers. Line[74] then inserts the modified `Slot` object into the result, and sets the `OK` field to 1 (true) to indicate success.

```
    ∇ R←MakeBooking ARGS;CODE;COURSE;SLOT;TEETIME;GOLFID;
                    COURSECODES;COURSES;INDEX;COURSEI;IDN;
                    DATES;COMPS;IDATE;TEETIMES;GOLFERS;
                    OLD;COMP;HOURS;MINUTES;NEAREST;TIME;
                    NAMES;FREE;FREETIMES;I;J;DIFF
[1]  ⍝
[2]   :Access WebMethod
[3]   :Signature Booking←Int32 CourseCode,
                         DateTime TeeTime,
                         Boolean GimmeNearest,
                         String Name1,
                         String Name2,
                         String Name3,
                         String Name4
[4]
[5]
[6]  ⍝ If GimmeNearest=0, books (or fails) for specified time
[7]  ⍝ If GimmeNearest=1, books (or fails) for nearest to
                                          specified time
[8]
[9]    CODE TEETIME NEAREST←3↑ARGS
[10]   GOLFID←Application[⊂'GOLFID']
[11]   COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
[12]   COURSEI←COURSECODES⍳CODE
[13]   COURSE←⎕NEW GolfCourse,⊂CODE(COURSEI⊃COURSES,⊂'')
[14]   SLOT←⎕NEW Slot TEETIME
[15]   R←⎕NEW Booking (0 COURSE SLOT '')
[16]   :If COURSEI>⍴COURSECODES
[17]       R.Message←'Invalid course code'
[18]       :Return
[19]   :EndIf
[20]   :If TEETIME.Now>TEETIME
[21]       R.Message←'Requested tee-time is in the past'
[22]       :Return
[23]   :EndIf
[24]   :If  TEETIME>TEETIME.Now.AddDays 30
[25]       R.Message←'Requested tee-time is more than
                     30 days from now'
[26]       :Return
[27]   :EndIf
[28]   IDN←2 ⎕NQ'.' 'DateToIDN',TEETIME.(Year Month Day)
[29]   DATES COMPS←⎕FREAD GOLFID,COURSEI⊃INDEX
[30]   IDATE←DATES⍳IDN
[31]   :If IDATE>⍴DATES
```

```
[32]       TEETIMES←(60×7)+10×¯1+⍳1+8×6
                   ⍝ 10 minute intervals, 07:00 to 15:00
[33]       GOLFERS←((⍴TEETIMES),4)⍴⊂''
                   ⍝ up to 4 golfers allowed per tee time
[34]       :If 0=OLD←⊃(DATES<
                  2 ⎕NQ'.' 'DateToIDN',3↑⎕TS)/⍳⍴DATES
[35]           COMP←(TEETIMES GOLFERS)⎕FAPPEND GOLFID
[36]           DATES,←IDN
[37]           COMPS,←COMP
[38]           (DATES COMPS)⎕FREPLACE GOLFID,COURSEI⊃INDEX
[39]
```

```
          :Else
[40]          DATES[OLD]←IDN
[41]          (TEETIMES GOLFERS)⎕FREPLACE
                                GOLFID,COMP←OLD⊃COMPS
[42]          DATES COMPS ⎕FREPLACE GOLFID,COURSEI⊃INDEX
[43]      :EndIf
[44]  :Else
[45]      COMP←IDATE⊃COMPS
[46]      TEETIMES GOLFERS←⎕FREAD GOLFID COMP
[47]  :EndIf
[48]  HOURS MINUTES←TEETIME.(Hour Minute)
[49]  NAMES←(3↓ARGS)~⊂''
[50]  TIME←60⊥HOURS MINUTES
[51]  TIME←10×⌊0.5+TIME÷10 ⍝ Round to nearest
                           10-minute interval
[52]  :If ~NEAREST
[53]      I←TEETIMES⍳TIME
[54]      :If I>⍴TEETIMES
[55]      :OrIf (⍴NAMES)>⊃,/+/0=⍴¨GOLFERS[I;]
[56]          R.Message←'Not available'
[57]          :Return
[58]      :EndIf
[59]  :Else
[60]      :If ~∨/FREE←(⍴NAMES)≤⊃,/+/0=⍴¨GOLFERS
[61]          R.Message←'Not available'
[62]          :Return
[63]      :EndIf
[64]      FREETIMES←(FREE×TEETIMES)+32767×~FREE
[65]      DIFF←|FREETIMES-TIME
[66]      I←DIFF⍳⌊/DIFF
[67]  :EndIf
[68]  J←(⊃,/0=⍴¨GOLFERS[I;])/⍳4
[69]  GOLFERS[I;(⍴NAMES)↑J]←NAMES
[70]  (TEETIMES GOLFERS)⎕FREPLACE GOLFID COMP
[71]  TEETIME←⎕NEW DateTime,⊂TEETIME.(Year Month Day),
                                3↑24 60⊤I⊃TEETIMES
[72]  SLOT.Time←TEETIME
[73]  SLOT.Players←(⊃,/0<⍴¨GOLFERS[I;])/GOLFERS[I;]
[74]  R.(OK TeeTime)←1 SLOT
  ∇
```

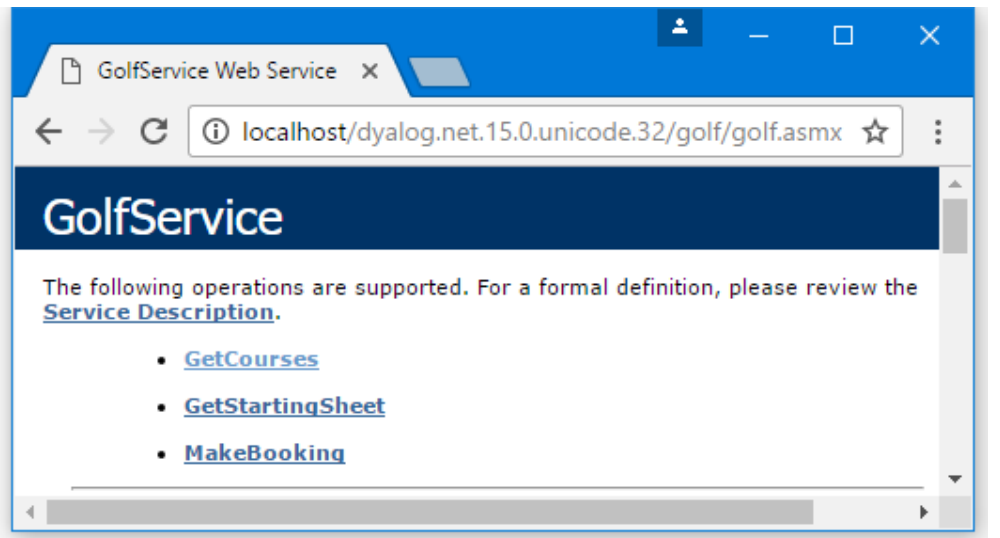
Testing GolfService from a Browser

If you point your browser at the URL:

```
http://localhost/dyalog.net.15.0.unicode.32/Golf/Golf.asmx
```

`GolfService` will be compiled and ASP.NET will fabricate a page about it for the browser to display as shown below.

The three methods exposed by `GolfService` are listed.

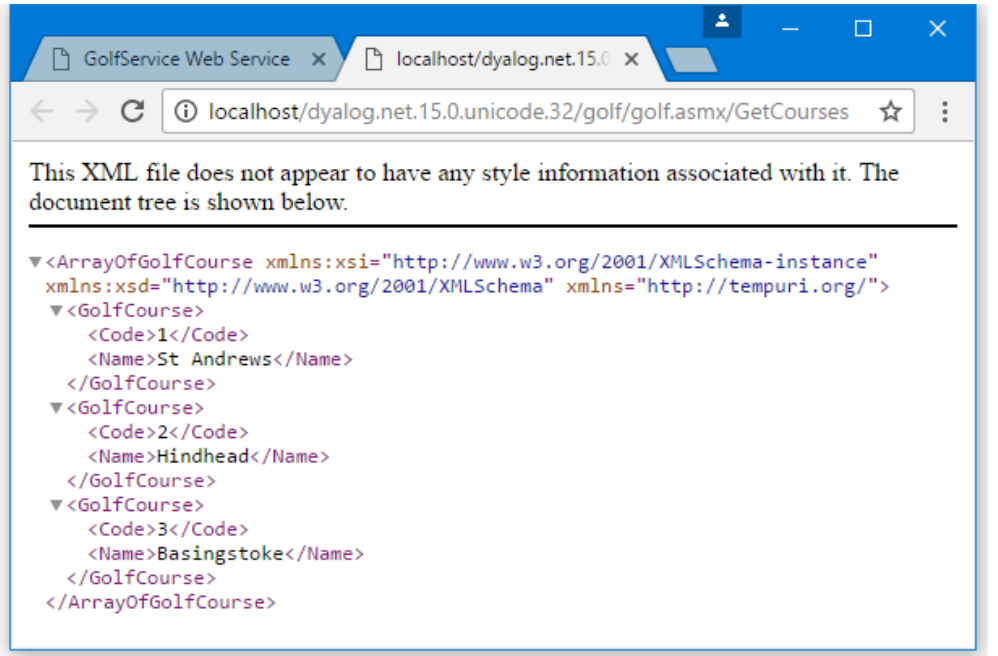


Invoking the `GetCourses` method generates the following output.

Notice that the data type of the result is `ArrayOfGolfCourse`, and the data type of each element of the result is `GolfCourse`. Furthermore, the public fields defined for the `GolfCourse` object are clearly named.

All this information is derived from the declarations in the `Golf.asmx` script.

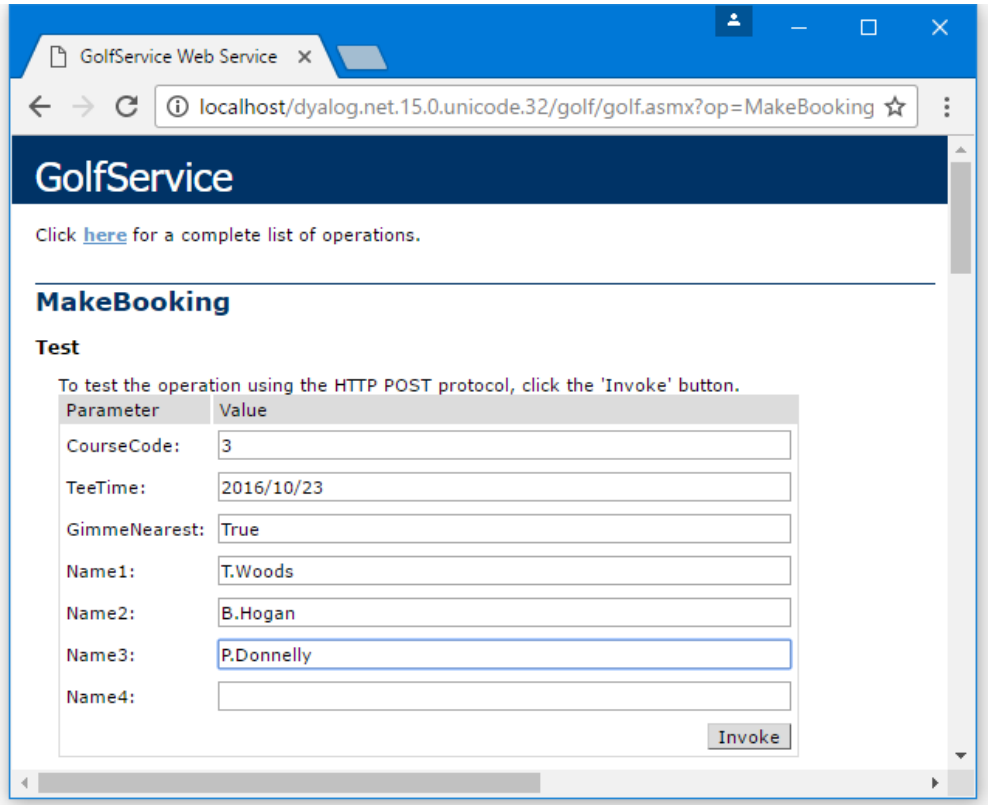
As supplied, the `GolfData` component file contains only 3 golf courses as shown below.



ASP.NET generates a Form containing fields that allow the user to invoke the `MakeBookings` method as shown below.

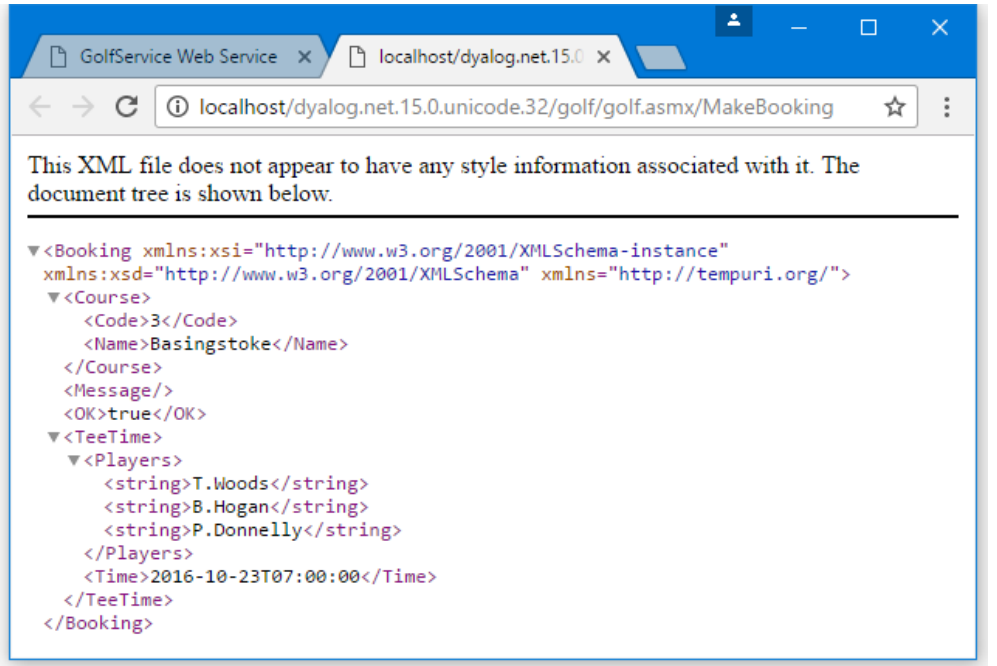
Notice the way a `DateTime` value is specified. Note too that the `GimmeNearest` parameter is `Boolean`, so you must enter `"True"` or `"False"`. If you enter 0 or 1, it will cause an error and the application will refuse to try to call `MakeBookings` because you have specified the wrong type for a parameter.

When you try this yourself, remember to enter a date that is within the next 30 days, and a time between 07:00 and 15:00. Alternatively, you may wish to experiment with invalid data to check the error handling.



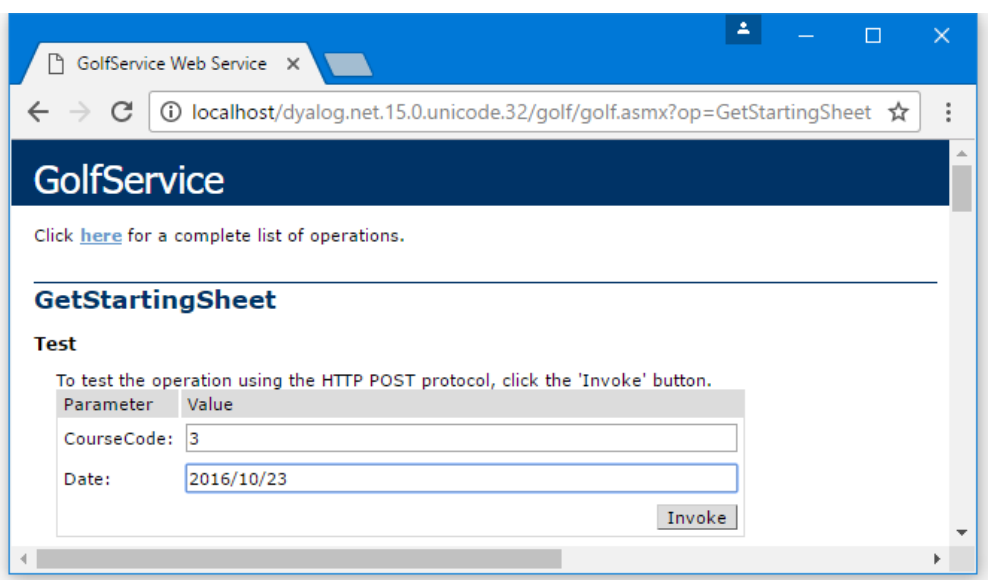
The result of invoking `MakeBooking` with this data is shown below.

Notice how all the information about the `Booking` object structure, including the structure of the sub-objects, is provided.



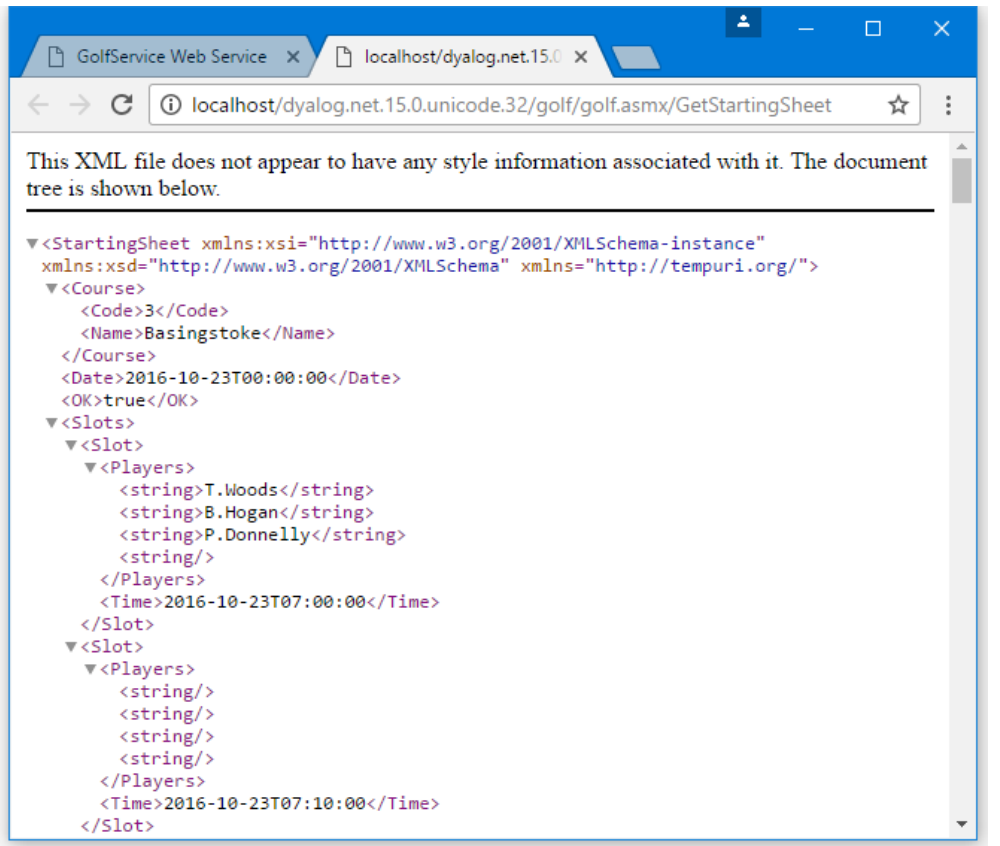
The following picture shows data suitable for invoking the `GetStartingSheet` method.

If you try this for yourself, choose a course and date on which you have made at least one successful booking.



Finally, the result of the `GetStartingSheet` function is illustrated below.

The output clearly shows that the result, a `StartingSheet` object, contains an array of `Slot` objects, each of which contains a `Time` field and a `Players` field.



Using GolfService from C

The `csharp` sub-directory in `samples\asp.net\golf` contains sample files for accessing the `GolfService` Web Service from C#. The C# source code in `Golf.cs` is shown below.

```
using System;

class MainClass  {

    static void Main(String[] args)
        {
        GolfService golf = new GolfService();
        int nArgs = args.Length;
        Booking booking;

        booking=golf.MakeBooking(
/* Course Code     */    1,
/* Desired Tee Time */  DateTime.Parse(args[0]),
/* nearest is OK    */  true,
/* player 1        */    (nArgs > 1) ? args[1] : "",
/* player 2        */    (nArgs > 2) ? args[2] : "",
/* player 3        */    (nArgs > 3) ? args[3] : "",
/* player 4        */    (nArgs > 4) ? args[4] : ""
            );

        Console.WriteLine(booking.OK);
        Console.WriteLine(booking.TeeTime.Time.ToString());
        foreach (String player in booking.TeeTime.Players)
            Console.WriteLine(player);
        }
}
```

The following example shows how you may run the C# program `golf.exe` from a Command Prompt window. Please remember to specify a reasonable date and time rather than the one used in this example.

