dyalog APL/W

Dyalog *APL*™ for Windows

.NET Interface Guide

Dyadic Systems Ltd

Riverside View
Basing Road, Old Basing
Basingstoke
Hampshire, RG24 7AL
United Kingdom

tel: +44 (0)1256 811125
fax: +44 (0)1256 811130
email: support@dyadic.com
http://www.dyadic.com

Version 10.0

First Edition March 2003

Printed in the United Kingdom.

Contents

CHAPTER 1

Overview

Introduction

This manual describes the Dyalog APL 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 APL that can then be used from other .NET languages such as C# and VB.NET.
- The ability to write Web Services in Dyalog APL.
- The ability to write ASP.NET web pages in Dyalog APL

.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 APL 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 APL. 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 APL 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. Unlike the current Windows GUI, it is thoroughly object-oriented and based upon a single consistent programming model. Visual Studio .NET, which is used to develop GUI applications in Visual Basic and C#, produces code that uses `System.Windows.Forms`. Dyalog APL allows you to use `System.Windows.Forms`, instead of (and in some cases, in conjunction with) the built-in Dyalog APL GUI objects such as the Dyalog APL 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 APL directly as a scripting language (*APLScript*) to write ASP.NET web pages. In addition, you can call Dyalog APL 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 APL to build browser-based applications for the Internet, or for your corporate Intranet.

Prerequisites

The Dyalog APL .NET interface requires a computer running Windows 2000 or Windows XP Professional with the following elements installed:

- The Microsoft .NET Framework SDK V1.0.3705 or higher.
- Microsoft Internet Information Services (IIS) 5.0 or 5.1
- Microsoft Internet Explorer Version 6.00.

The dotnet Sub-directory

The `dotnet` sub-directory contains files that are used to support the .NET interface, namely:

- `aplc.exe;` the `APLScript` compiler that is itself written in Dyalog APL and packaged as an executable.
- `aplprovider.dll`; which performs the initial processing of an `APLScript` file.
- `bridge.dll`; the interface library to the .NET framework
- `dyalog10.dll`; the developer/debug version of the dynamic link library that hosts the execution of Dyalog APL classes and COM objects.
- `dyalog10rt.dll`; the re-distributable run-time version of `dyalog10.dll`.
- `dyadic.dll`; a subsidiary library

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 `apl.net`, and contains various sample APL Web applications.
- `winforms`; a sub-directory that contains sample applications that use the `System.Windows.Forms` GUI classses.

CHAPTER 2

Accessing .NET Classes

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. An object is simply an instance of a .NET Framework class. An object is created by calling the class's constructor function.

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

An assembly is a logical DLL that contains all of the code and metadata for a Class or a number of classes. Assemblies are intended to resolve the DLL Hell (version conflicts) inherent in previous versions of Windows. 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.

Locating .NET Classes and Assemblies

Unlike COM objects, which are referenced via the Windows Registry, .NET assemblies and the classes they contain, are totally self-contained independent entities. 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 APL 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 `System.Windows.Forms` is the .NET namespace represented by the `System.Windows.Forms.dll` assembly.

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 APL, 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 (e.g. c:\winnt\Microsoft.NET\Framework\v1.0.3705)

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`.

Note that `⎕USING` has Namespace scope, i.e. each Dyalog APL Namespace 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.

Using .NET Classes

To create a Dyalog APL object as an instance of a .NET class, you invoke the `New` method of the class.

This is a monadic method whose argument depends upon the particular class. Its result is a namespace reference to the newly created object. For example, to create a `DateTime` object whose value is the 30th April 2001:

```
      ⎕USING←'System'

      mydt←DateTime.New 2001 4 30
```

The result of the `New` method is a namespace reference with name class 9.

```
      ⎕NC'mydt'
9
```

If you type the name of 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
30/04/2001 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←System.DateTime.New 2001 4 30
```

Although, in the case of a `DateTime` class, it is appropriate to call the *`New`* method with a specific argument (a date) it is common for a class *not* to require parameters when creating a new instance.

Typically, to create a default object in other languages, you call the `New` method with no argument. This syntax is not possible in APL, so instead you specify an argument of `⍬` (zilde). For example to obtain a default Button object (a class exposed by `System.Windows.Forms`):