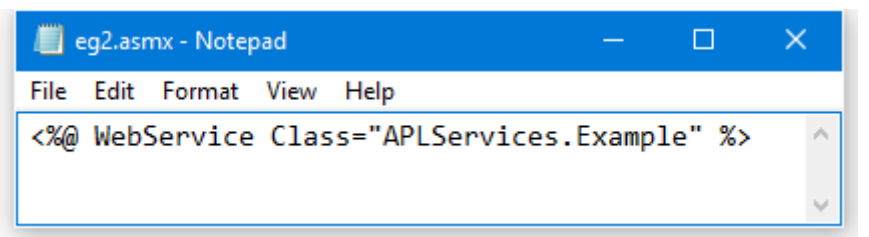
```
csharp>golf 2006-08-07T08:00:00 T.Woods A.Palmer P.Donnelly
True
25/08/2008 08:00:00
T.Woods
A.Palmer
P.Donnelly

csharp>
```

8.11 Sample Web Service: EG2

In all the previous examples, we have relied upon ASP.NET to compile the `APLScript` into a .NET class prior to running it. This sample illustrates how you can make a .NET class yourself.

For this example, the Web Service script, which is supplied in the file `Samples\asp.net\webservices\eg2.asmx` (mapped via an IIS Virtual Directory to the URL `http://localhost/dyalog.net/webservices/eg2.asmx`)is reduced to a single statement that merely invokes the pre-defined class called `APLServices.Example`.

The entire file, viewed in `Notepad`, is shown below.



Given this instruction, ASP.NET will locate the `APLServices.Example` Web Service by searching the `bin` sub-directory for assemblies. Therefore, to make this work, we have only to create a .NET assembly in `samples\asp.net\aplservices\bin`. The assembly should contain a .NET Namespace named `APLServices`, which in turn defines a class named `Example`.

The procedure for creating .NET classes and assemblies in Dyalog APL was discussed in *Section 6.1*. Making a WebService class is done in exactly the same way.

Note that the sub-directory `samples\asp.net\aplservices\bin` already contains copies of the dependent Dyalog DLLs that are required to execute the code.

Start Dyalog as Administrator. This is essential both to allow you to create an assembly.

Starting with a `CLEAR WS`, create a namespace called `APLServices`. This will act as the container corresponding to a .NET Namespace in the assembly.

```
      )NS APLServices
#.APLServices
```

Within `APLServices`, create a class called `Example` that inherits from System.Web.Services.WebService. This is the Web Service class.

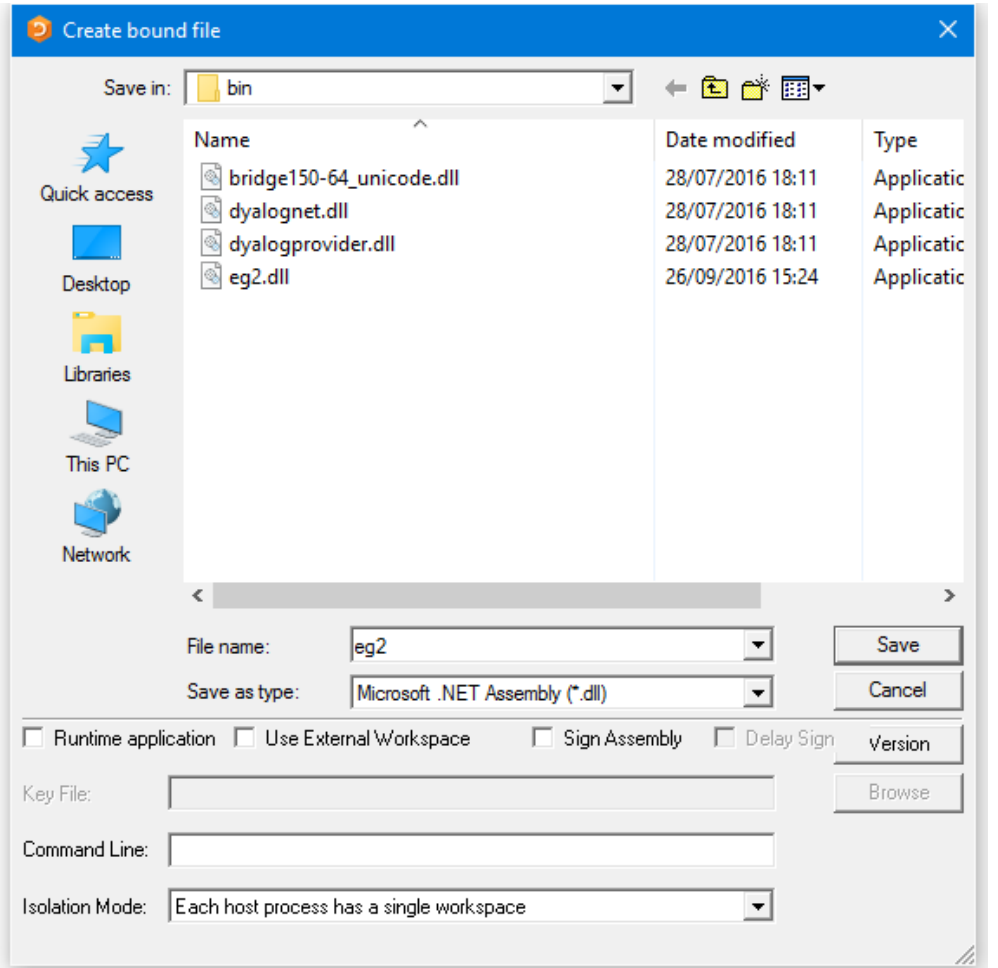
```
      )CS APLServices
#.APLServices
      )ED ⍥Example

:Class Example: WebService
:Using System
:Using System.Web.Services,System.Web.Services.dll
    ∇ R←Add arg
      :Access webmethod
      :Signature Int32←Add Int32 arg1, Int32 arg2
      R←+/arg
    ∇
:EndClass
```

Within `APLServices.Example`, we have a function called `Add` that will represent the single method to be exported by this Web Service.

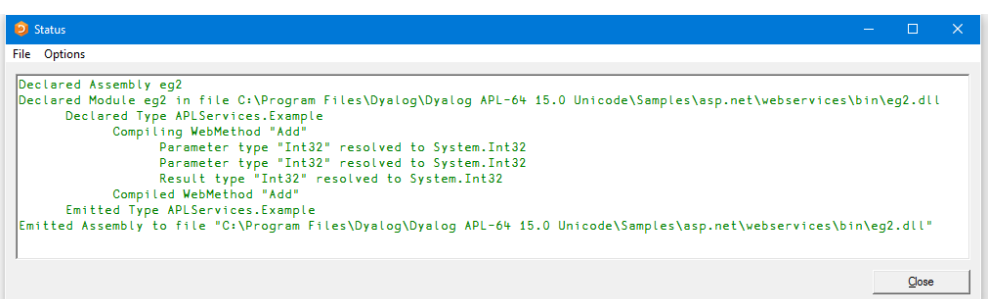
Fix the class, then click *File/Save As* … in the Session menubar and save the workspace in `samples\asp.net\aplwebservices\bin`.

```
C:\Program Files\Dyalog\Dyalog APL 15.0
Unicode\Samples\asp.net\webservices\bin\eg2.dws saved Mon Sep 26 15:31:5
6 2016
```

Select the *Export…* item from the Session *File* menu, and save the assembly as `eg2.dll` in the same directory, that is, `samples\asp.net\webservices\bin`.



When you click *Save*, the Status Window displays the following information to confirm that the assembly has been created correctly.



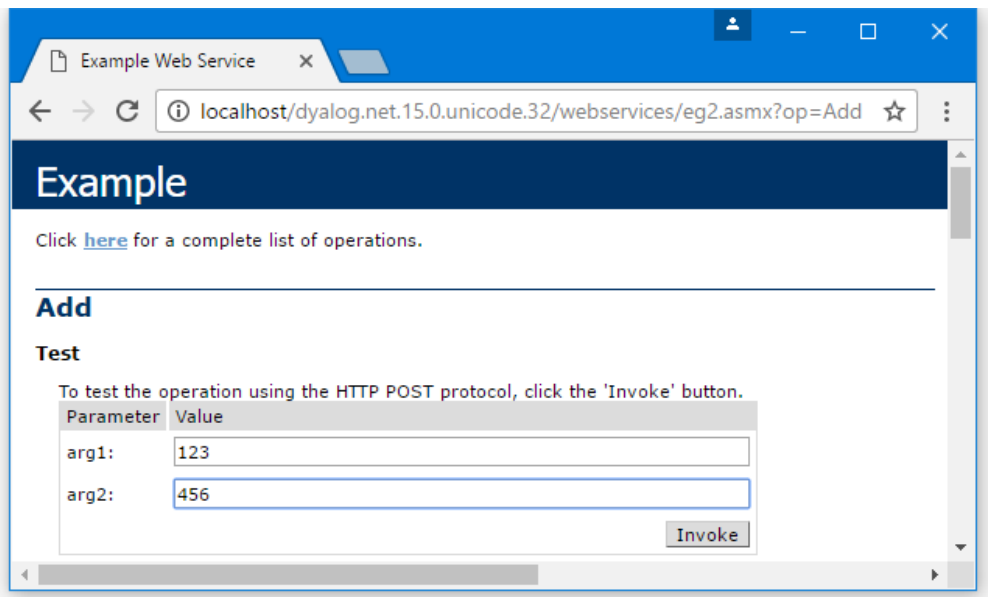
Testing EG2 from a Browser

If you point your browser at the URL:

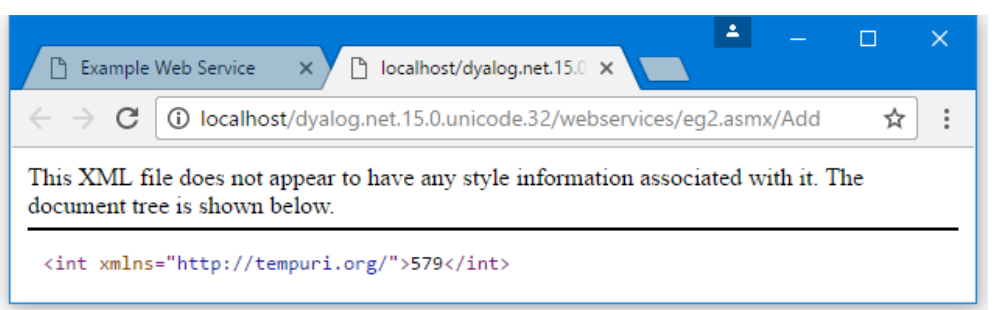
```
http://localhost/dyalog.net.15.0.unicode.32/webservices/eg2.asmx
```

ASP.NET will fabricate a page about it for the browser to display as shown below.

The `Add` method exposed by `APLServices.Example` is shown, together with a Form from which you can invoke it.



If you enter the numbers 123 and 456 in the fields provided, then press *Invoke*, the method will be called and the result displayed as shown below.



9 Calling Web Services

9.1 Introduction

In order to call a Web Service, you need a "proxy class" on the client, which exposes the same methods and properties as the web service. The proxy creates the illusion that the web service is present on the client. Client applications create instances of the proxy class, which in turn communicate with the Web Service via IIS, using TCP/IP and HTTP/XML protocols.

Microsoft provides a utility called `WSDL.EXE` that queries the metadata (Web Service Definition Language) of a Web Service and generate C# source code for a matching proxy class.

9.2 The MakeProxy function

The `MakeProxy` function is provided in the supplied workspace `samples\asp.net\webservices\webservices.dws`.

`MakeProxy` is monadic and its argument specifies the URL of the Web Service to which you want to connect. For example, the following expressions uses `MakeProxy` to connect to the LoanService sample Web Service provided with Dyalog .NET:

```
    MakeProxy'http://localhost/dyalog.net/Loan/Loan.asmx'
```

`MakeProxy` runs the Microsoft utility `WSDL.EXE` passing the name of your URL to it as an argument. The utility then creates a C# source code file in your current directory that contains the code necessary to create a proxy class. The name of the C# file is the name of the Web Service (as declared in its header line) followed by the extension .cs.

`MakeProxy` then calls the C# compiler to compile this file, creating an assembly with the same name, but with a .dll extension, in your current directory. This assembly contains a .NET class of the same name.

`MakeProxy` attempts to determine the correct path for `WSDL.EXE` and `CSC.EXE`, but future versions of Microsoft.NET or Visual Studio require changes, in which case you will have to modify this function to locate these tools.

9.3 Using LoanService from Dyalog APL

For example, the above call to `MakeProxy` will create a C# source code file called `LoanService.cs`, and an assembly called `LoanService.dll` in your current directory. The name of the proxy class in `LoanService.dll` is `LoanService`.

You use this proxy class in exactly the same way that you use any .NET class. For example:

```
      ⎕USING ←,⊂',.\LoanService.dll'
      LN←⎕NEW LoanService
      LN.CalcPayments 100000 20 10 15 2
LoanResult
```

Notice that, as expected, the result of `CalcPayments` is an object of type `LoanResult`. For convenience, we will assign this to `LR` and then reference its fields:

```
      LR←LN.CalcPayments 100000 20 10 15 2
      LR.Periods
10 11 12 13 14 15 16 17 18 19 20
      LR.InterestRates
2 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 7.5 8 8.5 9 9.5 10 10.5 ...
      LR.(((⍴InterestRates),⍴Periods)⍴Payments)
920.1345384 844.5907851 781.6836919 728.4970675 682.947 ...
```

The `Payments` field is, of course, a vector because it was defined that way. However, as can be seen above, it is easy to give it the "right" shape.

When you execute the `CalcPayments` method in the proxy class, the class transforms and packages up your arguments into an appropriate SOAP/XML stream and sends them, using TCP/IP, to the URL that represents the Web Service wherever that URL is on the internet or your Intranet. It then decodes the SOAP/XML that comes back, and returns the response as the result of the method.

Note that, depending upon the speed of your connection, and the logical distance away of the Web Service itself, calling a Web Service method can take a significant amount of time; regardless of how much time it actually takes to execute on its server.

9.4 Using GolfService from Dyalog APL

The workspace `samples\asp.net\webservices\webservices` contains functions that present a GUI interface to the `GolfService` web service.

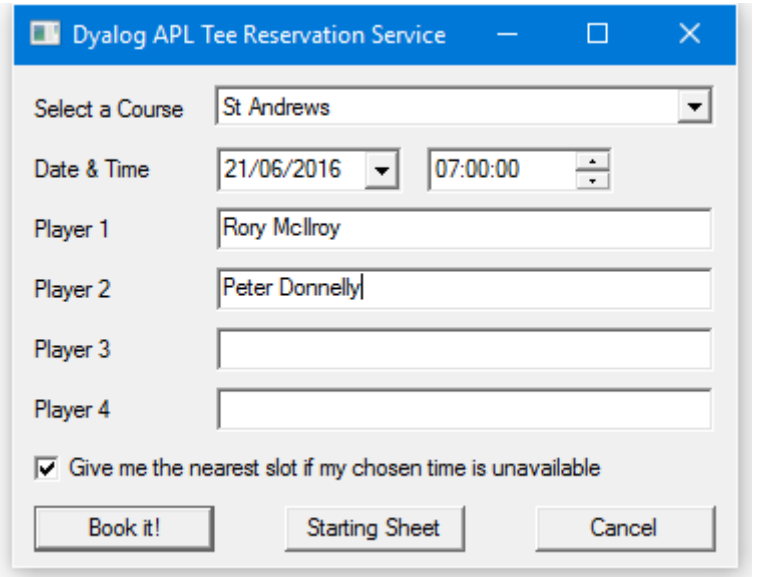
The `GOLF` function accesses `GolfService` through a proxy class. `GOLF` is called with an argument of 0 or 1. Use 1 to force `GOLF` to create or rebuild the proxy class, which it does by calling `MakeProxy`. You must use an argument of 1 the first time you call `GOLF`, or if you ever change the `GolfService` APL code.

Note that you cannot make the proxy for `GolfService` unless the Web Server class has been compiled on the server. At present, the only way to trigger the compilation of golf.asmx into a Web Service is to visit the page once using Internet Explorer as described in the previous chapter.

The first few lines of the function are listed below. If the argument is 1, line [2] makes the proxy class `GolfService.DLL` in the current directory; if not it is assumed to be there already. Line [6] defines `⎕USING` to use it, and Line [7] creates a new instance which is assigned to `GS`. Line [8] calls the `GetCourses` method, which returns a vector of `GolfCourse` objects. Notice how namespace reference array expansion is used to extract the course codes and names from the `Code` and `Name` fields respectively.

```
     ∇ GOLF FORCE;F;DLL;COURSES;COURSECODES;N;GS;⎕USING
[1]    :If FORCE≠0
[2]        DLL←MakeProxy
             'http://localhost/dyalog.net/golf/golf.asmx'
[3]    :Else
[4]        DLL←'.\GolfService.dll'
[5]    :EndIf
[6]    ⎕USING←'System'(',',DLL)
[7]    GS←⎕NEW GolfService
[8]    COURSECODES COURSES←↓⍉↑GS.GetCourses.(Code Name)
```

The following screen shot illustrates the user interface provided by `GOLF`. In this example, the user has typed the names of two golfers (one rather more famous than the other - at least in APL circles) and then presses the *Book it!* button.

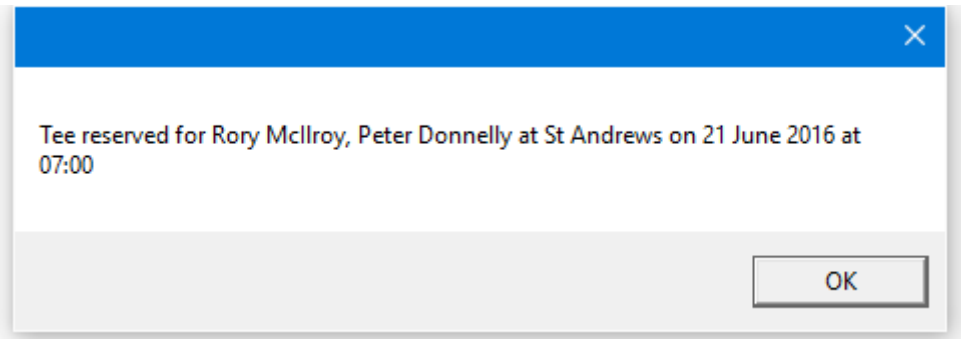


This action fires the `BOOK` callback function which is shown below.

```
     ∇ BOOK;CCODE;YMD;HOUR;MINUTES;FLAG;NAMES;BOOKING;M
[1]    CCODE←⊃F.COURSE.SelItems/COURSECODES
[2]    YMD←3↑F.DATE.(IDNToDate⊃DateTime)
[3]    HOUR MINUTES←2↑1↓F.TIME.DateTime
[4]    FLAG←1=F.Nearest.State
[5]    NAMES←F.(Name1 Name2 Name3 Name4).Text
[6]    BOOKING←GS.MakeBooking CCODE
        (⎕NEW DateTime (YMD,HOUR MINUTES 0)),FLAG,NAMES
[7]    'M'⎕WC'MsgBox'
[8]    :If BOOKING.OK
[9]        M.Text←'Tee reserved for
             ',¯2↓⊃,/BOOKING.TeeTime.Players,¨⊂', '
[10]       M.Text,←' at ',BOOKING.Course.Name
[11]       M.Text,←' on ',BOOKING.TeeTime.Time.
            (ToLongDateString,' at ',ToShortTimeString)
[12]   :Else
[13]       M.Text←BOOKING.(Course.Name,'',
                  TeeTime.Time.(ToLongDateString,
            ' at ',ToShortTimeString),' ',Message)
[14]   :EndIf
[15]   ⎕DQ'M'
     ∇
```

Line [6] calls the `MakeBooking` method of the `GS` object, passing it the data entered by the user. The result, a `Booking` object, is assigned to `BOOKING`. Line [8] checks its `OK` field to tell whether or not the reservation was successful. If so, lines [9-11] display the message box illustrated below.

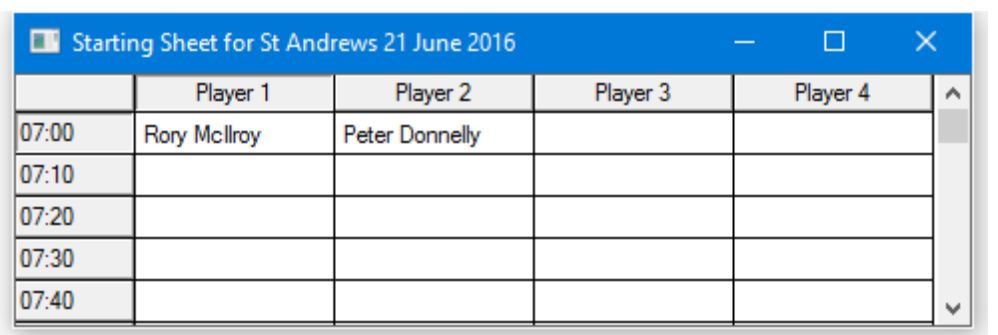
Notice how the various fields are extracted and notice how the `ToLongDateString` and `ToShortTime String` methods are employed.



Pressing the *Starting Sheet* button runs the `SS` callback listed below.

```
    ∇ SS;CCODE;YMD;M;SHEET;OK;COURSE;TEETIME;S;DATA;N
      ;TIMES
[1]   CCODE←⊃F.COURSE.SelItems/COURSECODES
[2]   YMD←3↑F.DATE.(IDNToDate⊃DateTime)
[3]   SHEET←GS.GetStartingSheet CCODE(⎕NEW DateTime YMD)
[4]   :If SHEET.OK
[5]       DATA←↑(SHEET.Slots).Players
[6]       TIMES←(SHEET.Slots).Time
[7]       'S'⎕WC'Form'('Starting Sheet for ',
             SHEET.Course.Name,' ',
             SHEET.Date.ToLongDateString)
             ('Coord' 'Pixel')('Size' 400 480)
[8]       'S.G'⎕WC'Grid'DATA(0 0)(S.Size)
[9]       S.G.RowTitles←TIMES.ToShortTimeString
[10]      S.G.ColTitles←'Player 1' 'Player 2'
                        'Player 3' 'Player 4'
[11]      S.G.TitleWidth←60
[12]      ⎕DQ'S'
[13]  :Else
[14]      'M'⎕WC'MsgBox'('Starting Sheet for ',
             SHEET.Course.Name,' ',
             SHEET.Date.ToLongDateString)
                        ('Style' 'Error')
[15]      M.Text←SHEET.Message
[16]      ⎕DQ'M'
[17]  :EndIf
    ∇
```

Line [3] calls the `GetStartingSheet` method of the `GS` object. The result, a `StartingSheet` object, is assigned to `SHEET`. Line [4] checks its `OK` field to see if the call succeeded. If so, lines [5-12] display the result in a Grid, which is illustrated below.

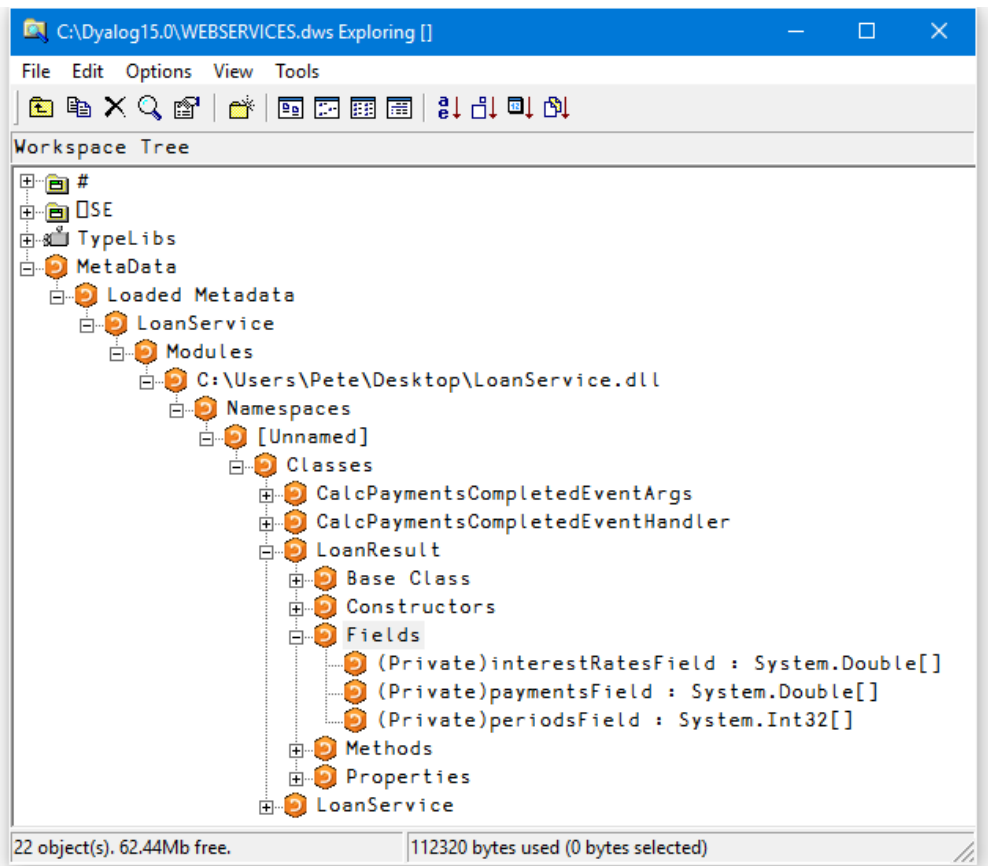


9.5 Exploring Web Services

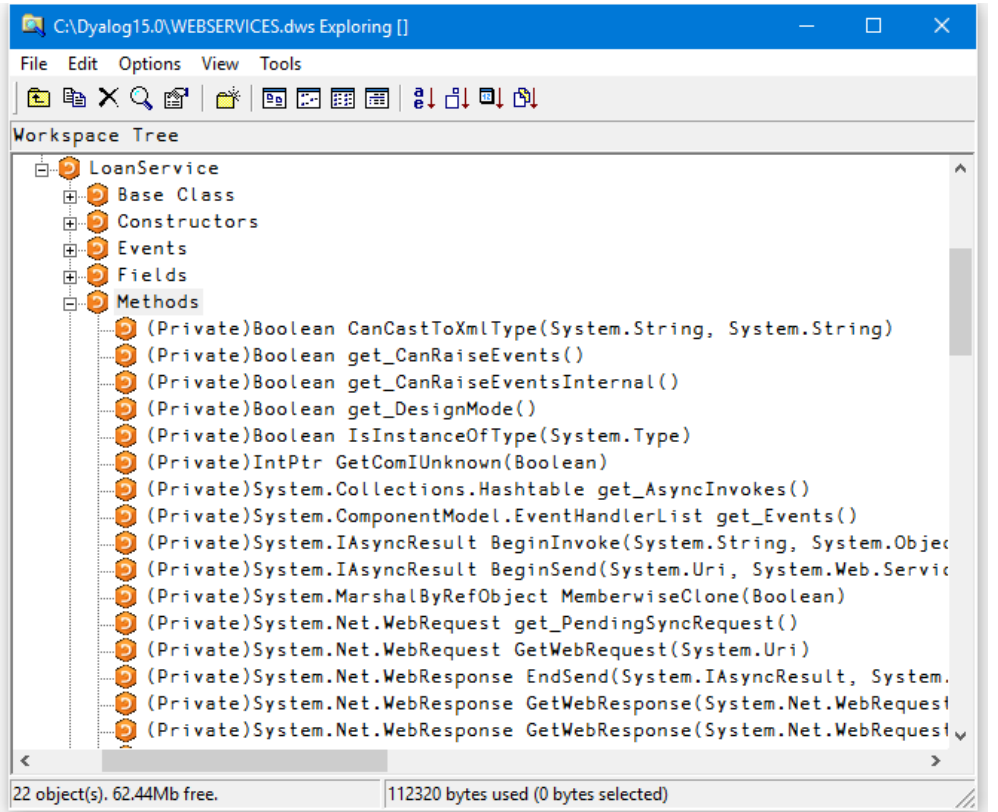
You can use the Workspace Explorer to browse the proxy class associated with a Web Service, in exactly the same way that you can browse any other .NET Assembly. The following screen shots show the *Metadata* for `LoanService`, loaded from the `LoanService.dll` proxy.

Remember, `LoanService` was written in `APLScript`, but it appears and behaves just like any other .NET class.

The first picture displays the structure of the `LoanResult` class.



The second picture shows the methods exposed by `LoanService`. In addition to `CalcPayments`, which was written in `APLScript`, there are a large number of other methods, which have been inherited from the base class.



9.6 Asynchronous Use

Web Services provide both synchronous (client calls the function and waits for a result) and asynchronous operation.

Each method is exposed as a function with the same name (the synchronous version) together with a pair of functions with that name prefixed with `Begin` and `End` respectively.

The `Beginxxx` functions take two additional parameters; a delegate class that represents a callback function and a state parameter.

To initiate the call, you execute the `Beginxxx` method using the standard parameters followed by two objects. The first is an object of type `System.AsyncCallback` that represents an asynchronous callback, that is, a callback to be invoked when the asynchronous call is complete. The second is an object which is used to supply extra information. We will see how callbacks are used later in this section. If you are not

using a callback, these items should be null object references. You can specify a reference to a null object using the expression `(⎕NS'')`. For example, using the `LoanService` sample as above:

```
      A←LN.BeginCalcPayments 10000 16 10 12
                             9(⎕NS'')(⎕NS'')
```

The result is an object of type `WebClientAsynchResult`.

```
      A
System.IAsyncResult ⎕CLASS System.Web.Services.Protocols.WebClientAsyncR
esult
```

Then, some time later, you call the `Endxxx` method with this object as a parameter. For example:

```
      LN.EndCalcPayments A
LoanResult
```

You can execute several asynchronous calls in parallel:

```
      A1←LN.BeginCalcPayments 20000 20 10 15
                              7(⎕NS'')(⎕NS'')
      A2←LN.BeginCalcPayments 30000 10  8 12
                              3(⎕NS'')(⎕NS'')

      LN.EndCalcPayments A1
LoanResult
      LN.EndCalcPayments A2
LoanResult
```

Using a callback

The simple approach described above is not always practical. If it can take a significant amount of time for the web service to respond, you may prefer to have the system notify you, via a callback function, when the result from the method is available.

The example function `TestAsyncLoan` in the workspace `samples\asp.net\webservices\webservices.dws` illustrates how you can do this. It is somewhat artificial, but hopefully explains the mechanism that is involved.

`TestAsyncLoan` itself is just a convenience function that calls `AsyncLoan` with suitable arguments. `TestAsyncLoan` takes an argument of 1 or 0 that determines whether or not a Proxy class for `LoanService` is to be built.

```
     ∇ TestAsyncLoan MAKEPROXY
[1]    (⍕MAKEPROXY),' AsyncLoan 10000 10 8 5 3'
[2]    MAKEPROXY AsyncLoan 10000 10 8 5 3
     ∇
```

The `AsyncLoan` function and its callback function `GetLoanResult` are more interesting.

```
     ∇ {MAKEPROXY}AsyncLoan ARGS;DLL;SINK;LN;AS;AR
[1]    :If 2≠⎕NC'MAKEPROXY' ⋄ MAKEPROXY←0 ⋄ :EndIf
[2]    :If MAKEPROXY
[3]       DLL←MakeProxy'http://localhost/dyalog.net/loan/
                        loan.asmx'
[4]    :Else
[5]       DLL←'.\LoanService.dll'
[6]    :EndIf
[7]    ⎕USING←'System'(',',DLL)
[8]    LN←⎕NEW LoanService
[9]    AS←⎕NEW System.AsyncCallback,⊂⎕OR'GetLoanResult'
[10]   AR←LN.BeginCalcPayments ARGS,AS,LN
[11]   'AsyncLoan waits for async call to complete'
[12]   :While 0=AR.IsCompleted
[13]       ⍞←'.'
[14]   :EndWhile
     ∇
     ∇ GetLoanResult arg;OBJ;LR;RSLT
[1]    'GetLoanResult callback fires ...'
[2]    OBJ←arg.AsyncState
[3]    LR←OBJ.EndCalcPayments arg
[4]    RSLT←LR.(((⍴Periods),(⍴InterestRates))⍴Payments)
[5]    RSLT←((⊂''),LR.Periods),(LR.InterestRates),[1]RSLT
[6]    'Result is'
[7]    ⎕←RSLT
     ∇
```

The effect of running `TestAsyncLoan` is as follows:

```
      TestAsyncLoan 0
0 AsyncLoan 10000 10 8 4 3

...AsyncLoan waits for async call to complete...
...GetLoanResult callback fires ...

...Result is
      3           3.5         4
 8  117.2957193 105.7694035  96.5607447
 9  119.5805173 108.0741442  98.88586746
121.892753  110.409689  101.2451382
```

`AsyncLoan[8]` creates a new instance of the `LoanService` class called `LN`. The next line creates an object of type `System.AsyncCallback` named `AS`. This object, which is termed a *delegate*, identifies the callback function that is to be invoked when the asynchronous call to `CalcPayments` is complete. In this case, the name of the callback function is `GetLoanResult`. Note that `⎕OR` is necessary because the `AsyncCallback` constructor must be called with a parameter of type `System.Object`. The line `AsyncLoan[10]` calls `BeginCalcPayments` with the parameters for `CalcPayments`, followed by references to `AS` (which identifies the callback) and `LN`, which identifies the object in question. The latter will turn up in the argument supplied to the `GetLoanResult` callback. Lines[12-14] loop, displaying dots, until the asynchronous call is complete. `GetLoanResult` will be invoked during or immediately after this loop, and will be executed in a different APL thread.

When the `GetLoanResult` callback is invoked, its argument `arg` is an object of type `System.Web.Services.Protocols.WebClientAsyncResult`. It is in fact a reference to the same object `AR` that was the result returned by `BeginCalcPayments`.

This object has an `AsyncState` property that references the `LoanService` object `LN` that we passed as the final parameter to `BeginCalcPayments`. `GetLoanResult[2]` retrieves this object and assigns it to `OBJ`. `GetLoanResult[3]` calls the `EndCalcPayments` method, passing it `arg` as the `AsyncResult` parameter as before. The resulting `LoanResult` object is then formatted and displayed.

10 Writing ASP.NET Web pages

10.1 Introduction

Under Microsoft IIS, a *static* web page is defined by a simple text file with the extension .htm or .html that contains simple HTML. When a browser requests such a page, IIS simply reads it and sends its content back to the client. The contents of a static web page are constant and, until somebody changes it, the page appears the same to all users at all times.

A *dynamic* web page is represented by a simple text file with the extension .aspx. Such a file may contain a mixture of (static) HTML, ASP.NET objects and a *server-side script*. ASP.NET objects are built-in .NET classes that generate HTML when the page is processed. Scripts contain functions and subroutines that are invoked by events (such as the Page_Load event) or by user interaction.

Typically, a script will generate HTML dynamically, when the page is loaded. For example, a script could perform a database operation and return an HTML table containing a list of products and prices. A script may also contain code to process user interaction, for example to process the contents of a Form that is filled in and then submitted by the user. These scripts are referred to as server-side scripts because they are executed on the server. The browser sees only the results produced by the scripts and not the scripts themselves. Code in a server-side script always involves the generation of a new page by the server for display in the browser.

The first time ASP.NET processes a .NET web page, it compiles the entire page into a .NET Assembly. Subsequently, it calls the code in the assembly directly. The language used to compile the page is defined in the <script> section, which is typically defined at the top of the page. If the <script> section is omitted, or if it fails to explicitly specify the language attribute, the page is compiled using the default scripting language. This is configurable, but is typically VB or C#.

This Chapter is made up almost entirely of examples, the source code of which is supplied in the samples\asp.net directory and the sub-directories it contains. This directory is mapped as an IIS Virtual Directory named `dyalog.net`, so you may execute the examples by specifying the URL `http://localhost/dyalog.net/` followed by the name of the sub-directory and page. You can get an overview of the samples by starting on the page `http://localhost/dyalog.net/index.htm` and follow links from there.

To use `APLScript` effectively in Web Pages, you need to have a thorough understanding of how ASP.NET works.

In the first example, an outline description ASP.NET technology is provided. For further information, see the Microsoft .NET Framework documentation and *Beginning ASP.NET using VB.NET*, Wrox Press Ltd, ISBN 1861005040.

10.2 Your first APL Web Page

The first web page example is `tutorial/intro1.aspx`, which is listed below. This page displays a button whose text is reversed each time you press it.

Note that the example is intended to be run in the framework of the tutorial and contains two lines of code (shown in italic) that refer to this framework and should be ignored.

```
<%@Register TagPrefix="tutorial" Namespace="Tutorial"
Assembly="tutorial" %>
<script language="Dyalog" runat="server">

∇Reverse args
:Access public
:Signature Reverse Object,EventArgs
(⊃args).Text←⌽(⊃args).Text
∇

</script>

<html>
<body>
<Form runat=server>
      <asp:Button id="Pressme"
      Text="Press Me"
      runat="server"
      OnClick="Reverse"
      />
</form>
<tutorial:index runat="server"/>
</body>
</html>
```

In this example, the page language is defined in the `<script>` section to be `"Dyalog"`. This in turn is mapped to the `APLScript` compiler via information in the application's web.config file or the global IIS configuration file, `machine.config`.

The page layout is described in the section between the `<html>` and `</html>` tags. This page contains a Form in which there is a Button labelled (initially) "Press Me"

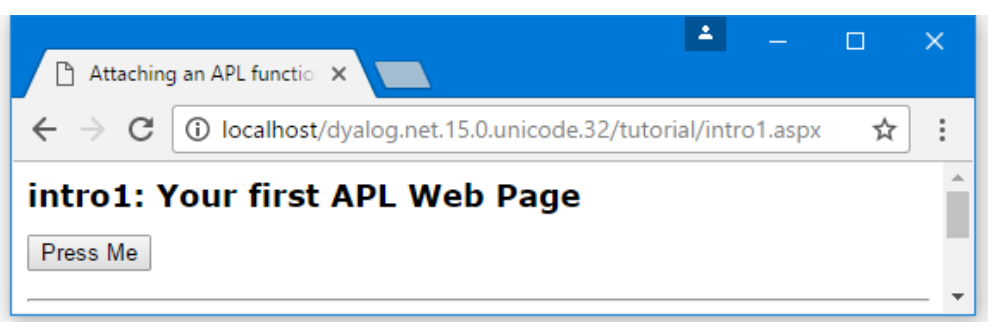
The Form and Button page elements may appear to be simple HTML, but in fact there is more to them than meets the eye and they are actually both types of ASP.NET *intrinsic controls*.

Firstly, the `runat="server"` attribute indicates that an HTML element should be parsed and treated as an HTML server control. Instead of being handled as pure text that is to be transmitted to the browser "as is", an HTML server control is effectively compiled into statements that then generate HTML when executed. Furthermore, an HTML server control can be accessed programmatically by code in the Script, whereas a pure HTML element cannot. On its own, `runat="server"` identifies the HTML element as a so-called *basic* intrinsic control.

When you add `runat="server"` to a Form, ASP.NET automatically adds other attributes that cause the values of its controls to be POSTed back to the same page. In addition, ASP.NET adds a HIDDEN control to the form and stores state information in it. This means that when the page is reloaded into the browser the state and contents of some or all of its controls can be maintained, without the need for you to write additional code.

The `asp:` prefix for the Button, identifies the control as a *special* ASP.NET intrinsic control. These are fully-fledged .NET Classes in the .NET Namespace `System.Web.UI.WebControls` that expose properties corresponding to the standard attributes that are available for the equivalent HTML element. You manipulate the control as an object, while it, at runtime, emits HTML that is inserted into the page.

At this point, it is instructive to study what happens when the page is first loaded and the appearance of the page is illustrated below.



The HTML that is transmitted to the browser is:

```
<html>
<body>
    <form name="ctrl1" method="post" action="intro1.aspx" id="ctrl1">
<input type="hidden" name="__VIEWSTATE"
value="YTB6NTQ3ODg0MjcyX19feA==5725bd57" />

        <input type="submit" name="Pressme" value="Press Me"
id="Pressme" />
    </form>
</body>
</html>
```

Firstly, notice that, as expected, the contents of the `<script>` section are not present. Secondly, because the Form and Button are intrinsic controls, ASP.NET has added certain attributes to the HTML that were not specified in the source code.

The Button now has the added attribute `input type="submit"`, which means that pressing the Button causes the contents of the Form to be transmitted back to the sever.

The Form now has `method="post"` and `action="intro1.aspx"` attributes, which means that, when the Form is submitted, the data is POSTed back to `intro1.aspx`, the page that generated the HTML in the first place.

So when the user presses the button, the browser sends back a POST statement, with the contents of the Form, including the value of the HIDDEN field, requesting the browser to load `intro1.aspx`.

In the server, ASP.NET reloads the page and processes it again. In fact, because of the stateless nature of HTTP, the server does not know that it is reprocessing the same page, except that it is being executed by a POST command with the hidden data embedded in the Form that it put there the first time around. This is the mechanism by which ASP.NET *remembers* the state of a page from one invocation to another.

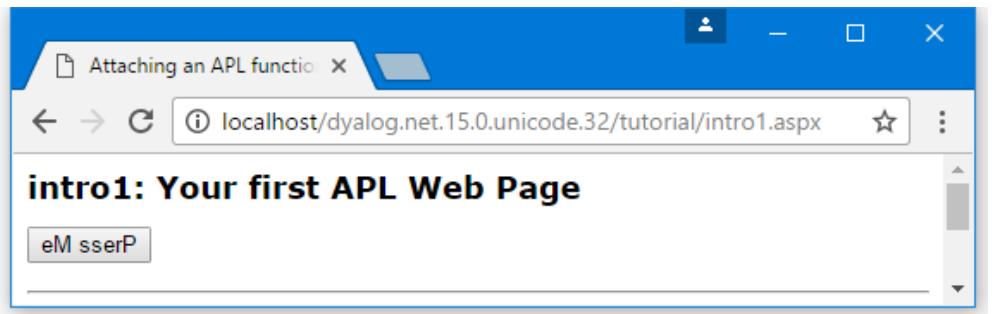
This time, because a POST back is loading the page, and because the `Pressme` button caused the POST, ASP.NET executes the function associated with its `onClick` attribute, namely the `APLScript` function `Reverse`.

When it is called, the argument supplied to Reverse contains two items. The first of these is an object that represents the control that generated the `onClick` event; the second is an object that represents the event itself. In fact, `Reverse` and its argument are very similar to a standard Dyalog callback function.

```
∇Reverse args
:Access public
:Signature Reverse Object,EventArgs
(⊃args).Text←⌽(⊃args).Text
∇
```

The code in the `Reverse` function is simple. The expression (`⊃args`) is a namespace reference (ref) to the Button, and (`⊃args`).Text refers to its Text property whose value is reversed. Note that `Reverse` could just as easily refer to the Button by name, and use `Pressme.Text` instead.