```
      mybtn←Button.New ⍬
```

Notice that when you create a new instance of a .NET class, you do not have to declare the class name in advance. Assuming that you have defined `⎕USING` correctly, you can simply use the .NET class name directly.

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 that must be used to create 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 APL using the *`New`* function.

How the *New* Function is implemented

The first time that Dyalog APL encounters a reference to a non-existent name (i.e. 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 special name class of 10 and is associated with the corresponding .NET namespace.

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

The resolution of the `New` method is also special. First, APL searches the class for a static method called New and, if it is found, calls it. If not, it calls the *constructor method* in the class and associates the symbol `New` (in the class) with the constructor.

If you call `New` with an argument of `⍬` (zilde), APL will attempt to call the version of the constructor that is defined to take no arguments. If no such version of the constructor exists, the call will fail with a `DOMAIN ERROR`. Otherwise, APL will call the version of the constructor whose parameters match the argument you have supplied. If no such version of the constructor exists, the call will fail with a `LIMIT ERROR`.

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 returns 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` appears to return the name of the object's Type.

Note that if you want to check the type of an object, this can be obtained using its `GetType` method.

Exploring .NET Classes

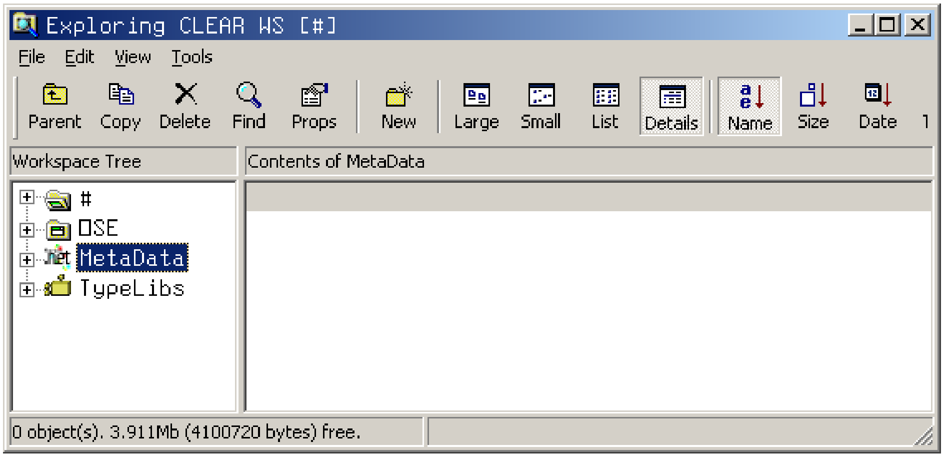
Microsoft supplies a tool for browsing .NET Class libraries called `ILDASM.EXE`.

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 enable the display of Metadata in the Workspace Explorer, you must have the *Type Libraries* option of the *View* menu checked.

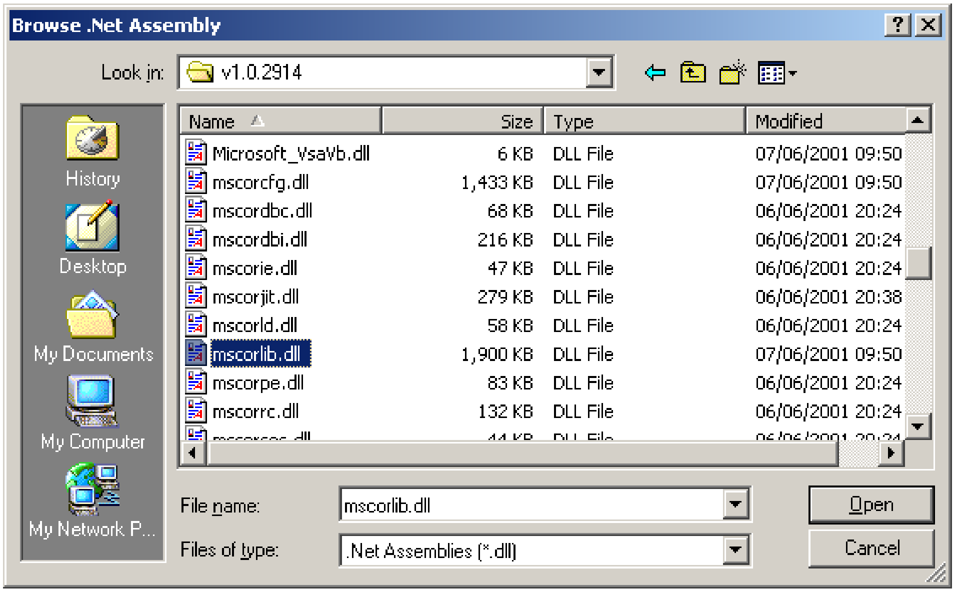
To gain information about one or more Net Classes, open the Workspace Explorer, right click the *Metadata* folder, and choose *Load*.



0 object(s). 3.911Mb (4100720 bytes) free.

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

Note that the .NET Classes provided with the .NET Framework are typically located in `C:\WINNT\Microsoft.NET\Framework\V1.0.3705`.



Look in: v1.0.2914

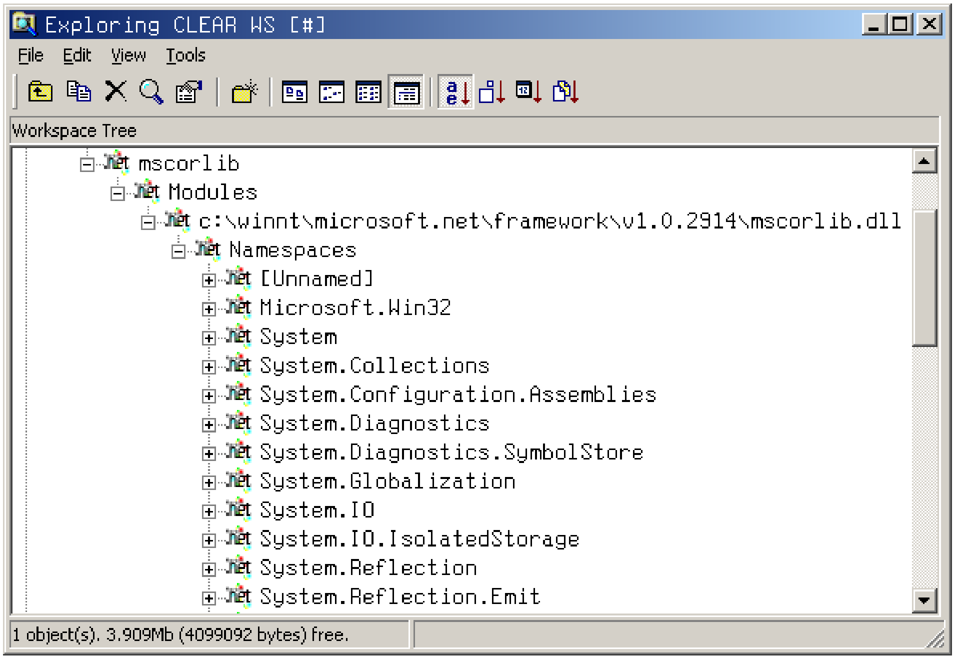
Name	Size	Type	Modified
Microsoft_VsaVb.dll	6 KB	DLL File	07/06/2001 09:50
mscorcfg.dll	1,433 KB	DLL File	07/06/2001 09:50
mscordbc.dll	68 KB	DLL File	06/06/2001 20:24
mscordbi.dll	216 KB	DLL File	06/06/2001 20:24
mscorie.dll	47 KB	DLL File	06/06/2001 20:24
mscorjit.dll	279 KB	DLL File	06/06/2001 20:38
mscorld.dll	58 KB	DLL File	06/06/2001 20:24
mscorlib.dll	1,900 KB	DLL File	07/06/2001 09:50
mscorpe.dll	83 KB	DLL File	06/06/2001 20:24
mscorrc.dll	132 KB	DLL File	06/06/2001 20:24

File name: mscorlib.dll

Files of type: .Net Assemblies (*.dll)

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.

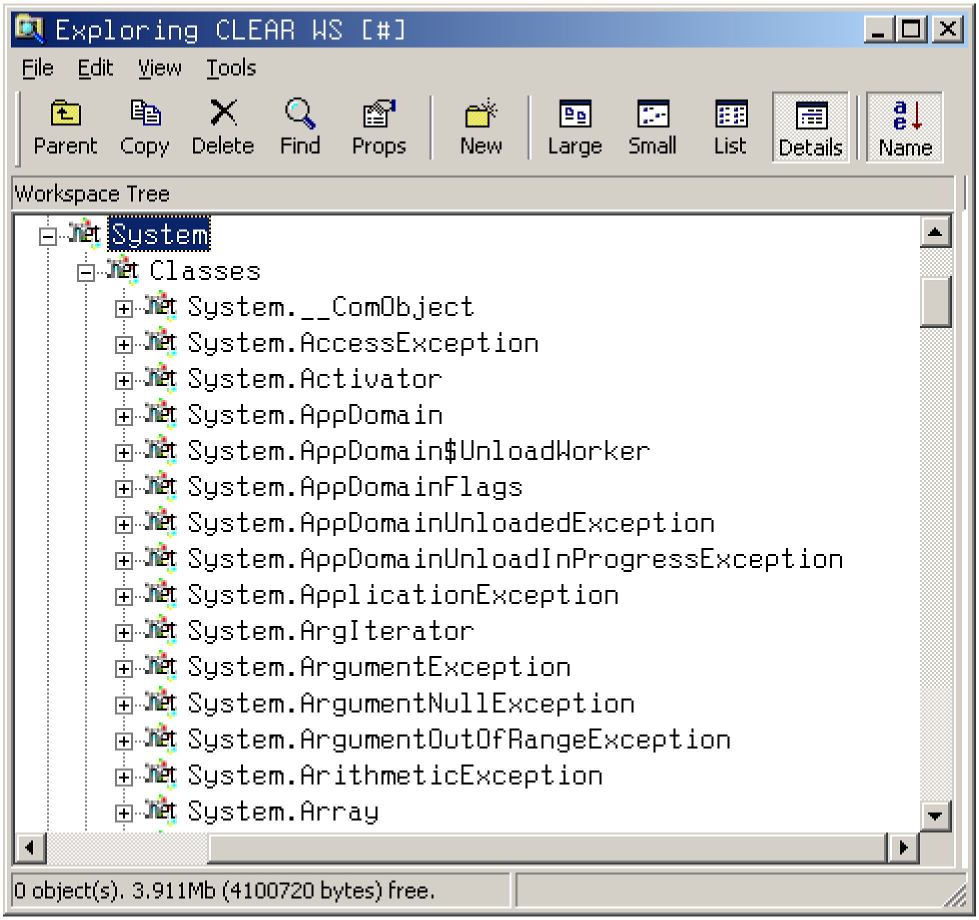
The result of opening this Assembly is illustrated in the following screen shot. The somewhat complex tree structure that is shown in the Workspace Explorer merely reflects the structure of the Metadata itself.