After pressing the button, the page is redisplayed as shown below:



This time, the HTML generated by `intro1.aspx` is:

```
<html>
<body>
    <form name="ctrl1" method="post" action="intro1.aspx" id="ctrl1">
<input type="hidden" name="__VIEWSTATE"
value="YTB6NTQ3ODg0MjcyX2Ewel9oejV6MXhfYTB6X2h6NXoxeF9hMHph
MHpoelRlXHh0X2VNIHNzZXJQeF9feF9feHhfeHhfeF9feA==45acf576"
/>

        <input type="submit" name="Pressme" value="eM sserP"
id="Pressme" />
    </form>
</body>
</html>
```

Returning to the `Reverse` function, note that the declaration statements at the top of the function are essential to make it callable in this context.

```
∇Reverse args
:Access public
:Signature Reverse Object,EventArgs
(⊃args).Text←⌽(⊃args).Text
∇
```

Firstly the `Reverse` function must be declared as a public member of the script. This is achieved with the statement.

```
:Access Public
```

Secondly, the .NET runtime will only call the function if it possesses the correct signature, which is derived from its parameters and their types.

The required signature for a method connected to an event, such as the OnClick event of a Button, is that it takes two parameters; the first of which is of type `System.Object` and the second is of type `System.EventArgs`. The `Reverse` function declares its parameters with the statements:

```
:Signature Reverse Object,EventArgs
```

Note that the parameter declarations do not include the `System` prefix. This is because when the script is compiled the names are resolved using the current value of `⎕USING`. When the `APLScript` is *compiled*, the default value for `⎕USING` is automatically defined to contain `System` along with most of the other namespaces that will be used when writing web pages

(Strictly speaking, the first argument is expected to be of type `System.Web.UI.WebControls.Button`, but as this type inherits ultimately from `System.Object` the function signature is satisfied.)

Note that if the `Reverse` function is defined with a signature that does not match that expected signature for the OnClick callback, the function will not be run.

Furthermore, if the function associated with the OnClick statement is not defined as a public method in the `APLScript` the page will appear to compile but the `Reverse` function will not get executed.

Note that unlike Web Services, there is no requirement for a `:Class` or `:EndClass` statement in the script. This is because a file with an `.aspx` extension implicitly generates a class that inherits from `System.Web.UI.Page`.

10.3 The Page_Load Event

`Intro3.aspx` illustrates how you can dynamically initialise the contents of a Web Page using the Page_Load event. This example also introduces another type of Web Control, the `DropDownList` object.

```
<%@Register TagPrefix="tutorial" Namespace="Tutorial"
Assembly="tutorial" %>
<script language="Dyalog" runat="server">

∇Page_Load
:Access Public
list.Items.Add ⊂'Apples'
list.Items.Add ⊂'Oranges'
list.Items.Add ⊂'Bananas'
∇

∇Select (obj ev)
:Access Public
:Signature Select Object obj, EventArgs ev
out.Text←'You selected ',list.SelectedItem.Text
∇
</script>

<html>
<head>
<title>Initialising the contents of the Page using the Page_Load
method</title>
<link rel="stylesheet" type="text/css" href="apl.css">
</head>

<body>
<h1>intro3: The Page_Load method</h1>
<form runat="server">
<asp:DropDownList id="list" runat="server"/>
<p>
<asp:Label id="out" runat="server" />
</p>
<asp:Button id="btn"
    Text="Submit"
    runat="server"
    OnClick="Select"/>
</form>
<tutorial:index runat="server"/>
</body>
</html>
```

When an ASP.NET web page is loaded, it generates a `Page_Load` event. You can use this event to perform initialisation simply by defining a public function called `Page_Load` in

your `APLScript`. This function will automatically be called every time the page is loaded. The `Page_Load` function should be niladic.

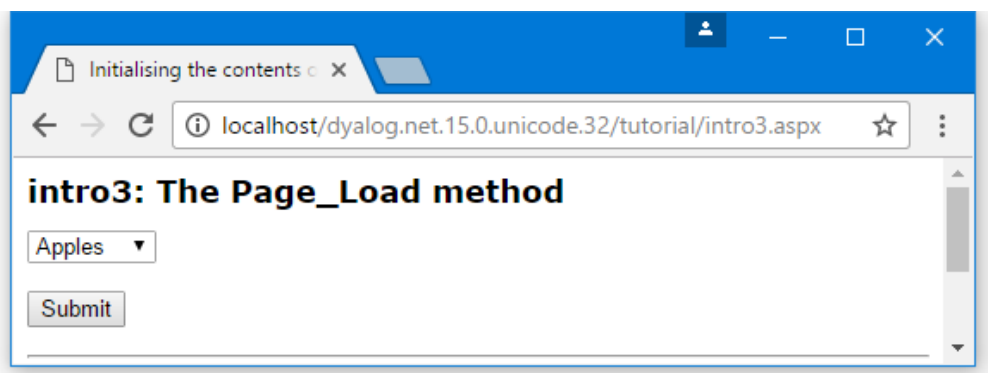
Note that, if the page employs the technique illustrated in `Intro1.aspx`, whereby the page is continually POSTed back to itself by user interaction, your `Page_Load` function will be run every time the page is loaded and you may not wish to repeat the initialisation every time. Fortunately, you can distinguish between the initial load, and a subsequent load caused by the post back, using the `IsPostBack` property. This property is inherited from the `System.Web.UI.Page` class, which is the base class for any `.aspx` page.

The `Page_Load` function in this example checks the value of `IsPostBack`. If 0 (the page is being loaded for the first time) it initialises the contents of the `list` object, adding 3 items "Apples", "Oranges" and "Bananas". The explanation for the statement:

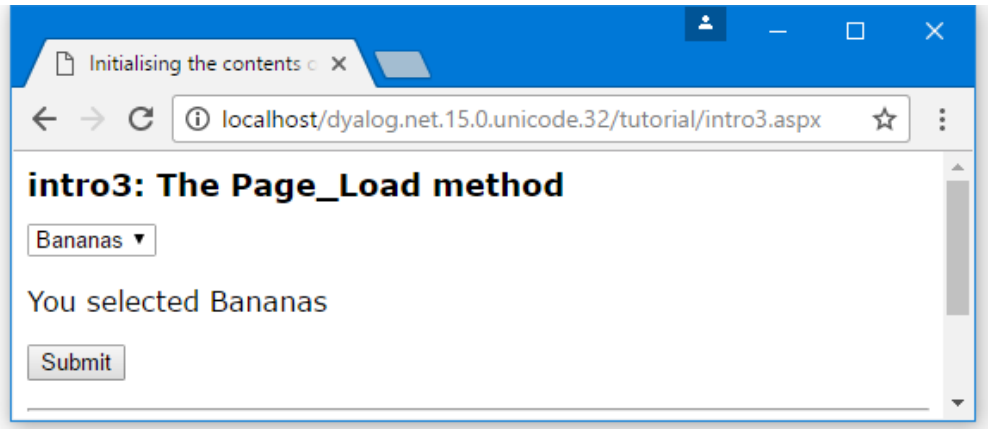
```
      list.Items.Add ⊂'...'
```

is that the `DropDownList` WebControl has an `Items` property that is a collection of `ListItem` objects. The collection implements an `Add` function that takes a `String` Argument that can be used to add an item to the list.

Notice that the name of the object `list` is defined by the `id="list"` attribute of the `DropDownList` control that is defined in the page layout section of the page.



In this example, the page is processed by a POST back caused by pressing the `Submit` button. As it stands, changing the selection in the `list` object does not cause the text in the `out` object to be changed; you have to press the `Submit` button first.



However, you can make this happen automatically by adding the following attributes to the `list` object (see `intro4.aspx`):

```
AutoPostback="true"
OnSelectedIndexChanged="Select"/>
```

`AutoPostback` causes the object to generate HTML that will provoke a post back whenever the selection is changed. When it does so, the `OnSelectedIndexChanged` event will be generated in the server-side script which in turn will call `Select`, which in turn will cause the text in the out object to change.

Note that this technique, which can be used with most of the ASP.NET controls including CheckBox, RadioButton and TextBox controls, relies on a round trip to the server every time the value of the control changes. It will not perform well except on a fast connection to a lightly loaded server.

10.4 Code Behind

It is often desirable to separate the code content of a page completely from the HTML and other text, layout or graphical information by placing it in a separate file. In ASP.NET parlance, this technique is known as *code behind*.

The `intro5.aspx` example illustrates this technique.

```
<%@Page Language="Dyalog"
    Inherits="FruitSelection"
    src="fruit.apl" %>
<%@Register TagPrefix="tutorial" Namespace="Tutorial"
Assembly="tutorial" %>
<html>
<head>
<title>Code behind: separating your code from the page layout</title>
<link rel="stylesheet" type="text/css" href="apl.css">
</head>
<body>
<h1>intro5: Code Behind</h1>
<p>This example illustrates how you can separate your code from the page
layout.</p>
<form  runat="server" >
<asp:DropDownList id="list"
    runat="server"
    autopostback="true"
    OnSelectedIndexChanged="Select"/>
<p>
<asp:Label id="out" runat="server" /></p>
</form>
</body>
<tutorial:index runat="server"/>
</html>
```

The statement

```
%@Page Language="Dyalog" Inherits="FruitSelection" src="fruit.apl" %>
```

says that this page, when compiled, should inherit from a class called `FruitSelection`. Furthermore, the `FruitSelection` class is written in the "Dyalog" language, and its source code resides in a file called `fruit.apl`. `FruitSelection` is effectively the *base class* for the `.aspx` page.

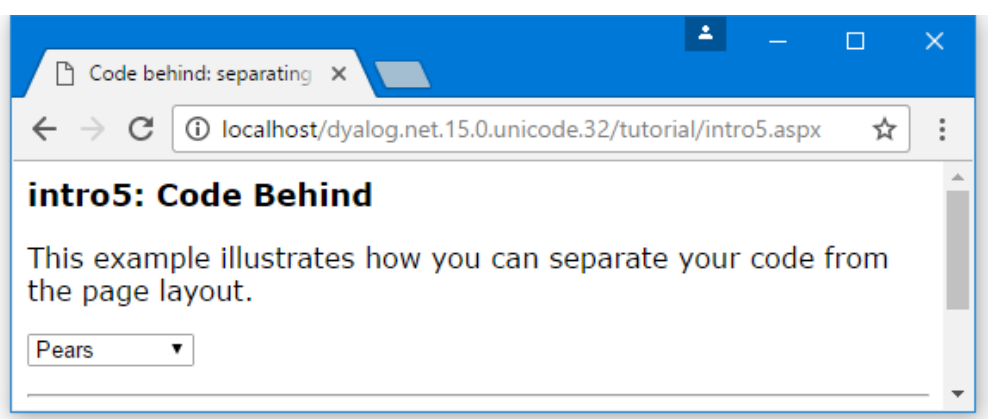
In this case, `fruit.apl` is simply another text file containing the `APLScript` code and is shown below.

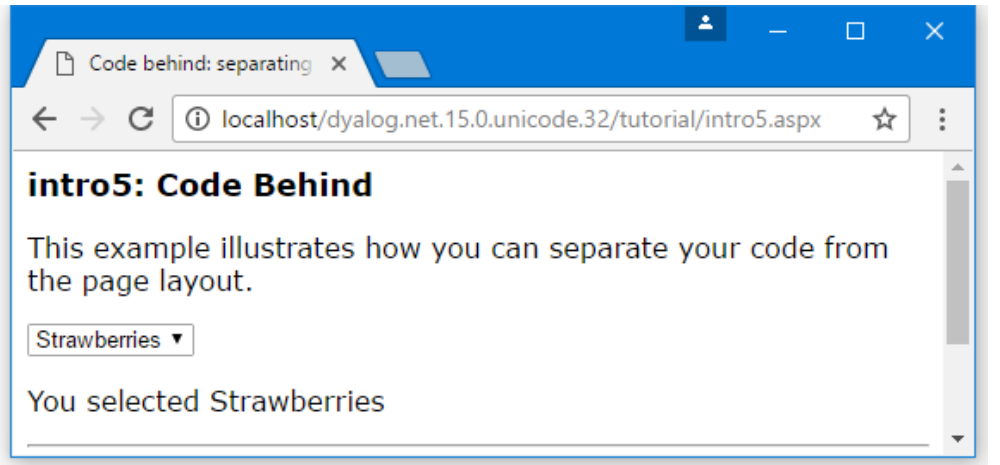
```
:Class FruitSelection: System.Web.UI.Page
:Using System

∇Page_Load
:Access Public
:if 0=IsPostBack
    list.Items.Add ⊂'Pears'
    list.Items.Add ⊂'Nectarines'
    list.Items.Add ⊂'Strawberries'
:endif
∇

∇Select args
:Access public
:Signature Select Object,EventArgs
out.Text←'You selected ',list.SelectedItem.Text
∇
:EndClass
```

The first thing to notice is that the file requires `:Class` and `:EndClass` statements. These are required to tell the `APLScript` compiler the name of the class being defined, and the name of its base class. When the source code is in a `.aspx` file, this information is provided automatically by the `APLScript` compiler.

The name of the class, in this case `FruitSelection`, must be the same name as is referenced in the `.aspx` web page file itself (`intro5.aspx`). The base class must be `System.Web.UI.Page`

The body of the script is just the same as the script section from the previous example. Only the names of the fruit have been changed so that it is clear which example is being executed.





10.5 Workspace Behind

The previous section discussed how APL logic can be separated from page layout, by placing it in a separate APLScript file which is referred to from the `.aspx` web page. It is also possible to have the code reside in a separate *workspace*. This allows you to develop web pages using a traditional workspace approach, and it is probably the quickest way to give an HTML front-end to an existing Dyalog APL application.

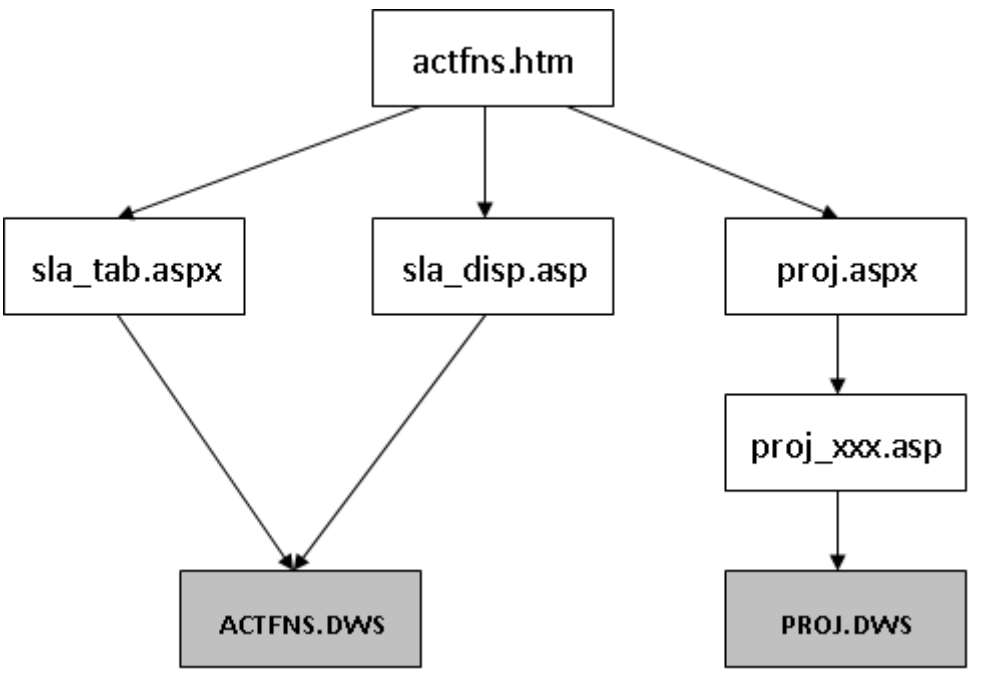
In the previous example, you saw that the `fruit.apl` file defined a new class called `FruitSelection` that inherits from `System.Web.UI.Page`. This class contains a `Page_Load` function that (by virtue of its name) overrides the `Page_Load` method of the underlying base class and will be called every time the web page is loaded or posted back. The `Page_Load` function takes whatever action is required; for example, initialisation. The class also contained a callback function to perform some action when the user pressed a button.

A similar technique is employed when the code behind the web page is implemented in a separate workspace. The workspace should contain a class that inherits from `System.Web.UI.Page`. This class may contain a `Page_Load` function that will be invoked every time the corresponding web page is loaded, and as many callback functions as are required to provide the application logic. The workspace is hooked up to one or more web pages by the `Inherits="<classname>"` and `src="<workspace>"` declarations in the Page directive statement that appears at the beginning of the web page script.

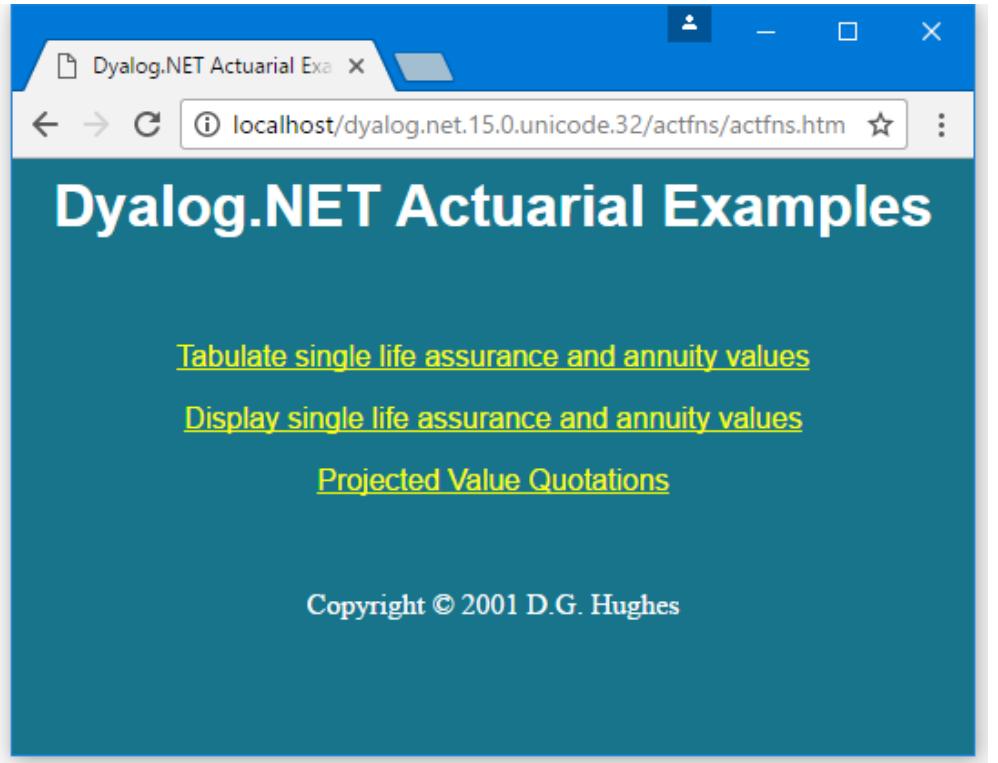
The `ACTFNS` subdirectory in `samples\asp.net` contains some examples of Dyalog APL systems that have been converted to run as Web applications using this technique.

Dyalog is grateful to David Hughes who provided the original workspaces and advised on their conversion.

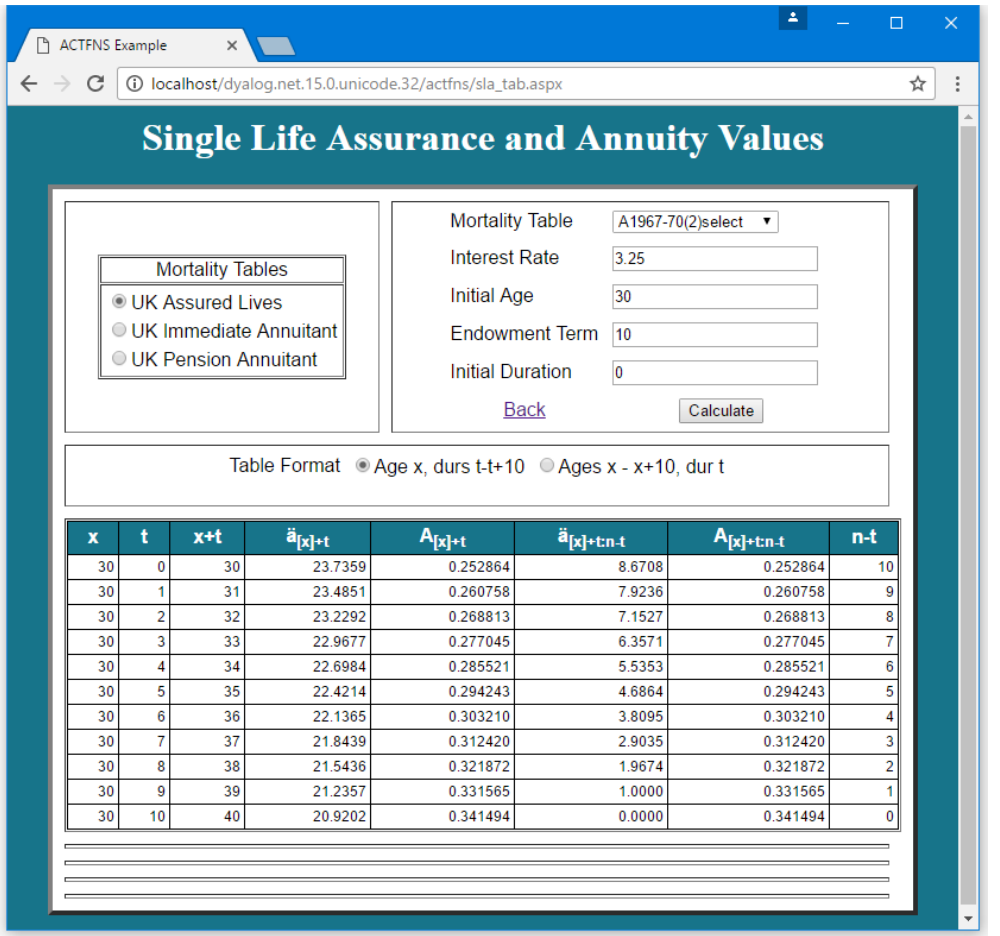
The two workspaces are named `ACTFNS.DWS` and `PROJ.DWS`. The original code used the Dyalog APL GUI to display an input Form, collect and validate the user's input, and calculate and display the results. The original logic supported field level validation and results were immediately recalculated whenever any field was changed. With some exceptions, this has been changed so that the user must press a button to tell the system to recalculate the results. This approach is more appropriate in an Internet application, especially when connection speed is low. Apart from this change, the applications run more-or-less as originally designed.



The diagram above illustrates the structure of the web application and the various files involved. The starting page, `actfns.htm`, simply provides a menu of choices which link to various `.aspx` web pages. These pages in turn are linked to one of the two workspaces via the `src=""` declaration



The `actfns.htm` start page offers 3 application choices



x	t	x+t	ä[x]+t	A[x]+t	ä[x]+t:n-t	A[x]+t:n-t	n-t
30	0	30	23.7359	0.252864	8.6708	0.252864	10
30	1	31	23.4851	0.260758	7.9236	0.260758	9
30	2	32	23.2292	0.268813	7.1527	0.268813	8
30	3	33	22.9677	0.277045	6.3571	0.277045	7
30	4	34	22.6984	0.285521	5.5353	0.285521	6
30	5	35	22.4214	0.294243	4.6864	0.294243	5
30	6	36	22.1365	0.303210	3.8095	0.303210	4
30	7	37	21.8439	0.312420	2.9035	0.312420	3
30	8	38	21.5436	0.321872	1.9674	0.321872	2
30	9	39	21.2357	0.331565	1.0000	0.331565	1
30	10	40	20.9202	0.341494	0.0000	0.341494	0

The result of choosing *Tabulate single life insurance and annuity values*

When you choose the first option, the system loads `sla_tab.aspx`. This defines the screen layout in terms of ASP.NET controls, including the `DataGrid` control for tabulating the results. The `sla_tab.aspx` script contains the declarations `Inherits="actuarial" src="actfns.dws"`, so ASP.NET loads the `actuarial` class from this workspace (via a call to Dyalog APL). When the page is loaded, it generates a `Page_Load` event, which in turn calls its `Page_Load` method. This populates the ASP controls with data, and the resulting web page is displayed. The mechanism is described below.

For further details, see the `sla_tab.aspx` script and `ACTFNS.DWS` workspace.

Converting an Existing Workspace

The steps involved in converting the workspaces were as follows:

1. Replace the Dyalog APL GUI with the equivalent HTML Forms, which are defined in one or more separate `.aspx` web pages. To retain consistency, it is helpful to give the ASP controls the same names as the original GUI controls, which they are replacing.
2. Attach the names of APL callback functions to the appropriate ASP controls; essentially, any controls that will be involved in a postback operation, such as the Submit button.

3. Starting with a `CLEAR WS`, create a `Class` that represents a .NET class based upon `System.Web.UI.Page`. For example, in converting the ACTFNS workspace, we started by creating the class:

```
      )ED ⍥actuarial
```

4. then defining `⎕USING` as follows:

```
:Using System
:Using System.Web.UI,system.web.dll
:Using System.Web.UI.WebControls
:Using System.Web.UI.HtmlControls
:Using System.Data,system.data.dll
```

   The name you choose for this class will replace `classname` in the `Inherits="classname"` declaration in the `.aspx` web page(s) that call it.

5. Create a namespace, change into it, and copy the workspace to be converted; in this case, the starting point was a workspace named `DH_ACTFNS`:

```
      )NS actuarial_utils
      )CS actuarial_utils
#.actuarial_utils
      )COPY DH_ACTFNS
DH_ACTFNS saved ...
```

6. Modify the code as appropriate, inserting a `Page_Load` function and whatever callback functions that are required.

7. Make sure the class 'actuarial' has an *:Include actuarial_utils* statement

The Page_Load function

The `Page_Load` function must be declared as :*Access Public*. `Page_Load` must be spelled correctly as it is this name that causes the function to supersede the base class `Page_Load` method of the same name.

For example, the Page_Load function of the `actuarial` class in ACTFNS.DWS is shown below:

```
∇ Page_Load;INT;AGE;DUR;TERM;TAB_DURS;MPC1;INT1;INT2;
            INTY;RUN_OPTION;OPT
  :Access public
  :Signature Page_Load
 ⍝ Overrides Page_Load method of Page class
 ⍝ Called when Page is loaded or re-loaded after postback
 ⍝ Initialise fields and calculate on initial load only
  :If 0=IsPostBack
      RUN_OPTION←GET_RUN_OPTION
      :Select RUN_OPTION
      :Case 1
          EINT.Text←⍕INT←3.25
          EAGE.Text←⍕AGE←30
          EDUR.Text←⍕DUR←0
          ETRM.Text←⍕TERM←10
          TA.Checked←TAB_DURS←1
          CHANGE_TABLES ⍬
      :Case 2
          CPLAN.Items.Clear
          :For OPT :In ↓⍉OPTSPLAN
          CPLAN.Items.Add{82∊⎕DR 1⍴⍵:⊂⍵ ⋄ ⍵}DETRAIL OPT
          :EndFor
          EMPC1.Text←⍕MPC1←100
          EINT1.Text←⍕INT1←3.25
          EINT2.Text←⍕INT2←3.25
          EINTY.Text←⍕INTY←99
          EAGE.Text←⍕AGE←30
          EDUR.Text←⍕DUR←0
          ETRM.Text←'N/A'
          CHANGE_TABLES ⍬
      :EndSelect
  :EndIf
∇
```

If exported correctly, `Page_Load` will be called every time the calling web page is loaded. This occurs when the page is loaded for the first time, and whenever the page is submitted back to the web server by the browser (postback). A postback will occur whenever a callback function is involved, and potentially at other times.

The `Page_Load` function may determine whether it is being invoked by a first time load, or by a postback, from the value of the `IsPostBack` property. This is a property that it inherits from its base class `System.Web.UI.Page`.

The `Page_Load` example shown above uses this property to control the initialisation of the controls in the calling web page. The names `EINT`, `EAGE`, `EDUR` and so forth refer to names of controls in the calling web page. When `Page_Load` is executed, the `actuarial` object is associated with the web page itself, and so the names of all its controls are visible as sub-objects within it.

Note that the `actuarial` class is used by two different web pages, and the function `GET_RUN_OPTION` function determines which of these are involved. (It does so by detecting the presence or otherwise of a particular control on the page).

Callback functions

The actuarial class in ACTFNS.DWS provides four callback functions named `CALC_FSLTAB_RESULTS`, `CALC_FSL_RESULTS`, `CHANGE_TABLES` and `CHANGE_TABLE_FORMAT`. The first two of these functions are attached as callbacks to the *Calculate* button in each of two separate web pages `sla_tab.aspx` and `sla_disp.aspx`. For example, the statement that defines the button in `sla_tab.aspx` is:

```
<asp:Button id=Button1 runat="server" Text="Calculate"
onClick="CALC_FSLTAB_RESULTS"></asp:Button>
```

The third callback, `CHANGE_TABLES`, is called by `sla_tab.aspx` when the user selects a different set of Mortality Tables from the three provided. `CHANGE_TABLE_FORMAT` is called when the user clicks either of the two radio buttons that select how the output is to be displayed.

Like the `Page_Load` function, callback functions must be declared as being *Public* methods. This is done using the :*Access* statement.

In addition, and this is **essential**, APL callback functions must be declared to have the correct signature expected of .NET callback functions. This means that they must be monadic, and their argument must be declared to be a 2-element nested array

containing two .NET objects; the object that generated the event, and an object that represents the arguments to the event.

Specifically, these parameters must be of type `System.Object` and `System.EventArgs` respectively. However, as our `⎕USING` contains System, it is not necessary to include the `System` prefix.

For example, the statements for the function `CALC_FSLTAB_RESULTS` is shown below:

```
:Access Public
:Signature CALC_FSLTAB_RESULTS Object obj, EventArgs ev
```

Validation functions

In a Dyalog APL web page application, there are basically two approaches to validation. You can handle it entirely yourself or you can exploit the various validation controls that come with ASP.NET. The sample application uses the latter approach by way of illustration. For example:

```
<asp:TextBox id=EINT runat="server"></asp:TextBox>
<asp:RequiredFieldValidator id="RFVINT"
     ControlToValidate="EINT"
     ErrorMessage="Interest Rate must be a number
                   between 0 and 20"
     Text="*"
     runat="server"/></td>
```

These ASP.NET statements associate a `RequiredFieldValidator` named *RFVINT* with the *EINT* field, the field used to enter *Interest Rate*. If the user leaves this field blank, the system will automatically generate the specified error message. The page defines a separate `ValidationSummary` control as follows:

```
<asp:ValidationSummary id="Summary1"
    HeaderText="Please enter a value in the following fields"
    Font-Size="smaller"
    ShowSummary="false"
    ShowMessageBox="true"
    EnableClientScript="true"
    runat="server"/>
```

The `ValidationSummary` control collects error messages from all the other validation controls on the page, and displays them together. In this case, a pop-up message box is used. One advantage of this approach is that this type of validation can be carried out

client-side by local JavaScript that is generated automatically on the server and incorporated in the HTML that is sent to the browser.

Logical field validation for this page is carried out on the server by APL functions that are attached to `CustomValidator` controls. For example:

```
<asp:CustomValidator id="CustomValidator_INT"
     OnServerValidate="VALIDATE_INT"
     ControlToValidate="EINT"
     Display="Dynamic"
     ErrorMessage="Interest Rate must be a number between 0 and 20"
     runat="server"/>
```

These ASP.NET statements associate a `CustomValidator` control named *CustomValidator_INT* with the *Interest Rate* field *EINT*. The statement `OnServerValidate="VALIDATE_INT"` specifies that `VALIDATE_INT` is the validation function for the *CustomValidator_INT* object.

The `VALIDATE_INT` function and its .NET Properties page are shown below.

```
     ∇ VALIDATE_INT MSG;source;args
[1]   ⍝ Validates Interest Rate
[2]    :Access Public
[3]    :Signature VALIDATE_INT Object source,
                  ServerValidateEventArgs args
[4]    source args←MSG
[5]    :Trap 0
[6]        INT←Convert.ToDouble args.Value
[7]    :Else
[8]        args.IsValid←0
[9]        :Return
[10]   :EndTrap
[11]   args.IsValid←(0≤INT)^20≥INT
     ∇
```

To make the `VALIDATE_INT` function available to the calling web page, it is exported as a method. Its *calling signature*, namely that it takes two parameters of type `System.Object` and `System.Web.UI.WebControls.ServerValidateEventArgs` respectively, identifies it as a validation function. All these factors are essential in making it recognizable and callable.

`VALIDATE_INT[4]` assigns its (2-element) argument to `source` and `args` respectively. Both are namespace references to .NET objects. `source` is the object that fired the event (*CustomValidator_INT*). `args` is an object that represents the event. Its `Value`

property returns the text in the control being validated, in this case the control named *EINT1*.

`VALIDATE_INT[6]` converts the text in the EINT control to a number, using the `ToDouble` method of the `System.Convert` class. You could of course use `⎕VFI`, but the `Convert` methods automatically cater for National Language numerical formats. This statement is executed within a `:Trap` control structure because the method will generate a .NET exception if the data in the field is not a valid number.

`VALIDATE_INT[8 11]` set the `IsValid` property of the `ServerValidateEventArgs` object `args` to 0 or 1 accordingly. This also sets the `IsValid` property of the validation control represented by `source`. The system will automatically display the error message associated with any validation control whose `IsValid` property is 0. Furthermore, the page itself has an `IsValid` property, which is the logical-and of all the `IsValid` properties of all the validation controls on the page. This is used later by the calculation function `CALC_FSLTAB_VALUES`.

In this case, the validation function stores the numeric value of the control in a variable `INT`, which will subsequently be used by the calculation functions.

When the page is posted back to the server, ASP.NET executes its own built-in validation controls and then calls the functions associated with the `CustomValidator` controls, in the order they are defined on the page. In addition to the `VALIDATE_INT` function, there are eight other custom validation functions. Three of these, which validate the *Initial Age*, *Endowment Term* and *Initial Duration* fields, are listed below. Note that all of the `VALIDATE_xxx` functions have the same .NET signature as `VALIDATE_INT`.

```
     ∇ VALIDATE_AGE MSG;source;args
[1]    ⍝ Validates Age
[2]    :Access Public
[3]    :Signature VALIDATE_AGE Object source,
                  ServerValidateEventArgs args
[4]    source args←MSG
[5]    :Trap 0
[6]        AGE←Convert.ToInt32 args.Value
[7]    :Else
[8]        args.IsValid←0
[9]        :Return
[10]   :EndTrap
[11]   args.IsValid←(10≤AGE)^80≥AGE
     ∇
```

`VALIDATE_AGE` is similar to `VALIDATE_INT`, except that, because it expects an integer value, it uses the `ToInt32` method instead of the `ToDouble` method.

`VALIDATE_TERM`, which validates the *Endowment Term* field, is slightly more interesting because there are two levels of checking involved. The first check that the user has entered an integer number, is performed by lines [10-15] in the same way as in the previous examples, using the `ToInt32` method of the `System.Convert` class within a `:Trap` control structure. However, validation of the *Endowment Term* field depends upon the value of another field, namely *Initial Age*.

Not only must the user enter an integer, but also its value must be between 10 and (90-`AGE`) where `AGE` is the value in the *Initial Age* field. However, if the user has entered an incorrect value in the *Initial Age* field, this, the second level of validation cannot be performed.

```
     ∇ VALIDATE_TERM MSG;source;args
[1]    ⍝ Validates Endowment Term
[2]    :Access Public
[3]    :Signature VALIDATE_TERM Object source,
                  ServerValidateEventArgs args
[4]    source args←MSG
[5]    :If ^/(RFVAGE CustomValidator_AGE).IsValid
[6]          source.ErrorMessage←'Endowment Term must
             be an integer between 10 and ',(⍕90-AGE),
             ' (90-Age)'
[7]    :Else
[8]        source.ErrorMessage←'Endowment Term must
           be an integer between 10 and (90-Age)'
[9]     :EndIf
[10]    :Trap 0
[11]       TERM←Convert.ToInt32 args.Value
[12]   :Else
[13]       args.IsValid←0
[14]       :Return
[15]   :EndTrap
[16]   :If ^/(RFVAGE CustomValidator_AGE).IsValid
[17]       args.IsValid←(TERM≥10)^TERM≤90-AGE
[18]   :EndIf
     ∇
```

At this stage it is worth reviewing the sequence of events that occurs when a user action in the browser causes a *postback* to the server.

1. The page, including all the contents of its fields, is sent back to the ASP.NET server using an http POST command.
2. The postback causes the creation of a new instance of the page; which is represented by a new clone of the `actuarial` namespace.
3. The creation of a new page instance raises the `Page_Load` event which in turn invokes the `Page_Load` method associated with the Page class, or an override method is one is specified. In this case, it calls our `Page_Load` function in the newly cloned instance of the `actuarial` namespace. The `Page_Load` function typically deals with initialisation, such as opening a component file or establishing a connection to a data source. In this case, it does nothing on a postback.
4. Because the *Calculate* button was pressed (see *Forcing Validation*), each of the `CustomValidator` controls on the page raises an `OnServerValidate` event, which in turn calls the associated function in the current instance of the page. These events occur in the order the controls are defined within the page. Note that built-in validation controls, including any `RequiredFieldValidator` controls, are invoked first, potentially in the browser prior to the postback.
5. Because the *Calculate* button was pressed (see *Forcing Validation*), each of the `CustomValidator` controls on the page raises an `OnServerValidate` event, which in turn calls the associated function in the current instance of the page. These events occur in the order the controls are defined within the page. Note that built-in validation controls, including any `RequiredFieldValidator` controls, are invoked first, potentially in the browser prior to the postback.
6. The control that caused the postback raises an appropriate event, which in turn fires the associated callback function.
7. After all the control events have been raised and processed the `Page_UnLoad` event is raised and the associated function (if any) is invoked. This function is a good place to implement termination code, such as closing a component file or data source.
8. The instance of the page is destroyed. Any global variables in the namespace that were defined by the Page_Load function, the validation functions and the callback function are lost because the clone of the `actuarial` namespace disappears.

This means that within the life of the cloned instance of the actuarial namespace, the system runs our `Page_Load` function followed by `VALIDATE_INT`, followed by `VALIDATE_AGE`, `VALIDATE_TERM`, `VALIDATE_DUR` etc. and finally by `CALC_FSLTAB_RESULTS`. These functions take their input from the values passed in their arguments (as in the case of the `VALIDATE_xxx` functions) or from the properties of any

of the controls on the Page. They perform output by modifying these properties, or by invoking standard methods on the Page.

Notice that, if successful, the `VALIDATE_INT` function set up a global variable (strictly speaking, only global within the current instance of the actuarial namespace) called `INT` that contains the value in the *Interest Rate* field. Similarly, `VALIDATE_AGE` defines a variable called `AGE`. These variables are subsequently available for use by the calculation function.

This technique, of having each validation function define a variable for its associated field, saves repeating the conversion work in the calculation routine `CALC_FSLTAB_RESULTS` that will be called when the validation is complete. It also saves repeating the conversion work in a validation routine that needs to know the value of a previously validated field.

Returning to the explanation of `VALIDATE_TERM`, line [16] checks to see that both the `RequiredFieldValidator` and `CustomValidator` controls for the *Initial Age* field register that the value in the field is valid, before attempting to perform the second stage of the validation which depends upon `AGE`. Note that `AGE` must exist (and be a reasonable value) if `CustomValidator_AGE.IsValid` is true. Notice too that it is insufficient just to check the `CustomValidator` control, because its validation function will not be invoked (and the control will register that the field is valid) if the field is empty.

Line [5] uses similar logic to set up an appropriate error message, which is assigned to the `ErrorMessage` property of the corresponding `CustomValidator` control, represented by `source`.

`VALIDATE_DUR`, which validates the *Initial Duration* field, uses similar logic to check that the value in the *Endowment Term* field is correct and that `TERM`, on which it depends, is therefore defined. In addition, in line [8] it refers to the `Checked` property of the `RadioButton` controls named `TA` and `TB` respectively.

```
    ∇ VALIDATE_DUR MSG;source;args;DT
[1]    ⍝ Validates Initial Duration
[2]    :Access Public
[3]    :Signature VALIDATE_DUR Object source,
                  ServerValidateEventArgs args
[4]    source args←MSG
[5]    :If 2=GET_RUN_OPTION
[6]       DT←1
[7]   :Else
[8]      DT←+/10 1×(TA TB).Checked
[9]   :EndIf
[10]  :If ^/(RFVTRM CustomValidator_TERM).IsValid

[11]     source.ErrorMessage←'Initial Duration must be an
         integer between 0 and ',(⍕TERM-DT),
         ' (TERM-',(⍕DT),')'
[12]  :Else
[13]     source.ErrorMessage←'Initial Duration must be an
            integer between 0 and (Term-',(⍕DT),')'
[14]  :EndIf
[15]  :Trap 0
[16]     DUR←Convert.ToInt32 args.Value
[17]  :Else
[18]     args.IsValid←0
[19]     :Return
[20]  :EndTrap
[21]  :If ^/(RFVTRM CustomValidator_TERM).IsValid
[22]     args.IsValid←(0≤DUR)^DUR≤TERM-DT
[23]  :EndIf
    ∇
```

Forcing Validation

Validation controls are automatically invoked when the user activates a Button control, but not when other postbacks occur. For example, when the user selects a different Mortality Table (represented by a `RadioButtonList` control), the page calls the `CHANGE_TABLES` function.

```
<asp:RadioButtonList id=MT runat="server"
     RepeatDirection="Vertical" RepeatRows="3"  tabIndex=1
     onSelectedIndexChanged="CHANGE_TABLES"
     AutoPostBack="true">
<asp:ListItem Value="UK Assured Lives">
     Selected="True">UK Assured Lives</asp:ListItem>
<asp:ListItem Value="UK Immediate Annuitant">
     UK Immediate Annuitant</asp:ListItem>
<asp:ListItem Value="UK Pension Annuitant">
     UK Pension Annuitant</asp:ListItem>
</asp:RadioButtonList>
```

A `RadioButtonList` control does not cause validation to occur, so this must be done explicitly. This is easily achieved by calling the `Validate` method of the Page itself as shown in `CHANGE_TABLES[11]` below.

```
     ∇ CHANGE_TABLES ARGS;TableNames;TableName;OPTSMORT;
                MORT_OPTION;RUN_OPTION
[1]   :Access public
[2]   :Signature CHANGE_TABLES  Object obj, EventArgs ev
[3]    RUN_OPTION←GET_RUN_OPTION
[4]    MORT_OPTION←1+MT.SelectedIndex
[5]    OPTSMORT←MORT_OPTION⊃OPTSMORT_ASS OPTSMORT_ANNI
                          OPTSMORT_ANNP
[6]    TableNames←⊃OPTSMORT       ⍝ Assured lives/term
                                     assurance tables
[7]    TableNames←↓(2=⎕NC 0 1↓3⊃OPTSMORT)⌿TableNames
[8]    TableNames←TableNames~¨' '
[9]    CMTAB.Items.Clear
[10]   :For TableName :In TableNames
[11]       CMTAB.Items.Add TableName
[12]   :EndFor
[13]   Page.Validate ⍝ Force page validation
[14]   :Select RUN_OPTION
[15]   :Case 1
[16]       CALC_FSLTAB_RESULTS ⍬
[17]   :Case 2
[18]       CALC_FSL_RESULTS ⍬
[19]   :EndSelect
     ∇
```

Calculating and Displaying Results

The function `CALC_FSLTAB_RESULTS`, which for brevity is only partially shown below, is used by the `sla_tab.aspx` page to calculate and display results.

```
    ∇ CALC_FSLTAB_RESULTS ARGS;X;ULT;MORTOPT;QTAB;TABLE;
        TAB_DURS;RUN_OPTION;MORT_OPTION;UNIX;DOS;
        CURRENTDATE;CURRENTTIME;OPTSMORT;TABLES;MSG;data
[1]     :If IsValid ⍝ Is page valid ?
...
[6]         MORT_OPTION←1+MT.SelectedIndex
[7]         OPTSMORT←MORT_OPTION⊃OPTSMORT_ASS
                              OPTSMORT_ANNI
                              OPTSMORT_ANNP
[8]
[9]         TABLES←↓3⊃OPTSMORT
[10]        MORTOPT←(⍴TABLES)⍴0
[11]        MORTOPT[1+CMTAB.SelectedIndex]←1
[12]        TABLE←⊃MORTOPT/TABLES
...
[15]        TAB_DURS←TA.Checked
...
[41]        FSLT←((⍴X)⍴(3 0)(3 0)(3 0)(11 4)(11 6)(12 4)
                 (11 6)(8 0))⍪¨X
[42]        FSLT←FSLT~¨' '
[43]        :With data←⎕NEW DataTable
[44]            cols←Columns.Add¨⊂¨##.FSL_HEADER
[45]            {
[46]                row←NewRow ⍬
[47]                row.ItemArray←⍵
[48]                Rows.Add row
[49]            }¨↓##.FSLT
[50]        :EndWith
[51]        fsl.DataSource←⎕NEW DataView data
[52]        fsl.DataBind
[53]        fsl.Visible←1
[54]    :Else
[55]        fsl.Visible←0
[56]    :EndIf
    ∇
```

The results of the calculation are displayed in a DataGrid object named `fsl`. This is defined within the `sla_tab.aspx` page as follows:

```
<asp:DataGrid id="fsl" runat="server" Width="700"
     AllowPaging="false" BorderColor="black" CellPadding="3"
     CellSpacing="0" Font-Size="9pt" PageSize="10">
     <ItemStyle HorizontalAlign="right" Width="100">
     </ItemStyle>
     <HeaderStyle HorizontalAlign="center"
     Font-Size="12pt" Font-Bold="true" BackColor="#17748A"
     ForeColor="#FFFFFF"></HeaderStyle>
     </asp:DataGrid>
```

`CALC_FSLTAB_RESULTS[1]` checks to see if the user input is valid. If not, `[55]` hides the `DataGrid` object `fsl` so that no results are displayed in the page. The display of error messages is handled separately, and automatically, by the `ValidationSummary` control on the page.

`CALC_FSLTAB[11 15]` obtain the values of the *CMTAB* (`DropDownList`) and *TA* (`RadioButton`) controls on the page.

`CALC_FSLTAB[43-53]` store the calculated data table `FSLT` in the `DataGrid` `fsl`.

11 Writing Custom Controls for ASP.NET

11.1 Introduction

The previous chapter showed how you can build ASP.NET Web Pages by combining APL code with the Web Controls provided in the .NET Namespace `System.Web.UI.WebControls`. These controls are in fact just ordinary .NET classes. In particular, they are extensible components that can be used to develop more complex controls that encapsulate additional functionality.

This chapter describes how you can go about building custom server-side controls, for deployment in ASP.NET Web Pages.

A custom control is simply a .NET class that inherits from the `Control` class in the .NET Namespace `System.Web.UI`, or inherits from a higher class that is itself based upon the `Control` class. Like any other .NET class, a custom control is implemented in an assembly, physically as a DLL file. This chapter explores three different ways to implement a custom control.

The `Control` class provides a `Render` method whose job is to generate the HTML that defines appearance of the control. The first example, the `SimpleCtl` control, overrides the `Render` method to display a simple string "Hello World" in the browser.

The `TemperatureConverterCtl1` control is an example of a compositional control, that is, one that is composed of other standard controls packaged with special functionality. The `TemperatureConverterCtl2` control, uses the basic approach of the `SimpleCtl` control, but provides the same functionality as `TemperatureConverterCtl1`. The `TemperatureConverterCtl3` control illustrates how to generate events for the hosting page to catch and process.

These examples, which are based upon a series of articles called *Advanced ASP.NET Server-Side Controls* by George Shepherd that appeared in the *msdn magazine* (October 2000, January 2001 and March 2001 issues), are implemented as Dyalog classes in a namespace called `DyalogSamples` in the workspace `samples\asp.net\temp\bin\temp.dws`. The corresponding .NET Assembly `samples\asp.net\temp\bin\temp.dll` was generated from this workspace.

```
      )LOAD "C:\Program Files (x86)\Dyalog\Dyalog APL 15.0
Unicode\Samples\asp.net\temp\bin\temp.dws"
```

```
C:\Program Files (x86)\Dyalog\Dyalog APL 15.0
Unicode\Samples\asp.net\temp\bin\temp.dws saved Tue Nov 22 15:04:11 2016
```

```
      )OBS
DyalogSamples
      )CS DyalogSamples
#.DyalogSamples
      )CLASSES
SimpleCtl       TemperatureConverterCtl1
TemperatureConverterCtl2        TemperatureConverterCtl3
```

11.2 The SimpleCtl Control

The `SimpleCtl` Class is illustrated below:

```
:Class SimpleCtl: Control
:Access public
:Using System
:Using System.Collections.Specialized,System.dll
:Using System.Web,System.Web.dll
:Using System.Web.UI
:Using System.Web.UI.WebControls
:Using System.Web.UI.HtmlControls

    ∇ Render output;HTML
      :Access public override
      :Signature Render  HtmlTextWriter output
      HTML←'<h3>Hello World</h3>'
      output.WriteLine⊂HTML
    ∇

:EndClass ⍝ SimpleCtl
```

The `Render` function **supercedes** (see *Programming: Access*) the `Render` method that `SimpleCtl` has inherited from its base class, `System.Web.UI.Control`.

The `Render` method defined by the `System.Web.UI.Control` base class is `void` and takes a parameter of type `HtmlTextWriter`. When the `SimpleCtl` control is referenced in a Web Page, ASP.NET creates an instance of it and calls its `Render` method because it is a `Control` and is expected to have one. Moreover, ASP.NET supplies an object of type `HtmlTextWriter` as its parameter. You do not need to worry where this object came

from, or what it actually represents. You need only know that an `HtmlTextWriter` provides a method called `WriteLine` that may be used to output a text string to the browser. The mechanics of how this actually happens are handled by the `HtmlTextWriter` object itself.

In APL terms, the argument to our `Render` function, `output`, will be a namespace reference, and the function can simply call its `WriteLine` method with a character vector argument. This argument can contain any valid HTML string and defines the appearance of the `SimpleCtl` control.

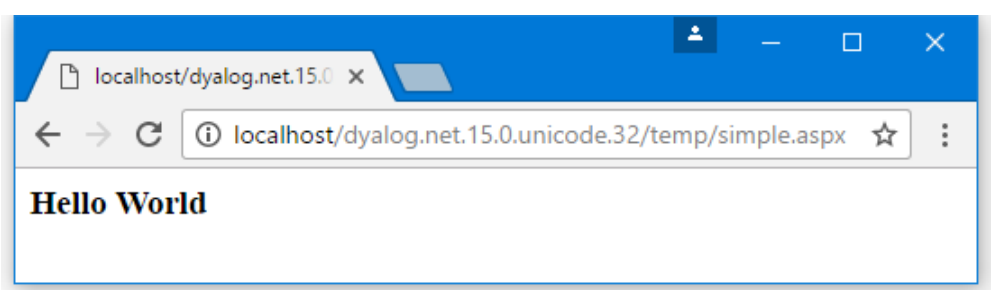
Using the `:Signature` statement, the `Render` function is defined to have the same syntax as the method it overrides, that is, it does not return a result `void` and takes a single parameter of type `HtmlTextWriter`. Note that to successfully replace the base class method, the `Render` function must have exactly this *:Signature*.

Using SimpleCtl

Our `SimpleCtl` control may now be included in any .NET Web Page from which `temp.dll` is accessible. The file `samples\asp.net\temp\Simple.aspx` is simply an example. The fact that this control is written in Dyalog is immaterial.

```
<%@ Register TagPrefix="Dyalog"
             Namespace="DyalogSamples" Assembly="temp" %>

<html>
<body>
<Dyalog:SimpleCtl runat=server/>
</body>
</html>
```

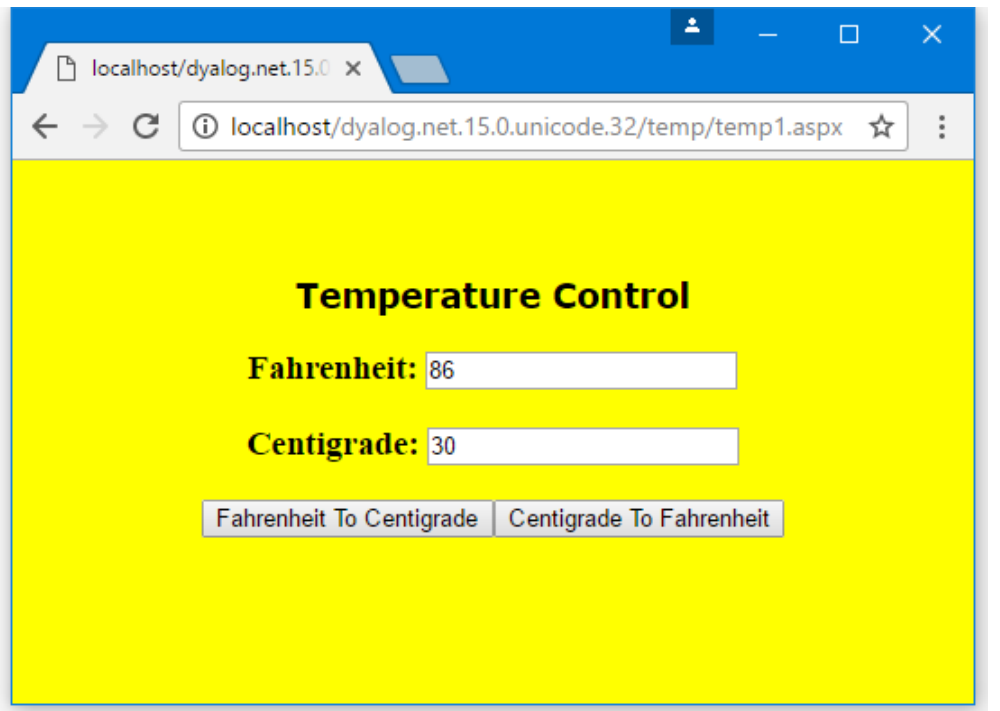
The first line of the script specifies that any controls referenced later in the script that are prefixed by `Dyalog:`, refer to custom controls in the .NET Namespace called `DyalogSamples` which is located in the Assembly `temp.dll` in the `bin` subdirectory.



11.3 The TemperatureConverterCtl1 Control

The `TemperatureConverterCtl1` control is an example of a *compositional* control, that is, a server-side custom control that is composed of other standard controls.

In this example, The `TemperatureConverterCtl1` control gathers together two textboxes and two push buttons into a single component as illustrated below. Type a number into the *Centigrade* box, click the *Centigrade To Fahrenheit* button, and the control converts accordingly. If you click the *Fahrenheit To Centigrade* button, the reverse conversion is performed.



The `TemperatureConverterCtl1` control *contains* other standard controls as *child controls*. A control that acts as a container must implement an interface called `INamingContainer`.

This interface does not in fact require any methods; it merely acts as a marker. So the `:Class` statement specifies that it provides this interface:

```
:Class TemperatureConverterCtl1: Control,
                  System.Web.UI.INamingContainer
```

Child Controls

Whenever ASP.NET initialises a `Control`, it calls its `CreateChildControls` method. The default `CreateChildControls` method does nothing). So we simply define a function called `CreateChildControls` with the appropriate public interface (no arguments and no result) as shown below.

```
     ∇ CreateChildControls
[1]    :Access Public override
[2]    :Signature CreateChildControls
[3]
[4]    Controls.Add ⎕NEW LiteralControl,⊂'<h3>Fahrenheit: '
[5]       m_FahrenheitTextBox←⎕NEW TextBox
[6]       m_FahrenheitTextBox.Text←,'0'
[7]       Controls.Add m_FahrenheitTextBox
[8]    Controls.Add ⎕NEW LiteralControl,⊂'</h3>'
[9]
[10]   Controls.Add ⎕NEW LiteralControl,⊂'<h3>Centigrade: '
[11]      m_CentigradeTextBox←⎕NEW TextBox
[12]      m_CentigradeTextBox.Text←,'0'
[13]      Controls.Add m_CentigradeTextBox
[14]   Controls.Add ⎕NEW LiteralControl,⊂'</h3>'
[15]
[16]   F2CButton←⎕NEW Button
[17]   F2CButton.Text←'Fahrenheit To Centigrade'
[18]   F2CButton.onClick←⎕OR'F2CConvertBtn_Click'
[19]   Controls.Add F2CButton
[20]
[21]   C2FButton←⎕NEW Button
[22]   C2FButton.Text←'Centigrade To Fahrenheit'
[23]   C2FButton.onClick←⎕OR'C2FConvertBtn_Click'
[24]   Controls.Add C2FButton
     ∇
```

Line[4] creates an instance of a `LiteralControl` (a label) containing the text "Fahrenheit" with an HTML tag "<h3>". `Controls` is a property of the `Control` class (from which `TemperatureConverterCtl1` inherits) that returns a `ControlCollection` object This has an `Add` method whose job is to add the specified control to the list of child controls managed by the object.

Lines[5-6] create a `TextBox` child control containing the text "0", and Line[7] adds it to the child control list.

Line[8] adds a second `LiteralControl` to terminate the "<h3>" tag.

Lines [10-14] do the same for Centigrade.

Lines[16-17] create a `Button` control labelled "Fahrenheit To Centigrade". Line[18] associates the callback function `F2CConvertBtn_Click` with the button's `onClick` event. Note that it is necessary to assign the `⎕OR` of the function rather than its name. Line[19] adds the button to the list of child controls.

Lines[21-24] create a Centigrade button in the same way.

This function is run every time the page is loaded; however in a postback situation, other code steps in to modify the values in the textboxes, as we shall see.

Fahrenheit and Centigrade Values

The `TemperatureConverterCtl1` maintains two public properties named `CentigradeValue` and `FahrenheitValue`, which may be accessed by a client application. These properties are not exposed directly as variables, but are obtained and set via *property get* (or *accessor*) and *property set* (or *mutator*) functions. (This is recommended practice for C#, so the example shows how it is done in APL). In this case, the values are simply stored in or obtained directly from the corresponding textboxes set up by `CreateChildControls`.

```
      :Property CentigradeValue
        ∇ C←get
          :Access Public
          :Signature Double←get_CentigradeValue
          C←⍎m_CentigradeTextBox.Text
        ∇

        ∇ set C
          :Access Public
          :Signature set_CentigradeValue Double Value
          m_CentigradeTextBox.Text←⍕C.NewValue
        ∇
    :EndProperty ⍝ CentigradeValue
```

Notice that the `Get` function uses `⍎` to convert the text in the textbox to a numeric value. Clearly something more robust would be called for in a real application

Similar functions to handle the `Fahrenheit` property are provided but are not shown here.

Responding to Button presses

We have seen how APL callback functions have been attached to the `onClick` events in the two buttons. The `C2FconvertBtn_Click` callback function simply obtains the `CentigradeValue` property, converts it to Fahrenheit using `C2F`, and then sets the `FahrenheitValue` property.

```
     ∇ C2FConvertBtn_Click args
      :Access Public
      :Signature C2FConvertBtn_Click Object,EventArgs
      FahrenheitValue←C2F CentigradeValue
     ∇

     ∇ f←C2F c
[1]    f←32+c×1.8
     ∇

     ∇ F2CConvertBtn_Click args
      :Access Public
      :Signature F2CConvertBtn_Click Object ,EventArgs
      CentigradeValue←F2C FahrenheitValue
     ∇

     ∇ c←F2C f
[1]    c←(f-32)÷1.8
     ∇
```

The `F2CconvertBtn_Click` callback function converts from Fahrenheit to Centigrade. Note that the functions `C2F` and `F2C` are internal functions that are private to the control, and it is therefore not necessary to define public interfaces for them.

Using the Control on the Page

The text of the script file `samples\temp\temp1.aspx` is shown below. There is really no difference between this example and the `simple.aspx` described earlier.

```
<%@ Register TagPrefix="Dyalog" Namespace="DyalogSamples"
                                Assembly="TEMP"%>
<html>
<body bgcolor="yellow">
<br><br>
<center>
<h2><font face="Verdana" color="black">Temperature Control</font></h2>

<form runat=server>
<Dyalog:TemperatureConverterCtl1 id=TempCvtCtl1 runat=server/>
</form>
</center>
</body>
</html>
```

The HTML generated by the control at run-time is shown below. Notice that in place of the server-side control declaration in `temp1.aspx`, there are two edit controls with numerical values in them, and two push buttons to submit data entered on the form to the server.

```
<html>
<body bgcolor="yellow">
<br><br>
<center>
<h2><font face="Verdana" color="black">Temperature Control</font></h2>

<form name="ctrl1" method="post" action="temp1.aspx" id="ctrl1">
<input type="hidden" name="__VIEWSTATE" value="YTB6MTc3MzAxNzYxNF9fX3g=0
3f01d88" />

<h2>Fahrenheit: <input name="TempCvtCtl1:ctrl1" type="text" value="32" /
></h2><h2>Centigrade: <input name="TempCvtCtl1:ctrl4" type="text"
value="0" /></h2><input type="submit" name="TempCvtCtl1:ctrl6" value="Fa
hrenheit To Centigrade" /><input type="submit" name="TempCvtCtl1:ctrl7"
value="Centigrade To Fahrenheit" />
</form>

</center>
</body>
</html>
```

11.4 The TemperatureConverterCtl2 Control

The previous example showed how to compose an ASP.NET custom control from other standard controls. This example shows how you can instead generate standard form elements on the browser by rendering the HTML for them directly.

In the composite temperature control `TemperatureConverterCtl1`, discussed previously, all the data transfers between the browser and the server, relating to the standard child controls that it contains, are handled automatically by the controls themselves. Rendered controls require a bit more programming because it is up to the control developer to do the data transfer. The data transfer is managed through two interfaces, namely `IPostBackDataHandler` and `IPostBackEventHandler`. We will see how these interfaces are used later.

The `:Class` statement for `TemperatureConverterCtl2` specifies that it provides these interfaces.

```
:Class TemperatureConverterCtl2: Control,
                  System.Web.UI.IPostBackDataHandler,
                  System.Web.UI.IPostBackEventHandler
```

Fahrenheit and Centigrade Values

Like the previous `TemperatureConverterCtl1` control, the `TemperatureConverterCtl2` maintains two public properties named `CentigradeValue` and `FahrenheitValue` using property *get* and property *set* functions.

This time, the control manages the current temperature values in two internal variables named `_CentigradeValue` and `_FahrenheitValue`, which we must initialise.

```
      _CentigradeValue←0
      _FahrenheitValue←0
```

The `CentigradeValue`'s `get` function simply returns the current value of `_CentigradeValue`. Its .NET Properties are defined as shown so that it is exported as a *property get* function for the `CentigradeValue` property, and returns a `Double`.

```
       ∇ C←get
        :Access Public
        :Signature Double←get
        C←_CentigradeValue
       ∇
```

The `CentigradeValue`'s `set` function simply resets the value of `_CentigradeValue` to that of its argument. Its .NET Properties are defined as shown so that it is exported as a *property set* function for the `CentigradeValue` property, and takes a `Double`.

```
      ∇ set C
       :Access Public
       :Signature set Double Value
       _CentigradeValue←C.NewValue
      ∇
```

The property *get* and property *set* functions for the `FahrenheitValue` property are similarly defined. The .signatures for these functions are similar to those for the `CentigradeValue` functions and are not shown.

Rendering the Control

Like the `SimpleCtl` example described earlier in this Chapter, the `TemperatureConverterCtl2` control has a `Render` function that generates the HTML to represent its appearance, and in this case its behaviour too.

```
     ∇ Render output;C;F;BF;CF
[1]    :Access Public override
[2]    :Signature Render HtmlTextWriter output
[3]
[4]    F←'<h3>Fahrenheit <input name='
[5]    F,←UniqueID
[6]    F,←' id=FahrenheitValue type=text value='
[7]    F,←⍕_FahrenheitValue
[8]    F,←'></h3>'
[9]    output.Write⊂F
[10]
[11]   C←'<h3>Centigrade <input name='
[12]   C,←UniqueID
[13]   C,←' id=CentigradeValueKey type=text value='
[14]   C,←⍕_CentigradeValue
[15]   C,←'></h3>'
[16]   output.Write⊂C
[17]
[18]   BF←'<input type=button value=FahrenheitToCentigrade '
[19]   BF,←' onClick="jscript:'
[20]   BF,←Page.GetPostBackEventReference ⎕THIS'FahrenheitToCentigrade'
[21]   BF,←'">'
[22]   output.Write⊂BF
[23]
[24]   CF←'<input type=button value=CentigradeToFahrenheit '
[25]   CF,←' onClick="jscript:'
[26]   CF,←Page.GetPostBackEventReference ⎕THIS'CentigradeToFahrenheit'
[27]   CF,←'">'
[28]   output.Write⊂CF
[29]
[30]   output.WriteLine∘⊂⍨'' '<br>' '<br>'
     ∇
```

As we saw in the `SimpleCtl` example, the `Render` method will be called by ASP.NET with a parameter that represents an `HtmlTextWriter` object. This is represented by the APL local name `output`.

Lines[4-9] and lines [11-16] generate HTML that defines two text boxes in which the user may enter the Fahrenheit and centigrade values respectively. Lines[9&16] use the `Write` method of the `HtmlTextWriter` object to output the HTML.

Lines[5&12] obtain the fully qualified identifier for this particular instance of the `TemperatureConverterCtl2` control from its `UniqueID` property. This is a property,

which it inherits from `Control` and is therefore also a property of the current (APL) namespace.

Lines[18-22] and Lines[24-28] generate and output the HTML to represent the two buttons that convert from Fahrenheit to Centigrade and from Centigrade to Fahrenheit respectively.

Lines[19-20] and [25-26]generate HTML that wires the buttons up to JavaScript handlers to be executed *by the browser*. The JavaScript simply causes the browser to execute a postback, that is, send the page contents back to the server. `GetPostBackEventReference` is a (shared) method provided by the `System.Web.UI.Page` class that generates a reference to a client-side script function. In this case it is called with two parameters, an object that represents the current instance of the `TemperatureConverterCtl2` control, and a string that will be passed to the server to indicate the cause of the postback (that is, which button was pressed). The first parameter is a reference to the current object, which is returned by the system function `⎕THIS`.

The client-side script is itself generated, and inserted into the HTML stream automatically.

To help to understand this process fully, it is instructive to examine the HTML that is generated by these functions. We will do this a bit later in the Chapter.

Loading the Posted Data

Once the server-side control has rendered the HTML for the browser, the user is free to type numbers into the text boxes and to press the buttons.

When the user presses a button, the browser runs the client-side JavaScript code that in turn generates a postback to the server.

The `:Class` statement for `TemperatureConverterCtl2` specifies that it supports the `IPostBackDataHandler` interface. This interface must be implemented by controls that want to receive postback data (that is, the contents of Form fields that the user may have entered or changed) `IpostBackDataHandler` has two methods `LoadPostData` and `RaisePostDataChangedEvent`. `LoadPostData` is automatically invoked when a postback occurs, and the postback data is supplied as a parameter.

So when the postback occurs, the server reloads the original page and, because this is a postback situation and our control has advertised the fact that it implements `IPostBackDataHandler`, ASP.NET invokes its `LoadPostBack` method. This method is called with two parameters. The first is a key and the second is a collection of name/

value pairs. This contains the names of all the Form fields on the page (and there may be others not directly associated with our custom control) and the values they had when the user pressed the button. The key provides the means to extract the relevant part of this collection. The `LoadPostData` function is shown below.

```
     ∇ R←LoadPostData args;postDataKey;values;controlValues;new
[1]    :Signature Boolean←IPostBackDataHandler.LoadPostData String postD
ataKey,NameValueCollection values
[2]    postDataKey values←args
[3]    controlValues←values[⊂postDataKey]
[4]    new←ParseControlValues controlValues
[5]    R←∨/new≠_FahrenheitValue _CentigradeValue
[6]    _FahrenheitValue _CentigradeValue←new
     ∇
```

Line[2] obtains the two parameters from the argument and Line[3] uses the key to extract the appropriate data from the collection. `ControlValues` is a comma-delimited string containing name/value pairs. The function `ParseControlValues` simply extracts the values from this string, that is, the contents of the Fahrenheit and Centigrade text boxes.

Postback Events

The result of `LoadPostData` is Boolean and indicates whether or not any of the values in a control have changed. If the result is `True` (1), ASP.NET will next call the `RaisePostDataChanged` method. This method is called with no parameters and merely signals that something has changed. The control knows *what* has changed by comparing the old with the new, as in `LoadPostData[5]`.

Finally, the page framework calls the `RaisePostBackEvent` method, passing it a string that identifies the page element that caused the post back.

The objective of these calls is to provide the control with the information it requires to synchronise its internal state with its appearance in the browser.

In this case, we are not interested in which of the two text box values the user has altered; what matters is which of the two buttons *FarenheitToCentigrade* or *CentigradeToFarenheit* was pressed. Therefore, in this case, the control uses `RaisePostBackEvent` rather than `RaisePostDataChanged` (or indeed, `LoadPostData` itself, which is another option). The reason is that `RaisePostBackEvent` receives the name of the button as its argument.

So in our case, the `RaisePostDataChanged` function does nothing. Nevertheless, it is essential that the function is provided and essential that it supports the correct public interface, namely that it takes no arguments are returns no result (`Void`).

```
     ∇ RaisePostDataChangedEvent
[1]  :Access public
[2]  :Signature RaisePostDataChangedEvent
[3]  ⍝ Do nothing
     ∇
```

The `RaisePostBackEvent` function simply switches on its argument, which is the name of the button that the user pressed, and recalculates `_CentigradeValue` or `_FahrenheitValue` accordingly.

```
     ∇ RaisePostBackEvent eventArgument
[1]    :Access public
[2]    :Signature RaisePostBackEvent  String eventArg
[3]    :Select eventArgument
[4]    :Case 'FahrenheitToCentigrade'
[5]        _CentigradeValue←F2C _FahrenheitValue
[6]    :Case 'CentigradeToFahrenheit'
[7]        _FahrenheitValue←C2F _CentigradeValue
[8]    :EndSelect
     ∇
```

Finally, the page framework calls the `OnPreRender` and `Render` functions again, which generate new HTML for the browser.

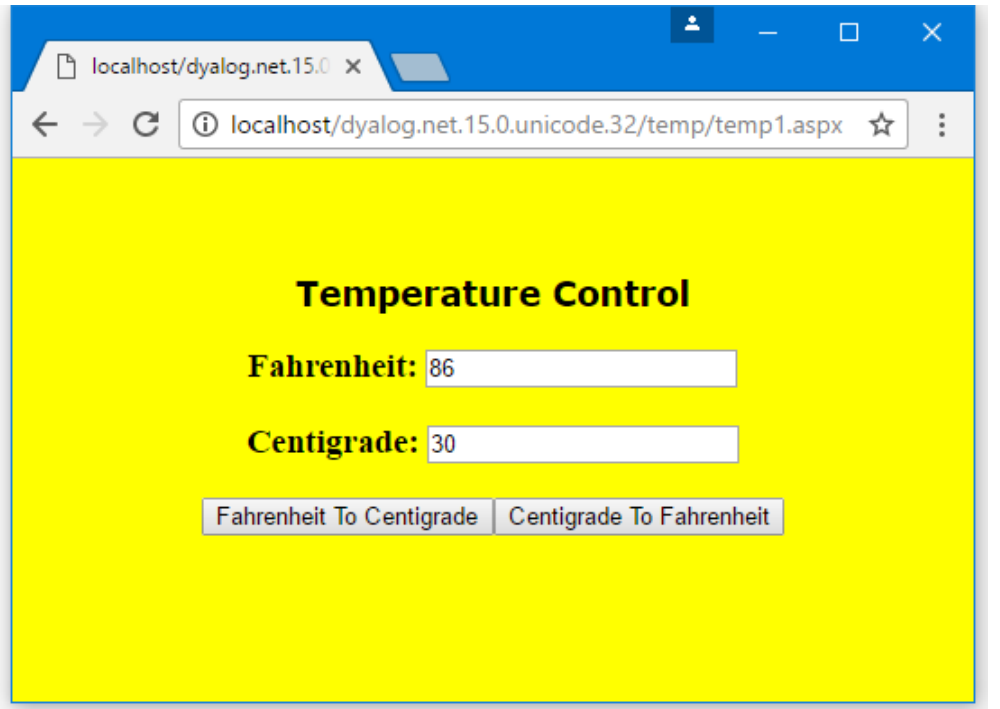
Using the Control on a Page

So long as it has access to this `DLL`, our custom control may be accessed from any ASP.NET Web Page, and a simple example is shown below.

```
<%@ Register TagPrefix="Dyalog" Namespace="DyalogSamples"
                                Assembly="TEMP" %>

<html>
<body bgcolor="yellow">
<center>
<h2><font face="Verdana" color="black">
Temperature Control</font></h3>
<h3><font face="Verdana" color="black">
Server-Side Noncompositional control</font></h4>

<form runat=server>
<Dyalog:TemperatureConverterCtl2 id=TempCvtCtl2
runat=server/>
</form>

</center>
</body>
</html>
```

The HTML that is generated by the control is illustrated below. Notice the presence of a JavaScript function named `__doPostBack`. This is generated by the `RegisterPostBackScript` method called from the `OnPreRender` function. The code that wires the buttons to this function was generated by the `GetPostBackEventReference` method called from the `Render` function.

```
<html>
<body bgcolor="yellow">
<center>
<h2><font face="Verdana" color="black">Temperature Control</font></h2>
<h4><font face="Verdana" color="black">Server-Side Noncompositional
control</font></h4>


<form name="ctrl1" method="post" action="temp2.aspx" id="ctrl1">
<input type="hidden" name="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" value="" />
<input type="hidden" name="__VIEWSTATE" value="YTB6MTc3MzAxNzYxM19fX3g=9
cfcfa5c" />

<script language="javascript">
<!--
    function __doPostBack(eventTarget, eventArgument) {
        var theform = document.ctrl1
        theform.__EVENTTARGET.value = eventTarget
        theform.__EVENTARGUMENT.value = eventArgument
        theform.submit()
    }
// -->
</script>

<h2>Fahrenheit <input name=TempCvtCtl2 id=FahrenheitValue type=text valu
e=0></h2><h2>Centigrade <input name=TempCvtCtl2 id=CentigradeValueKey ty
pe=text value=0></h2><input type=button value=FahrenheitToCentigrade  on
Click="jscript:__doPostBack('TempCvtCtl2','FahrenheitToCentigrade')"><in
put type=button value=CentigradeToFahrenheit  onClick="jscript:__doPostB
ack('TempCvtCtl2','CentigradeToFahrenheit')">
<br>
<br>

</form>

</center>
</body>
</html>
```



11.5 The TemperatureConverterCtl3 Control

In the previous examples, events generated by control have been internal events, that is, events that have been detected and processed internally by the control itself.

A separate requirement is to be able to design a custom control that generates *external* events, that is, events that can be detected and handled by the page that is hosting the control. This example illustrates how to do this.

The TemperatureConverterCtl3 namespace is a copy of TemperatureConverterCtl2 with a couple of changes.

The first change is that it describes an event that the control is going to generate. This is done using ⎕NQ inside TemperatureConverterCtl3 like this:

```
2 ⎕NQ '' 'SetEventInfo' 'Export'
          (('Double' 'Fahrenheit')
           ('Double' 'Centigrade'))
```

In this case, the name of the event is *Export* and it will report two parameters named *Fahrenheit* and *Centigrade* which are both of data type `Double`.

This version of the control presents a slightly different appearance to the previous one. The control itself is wrapped up in an HTML *Table*, with the conversion buttons arranged in a column. These buttons generate internal events that are caught and handled by the control itself. The third row of the table contains an additional button labelled *Export* which will generate the *Export* event when pressed. The `Render` function is shown below.

```
     ∇ Render output;TableRow;HTML;SET
[1]    :Access public override
[2]    :Signature Render  HtmlTextWriter output
[3]    TableRow←{
[4]        HTML←'<tr><td>',⍺,'</td><td><input name=',UniqueID
[5]        HTML,←' id=',⍺,'Value type=text '
[6]        HTML,←'value=',(⍕⍵),'></td>'
[7]        HTML,←'<td><input type=button value=Convert'
[8]        HTML,←' onClick="jscript:'
[9]        HTML,←(Page.GetPostBackEventReference ⎕THIS ⍺),'"></td></tr>'
[10]       HTML
[11]   }
[12]
[13]   HTML←''
[14]   HTML←'<table>'
[15]   HTML,←'Fahrenheit'TableRow _FahrenheitValue
[16]   HTML,←'Centigrade'TableRow _CentigradeValue
[17]
[18]   SET←'<tr><td><input type=button value=Export '
[19]   SET,←' onClick="jscript:'
[20]   SET,←Page.GetPostBackEventReference ⎕THIS'Export'
[21]   SET,←'"></td></tr>'
[22]   HTML,←SET,'</table>'
[23]   output.Write⊂HTML
     ∇
```

Notice that `Render[18]` causes the *Export* button to generate a Postback event which will call `RaisePostBackEvent` with the argument `'Export'`. Up to now, this is just an internal event just like the events generated by the conversion buttons.

The `RaisePostBackEvent` propagates this event to the host page.

```
     ∇ RaisePostBackEvent eventArgument
[1]    :Signature IPostBackEventHandler.RaisePostBackEvent String eventA
rg
[2]    :Select eventArgument
[3]    :Case 'Fahrenheit'
[4]        _CentigradeValue←F2C _FahrenheitValue
[5]    :Case 'Centigrade'
[6]        _FahrenheitValue←C2F _CentigradeValue
[7]    :Case 'Export'
[8]        4 ⎕NQ'' 'Export'_FahrenheitValue _CentigradeValue
[9]    :EndSelect
     ∇
```

This is simply done by the third `:Case` statement, so that when the function is invoked with the argument `'Export'`, it fires an Export event. This is done by line `[8]` using `4 ⎕NQ`. The elements of the right argument are:

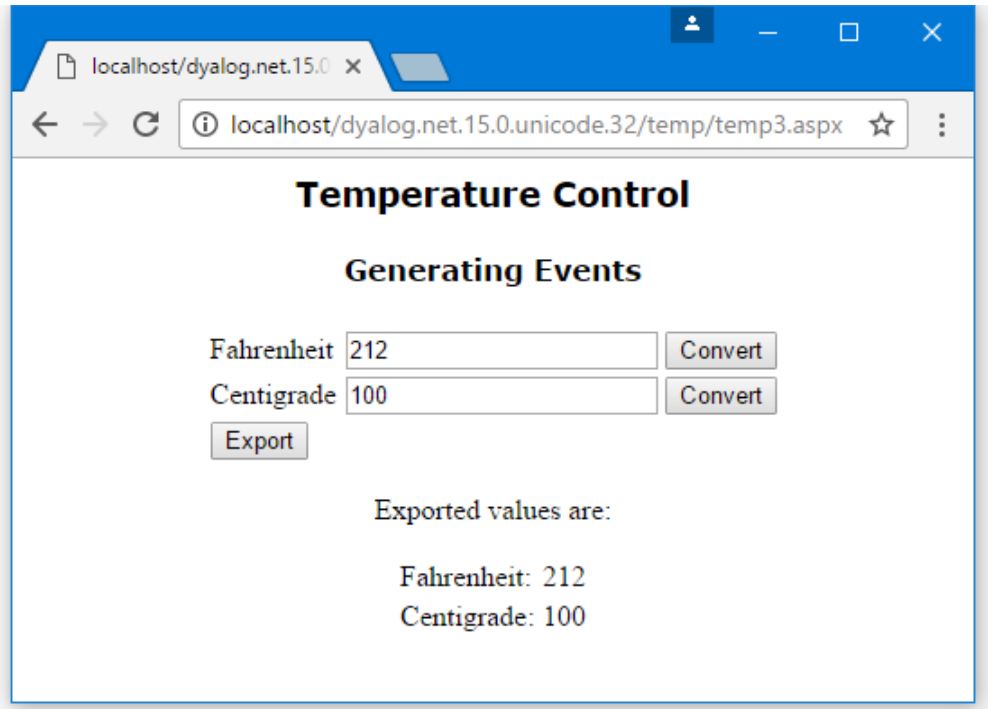
`[1]`	`''`	Specifies that the event is generated by this instance of the control
`[2]`	`'Export'`	The name of the event to be generated
`[3]`	`_FahrenheitValue`	The value of the first parameter, *Fahrenheit*
`[4]`	`_CentigradeValue`	The value of the second parameter, *Centigrade*

It is then up to the page that is hosting the control to respond to the event in whatever way it deems appropriate.

Hosting the Control on a Page

The following example illustrates an ASP.NET web page that hosts the TemperatureConverterCtl3 custom control and responds to its Export event. The page uses a section of the page. It simply sets the Text property of two Label controls to display the parameters reported by the event.

The picture below illustrates what happens when you run the page. Notice that the user can independently convert values between the two temperature scales and export these values from the control, to the host page, by pressing the Export button.


localhost/dyalog.net.15.0
localhost/dyalog.net.15.0.unicode.32/temp/temp3.aspx
Temperature Control
Generating Events
Fahrenheit 212 Convert
Centigrade 100 Convert
Export
Exported values are:
Fahrenheit: 212
Centigrade: 100

12 Implementation Details

12.1 Introduction

The Dyalog DLL is the Dyalog APL *engine* that hosts the execution of all .NET classes that have been written in Dyalog APL, including APL Web Pages and APL Web Services. The Dyalog DLL provides the interface between client applications (such as ASP.NET) and your APL code. It receives calls from client applications, and executes the appropriate APL code. It also works the other way, providing the interface between your APL code and any .NET classes that you may call.

The Development DLL (the full developer version of the Dyalog DLL) contains the APL Session, Editor, Tracer and so forth, and may be used to develop and debug an APL .NET class while it is executing. Note that to gain access to the various workspace tools, such as the Workspace Explorer and the Search/Replace Dialog, the corresponding DyaRes DLL must be present alongside (in the same directory as) the Development DLL.

The Run-Time DLL (the re-distributable run-time version of the Dyalog DLL) contains no debugging facilities.

For the names of these files corresponding to the version of Dyalog that you are using, see *Installation/Configuration: Files And Directories*.

12.2 Isolation Mode

For *each* application which uses a class written in Dyalog APL, at least one copy of the development or run-time version of the Dyalog DLL will be started in order to host and execute the appropriate APL code. Each of these *engines* will have an APL workspace associated with it, and this workspace will contain classes and instances of these classes. The number of engines (and associated workspaces) which are started will depend on the Isolation Mode which was selected when the APL assemblies used by the application were generated. Isolation modes are:

- Each host process has a single workspace
- Each appdomain has its own workspace
- Each assembly has its own workspace

Note that, in this context, Microsoft Internet Information Services (IIS) is a *single process*, even though it may be hosting a large number of different web pages. Each ASP.NET application will be running in a separate *AppDomain*, a mechanism used by .NET to provide isolation within an application. Other .NET applications may also be divided into different AppDomains.

In other words, if you use the first option, ALL classes and instances used by any IIS web page will be hosted in the same workspace and share a single copy of the interpreter. The second option will start a new Dyalog engine for each ASP.NET application; the final option an engine for each assembly containing APL classes.

12.3 Workspace Size

By default, there is no limit placed upon the size of the workspace used by the Dyalog DLL and it will grow (and shrink) according to user demand.

The maximum workspace size may be specified by the **maxws** parameter that is used to control the workspace size in the development and run-time versions of the Dyalog program. The difference is that the **maxws** parameter must be specified for the **host application**, the application in which the Dyalog DLL is embedded.

This is achieved by defining a Registry key named:

```
HKLM\Software\Dyalog\Embedded\<appname>
```

or on 64-bit Windows:

```
HKLM\Software\Wow6432Node\Dyalog\Embedded\<appname>
```

where `<appname>` is the name of the application, containing a String Value named `maxws` set to the desired size.

The name of the ASP.NET application is aspnet_wp.exe or w3wp.exe ((IIS 6 and above).

An additional way is to set the **maxws** parameter on the command line of the Assembly at export time. That might be be useful if you know that you are only using one Dyalog assembly or the IsolationMode is "Each Assembly". For more information, see *Section 12.2*.

12.4 Structure of the Active Workspace

Each engine which is started has a workspace associated with it that contains all the APL objects it is currently hosting.

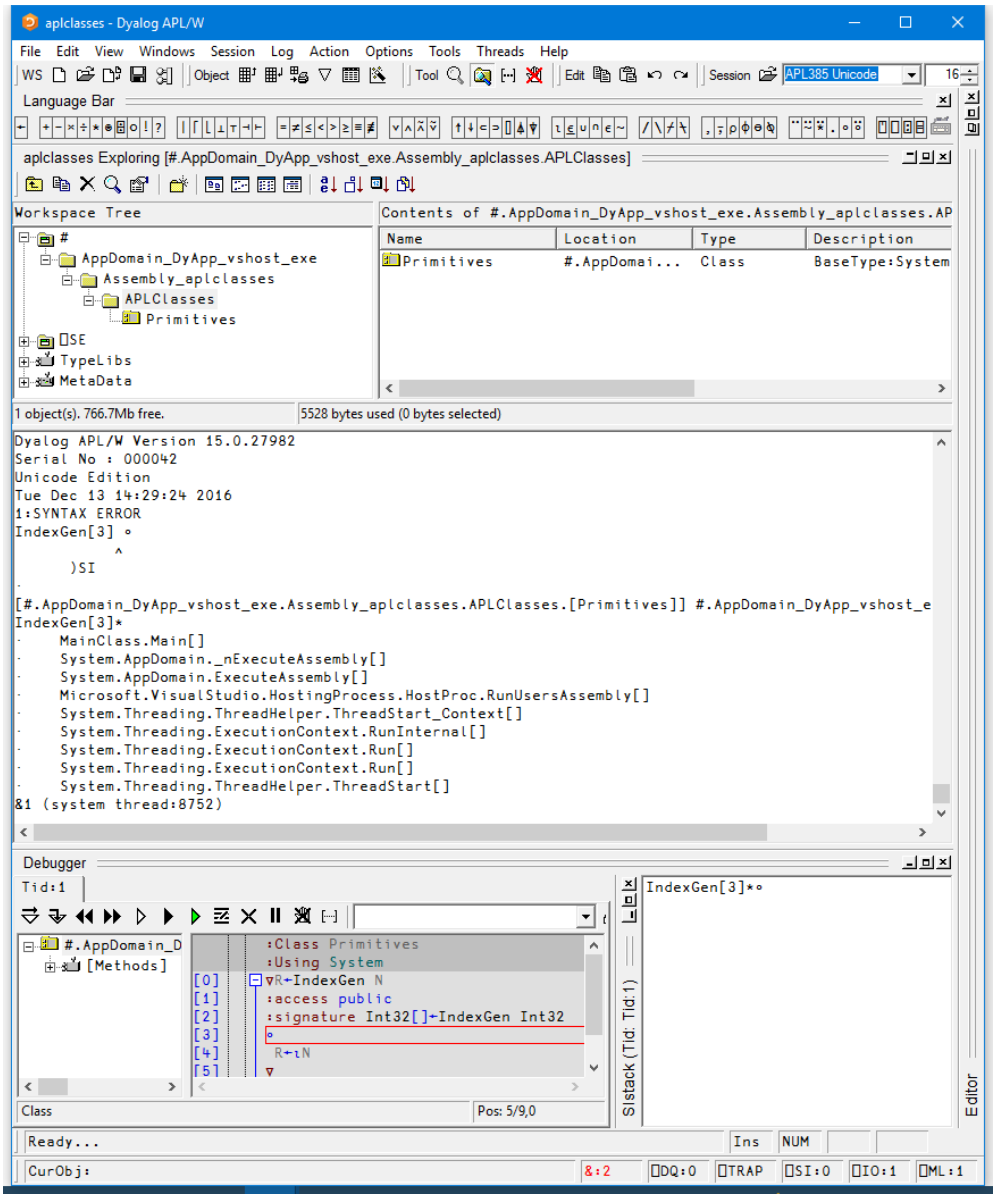
Unless the highest isolation mode, *Each assembly has its own workspace* has been selected, the workspace will contain one or more namespaces associated with .NET *AppDomains*. When .NET calls Dyalog APL to process an APL class, it specifies the AppDomain in which it is to be executed. To maintain AppDomain isolation and scope, Dyalog APL associates each different AppDomain with a namespace whose name is that of the AppDomain, prefixed by `AppDomain_`.

Within each `AppDomain_` namespace, there will be one or more namespaces associated with the different Assemblies from which the APL classes have been loaded. These namespaces are named by the Assembly name prefixed by `Assembly_`. If the APL class is a Web Page or a Web Service, the corresponding Assembly is created dynamically when the page is first loaded. In this case, the name of the Assembly itself is manufactured by .NET. Below the `Assembly_` namespace is a namespace that corresponds to the .NET Namespace that represents the container of your class. If the APL class is a Web Page or Web Service, this namespace is called `ASP`. Finally, the namespace tree ends with a namespace that represents the APL class. This will have the same name as the class. In the case of a Web Page or Web Service, this is the name of the `.aspx` or `.asmx` file.

Note that in the manufactured namespace names, characters that would be invalid symbols in a namespace name are replaced by underscores.

The following picture shows the namespace tree that exists in the Dyalog DLL workspace when the first example (see *Section 6.4*) in the chapter Writing .Net Classes is executed under Visual Studio. However, to cause the suspension, an error has been introduced in the method `IndexGen`.

In this case, there is a single AppDomain involved whose name, `DyApp_vshost_exe` is specified by .NET. APL has made a corresponding namespace called `AppDomain_DyApp_vshost_exe`. Next, there is a namespace associated with the Assembly `aplclasses`, named `Assembly_aplclasses`. Beneath this is a namespace called `APLClasses` associated with the .NET Namespace of the same name. Finally, there is the APL Class called `Primitives` .



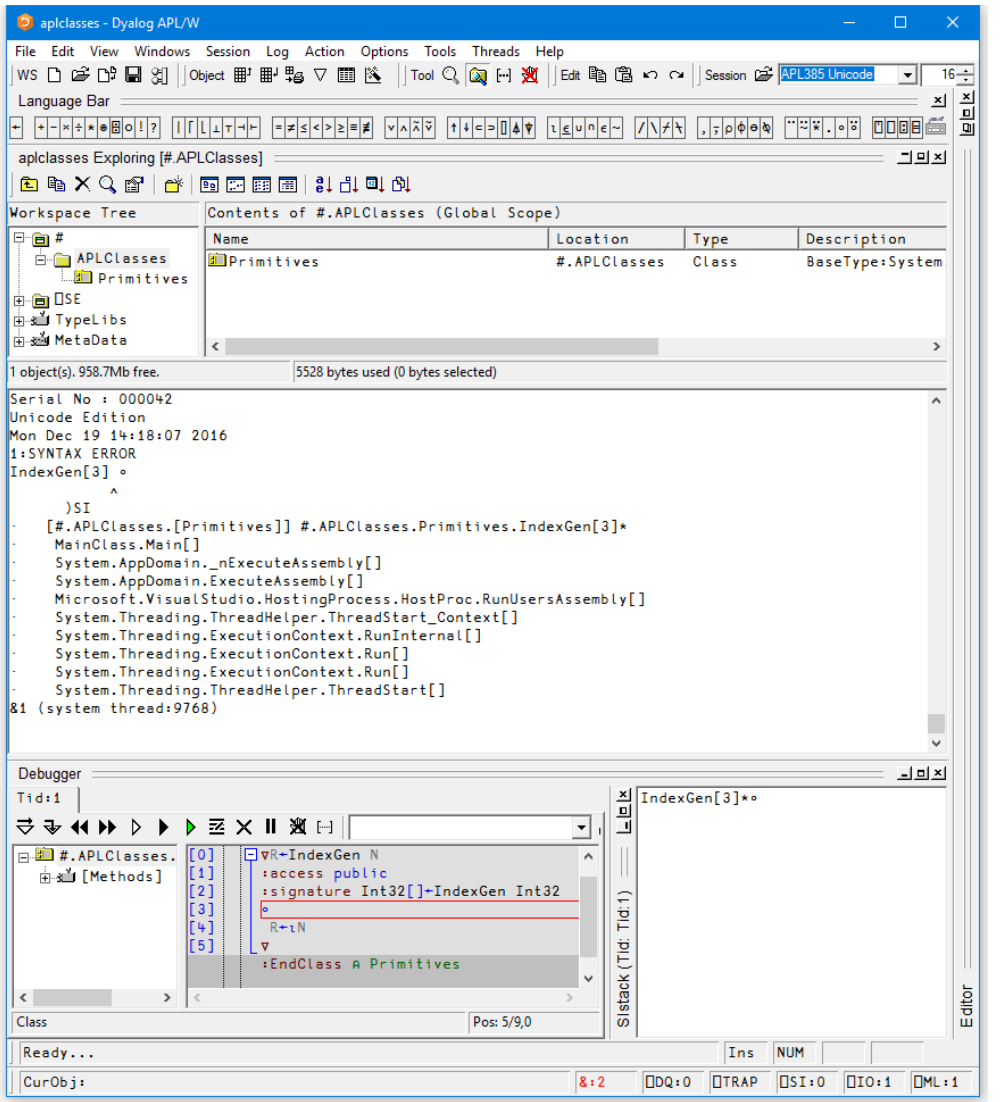
Notice that the state indicator displays the entire .NET calling structure, and not just the APL stack. In this case, the state indicator shows that `IndexGen` was called from `MainClass.Main`, which combines the class and method names specified in `aplfns.cs`. Note that .NET calls are slightly indented.

Notice too that `IndexGen` has been started on APL thread 1 which, in this case, is associated with system thread 8752. If the client application were to call `IndexGen` on

multiple system threads, this would be reflected by multiple APL threads in the workspace. This topic is discussed in further detail below.

The possibility for the client to execute code in several instances of an object at the same time requires that each executing instance is separated from all the others. Each instance will be created as an **unnamed** object in the workspace, within the relevant appdomain and assembly namespaces.

The picture below shows the workspace structure when the assembly was generated with isolation mode set to *Each assembly has its own workspace*. In this case, the AppDomain and Assembly structure is not created above the classes in the workspace, so the workspace structure is somewhat simpler:



12.5 Threading

The .NET Framework is inherently a multi-threaded environment. For example, ASP.NET runs its own thread pool from which it allocates system threads to its clients. Calls from ASP.NET into APL Web Pages and Web Services will typically be made from different system threads. This means that APL will receive calls from .NET while it is processing a previous call. The situation is further complicated when you write an APL Web Page that calls an APL Web Service, both of which may be hosted by a single Dyalog DLL inside ASP.NET. In these circumstances, ASP.NET may well allocate different system

threads to the .NET calls, which are made into the two separate APL objects. Although in the first example (multiple clients) APL could theoretically impose its own queuing mechanism for incoming calls, it cannot do so in the second case without causing a deadlock situation.

It is important to remember that whether running as DYALOG.EXE, or as the Dyalog DLL, the Dyalog interpreter executes in a *single system thread*. However, APL does provide the ability to run several APL threads at the same time. If you are unfamiliar with APL threads, see *Language Reference*, *Chapter 1* for an introduction to this topic.

To resolve this situation, Dyalog automatically allocates APL threads to .NET system threads and maintains a thread synchronisation table so that calls on the same system thread are routed to the same APL thread, and vice versa. This is important because a GUI object (cf. `System.Winforms`) is owned by the system thread that created it and can only be accessed by that thread.

The way that system threads are allocated to APL threads differs between the case where APL is running as the primary executable (DYALOG.EXE) or as a DLL hosted by another program. The latter is actually the simpler of the two and will be considered first.

DYALOG DLL Threading

In this case, all calls into the Dyalog DLL are initiated by Microsoft .NET.

When a .NET system thread first needs to run an APL function, APL starts a new APL thread for it, and executes the function in that APL thread. For example, if the first call is a request to create a new instance of an APL .NET object, its constructor function will be run in APL thread 1. An entry is made in the internal thread table that associates the originating system thread with APL thread 1. When the constructor function terminates, the APL thread is retained so that it is available for a subsequent call on its associated system thread. In this respect, the automatically created APL thread differs from an APL thread that was created using the spawn operator `&` (See Language Reference).

When a subsequent call comes in, APL locates the originating system thread in its internal thread table, and runs the appropriate APL function in the corresponding APL thread. Once again, when the function terminates, the APL thread is retained for future use. If a call comes in on a new system thread, a new APL thread is created.

Notice that under normal circumstances, APL thread 0 is never used in the Dyalog DLL. It is only ever used if, during debugging, the APL programmer explicitly changes to thread 0 by executing `)TID 0` and then runs an expression.

Periodically, APL checks the existence of all of the system threads in the internal thread table, and removes those entries that are no longer running. This prevents the situation arising that all APL threads are in use.

DYALOG.EXE Threading

In these cases, all calls to Microsoft .NET are initiated by Dyalog. However, these calls may well result in calls being made back from .NET into APL.

When you make a .NET call from APL thread 0, the .NET call is run on **the same system thread** that is running APL itself.

When you make a .NET call from any other APL thread, the .NET call is run on a different system thread. Once again, the correspondence between the APL thread number and the associated system thread is maintained (for the duration of the APL thread) so that there are no thread/GUI ownership problems. Furthermore, APL callbacks invoked by .NET calls back into APL will automatically be routed to the appropriate APL thread. Notice that, unlike a call to a DLL via ⎕NA, there is no way to control whether or not the system uses a different system thread for a .NET call. It will always do so if called from an APL thread other than APL thread 0.

Thread Switching

Dyalog will potentially *thread switch*, that is, switch execution from one APL thread to another, at the start of any line of APL code. In addition, Dyalog will potentially thread switch when a .NET method is called or when a .NET property is referenced or assigned a value. If the .NET call accesses a relatively slow device, such as a disk or the internet, this feature can improve overall throughput by allowing other APL code while a .NET call is waiting. On a multi-processor computer, APL may truly execute in parallel with the .NET code.

Note that when running DYALOG.EXE, .NET calls made from APL thread 0 will prevent any switching between APL threads. This is because the .NET code is being executed in the same system thread as APL itself. If you want to use APL multi-threading in conjunction with .NET calls, it is therefore advisable to perform all of the .NET calls from threads other than APL thread 0.

12.6 Debugging an APL.NET Class

All DYALOG.NET objects are executed by the Dyalog DLL. The full development version of the Dyalog DLL contains all of the development and debug facilities of the APL Session, including the Editors and Tracer. The run-time version contains no debugging

facilities at all. The choice of which version of the Dyalog DLL is used is made when the assembly is exported from APL using the *File|Export* menu, or compiled using `dyalogc.exe`.

If an APL .NET object that is bound to the full development version generates an untrapped APL error (such as a `VALUE ERROR`) **and** the client application is configured so that it is allowed to interact with the desktop, the APL code will suspend and the APL Session window will be displayed. Otherwise, it will throw an exception.

If an APL .NET object that is bound to the run-time version of the Dyalog DLL generates an untrapped APL error it will throw an exception.

Specifying the DLL

There are a number of different ways that you choose to which of the two versions of the Dyalog DLL your DYALOG.NET class will be bound. Note that the appropriate DLL must be available when the class is subsequently invoked. If the DLL to which the APL .NET class is bound is not present, it will throw an exception.

If you build a .NET class from a workspace using the *File/Export* menu item, you use the *Runtime application* checkbox. If *Runtime application* is unchecked, the .NET Class will be bound to the full development version. If *Runtime application* is checked, the .NET Class will be bound to the run-time version.

If you build a .NET class using the APLScript compiler, it will by default be bound to the full development version. If you specify the `/runtime` flag, it will be bound to the run-time version.

If your APL .NET class is a Web Page or a Web Service, you specify to which of the two DLLs it will be bound using the *Debug* attribute. This is specified in the opening declaration statement in the `.aspx`, `.asax` or `.asmx` file. If the statement specifies `"Debug=true"`, the Web Page or Web Service will be bound to the full development version. If it specifies `"Debug=false"`, the Web Page or Web Service will be bound to the run-time version.

If you omit the `Debug=` attribute in your Web page, the value will be determined from the various .NET `config` files on your computer.

Forcing a suspension

If an APL error occurs in an APL .NET object, a suspension will occur and the Session will be available for debugging. But what if you want to force this to happen so that you can Trace your code and see what is happening?

If your APL class is built directly from a workspace, you can force a suspension by setting stops in your code before using *Export* to build the DLL. If your class is a Web Page or Web Service where the code is contained in a workspace using the *workspace behind* technique (See Chapter 8), you can set stops in this workspace before you `)SAVE` it.

If your APL class is defined entirely in a Web Page, Web Service, or an APLScript file, the only way to set a break point is to insert a line that sets a stop explicitly using `⎕STOP`. It is essential that this line appears after the definition of the function in the script. For example, to set a stop in the `Intro\intro1.aspx` example discussed in Chapter 8, the script section could be as follows:

```
<script language="dyalog" runat="server">

∇Rotate args
:Access Public
:Signature Reverse Object,EventArgs

(⊃args).Text←⍕Pressme.Text
∇

3 ⎕STOP 'Rotate'

</script>
```

As an alternative, you can always insert a deliberate error into your code!

Finally, you can usually force a suspension by generating a Weak Interrupt. This is done from the pop-up menu on the APL icon in the System Tray that is associated with the full development version of the Dyalog DLL. Note that selecting Weak Interrupt from this menu will not have an immediate effect, but it sets a flag that will cause Dyalog APL to suspend when it next executes a line of APL code. You will need to activate your object in some way, for example, by calling a method, for this to occur. Note that this technique may not work if the Dyalog DLL is busy because a thread switch automatically resets the Weak Interrupt flag. In these circumstances, try again.

The run-time version of the Dyalog DLL does not display an icon in the System Tray.

Using the Session, Editor and Tracer

When an DYALOG.NET object suspends execution, all other active APL .NET objects bound to the full development version of the Dyalog DLL that are currently being executed by the same client application will also suspend. Furthermore, all the classes

currently being hosted by the Dyalog DLL are visible to the APL developer whether active (an instance is currently being executed) or not. Note that if a client application, such as ASP.NET, is also hosting APL .NET objects bound to the *runtime* version of the Dyalog DLL, these objects will be hosted in a separate workspace attached to the run-time version of the Dyalog DLL and will not be visible to the developer.

Debugging a running DYALOG.NET object is substantially the same process as debugging a stand-alone multi-threaded APL application. However, there are some important things to remember.

Firstly, the namespace structure above your APL class should be treated as being inviolate. There is nothing to prevent you from deleting namespaces, renaming namespaces, or creating new ones in the workspace. However, you do so at your peril!

Similarly, you should not alter, delete or rename any functions that have been automatically generated on your behalf by the APLScript compiler. These functions are also inviolate.

If execution in the Dyalog DLL is suspended, you may not execute `)CLEAR` or `)RESET`. You may execute `)OFF` or `⎕OFF`, but if you do so, the client application will terminate. If you attempt to close the APL Session window, you will be warned that this will terminate the client application and you may cancel the operation or continue (and exit).

If you fix a problem in a suspended function and then press *Resume* or *Continue* (Tracer) or execute a branch, and the execution of the currently invoked method succeeds, you will be left with an empty state indicator (assuming that no other threads are actively involved). The Dyalog DLL is at this stage idle, waiting for the next client request and the state indicator will be empty.

If, at this point, you close the APL Session window, a dialog box will give you the option of terminating the (client) application, or simply hiding the APL Session Window. If you execute `)OFF` or `⎕OFF` the client application will terminate.

Note that in the discussion above, a reference to terminating the client application means that APL executes `Application.Exit()`. This may cause the application to terminate cleanly (as with ASP.NET) or it may cause it to crash.

12.7 The web.config file

ASP.NET configuration parameters are defined in a file named `web.config` located in or above the root directory of an ASP.NET application. Parameters defined in these files supplement or override ASP.NET parameters which are defined system-wide.

The `web.config` file provided with Dyalog is located in the Dyalog sub-directory `samples\asp.net` and applies to all the examples residing in child directories of this directory. If you create a Dyalog ASP.NET application elsewhere on your system, you will need to copy this `web.config` into the application root directory. The parameters defined in the Dyalog `web.config` file are described below. Further details are provided in comments in the file.

DyalogBinDirectory

This specifies the full path to the Dyalog binaries (DLLs and script compiler).

dyalog (compiler)

This section defines an ASP.NET language named `dyalog` so that the expression `Language = "dyalog"` in a script file associates that script with the Dyalog APLScript compiler dyalogc.exe. Subsidiary parameters and keys for the dyalog compiler are:

debug	"true" (default) or "false" to bind the script to the Development DLL or the Run-time DLL
DyalogCompilerEncoding	"classic" or "unicode"..
DyalogCompilerOptions	This is used to define options for the script compiler. For example, to set `]WX` to 1 use "/wx:1".
DyalogCompilerEmitPragmas	Must be "true" if you are using workspace behind.

DyalogIsolationMode

This parameter specifies the isolation method. See *Section 12.2* for further details.

DyalogCacheDirectory may be used to define the directory used to save the cached files.