```
mscorlib
  Modules
    c:\winnt\microsoft.net\framework\v1.0.2914\mscorlib.dll
      Namespaces
        [Unnamed]
        Microsoft.Win32
        System
        System.Collections
        System.Configuration.Assemblies
        System.Diagnostics
        System.Diagnostics.SymbolStore
        System.Globalization
        System.IO
        System.IO.IsolatedStorage
        System.Reflection
        System.Reflection.Emit
```

1 object(s). 3.909Mb (4099092 bytes) free.

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

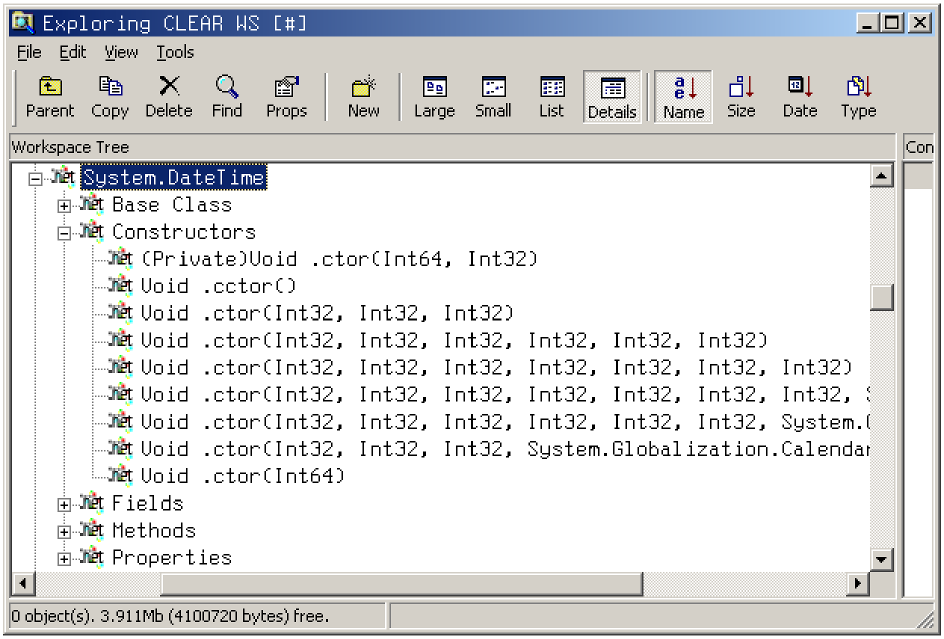


System
Classes
System.__ComObject
System.AccessException
System.Activator
System.AppDomain
System.AppDomain$UnloadWorker
System.AppDomainFlags
System.AppDomainUnloadedException
System.AppDomainUnloadInProgressException
System.ApplicationException
System.ArgIterator
System.ArgumentException
System.ArgumentNullException
System.ArgumentOutOfRangeException
System.ArithmeticException
System.Array

0 object(s). 3.911Mb (4100720 bytes) free.

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`.



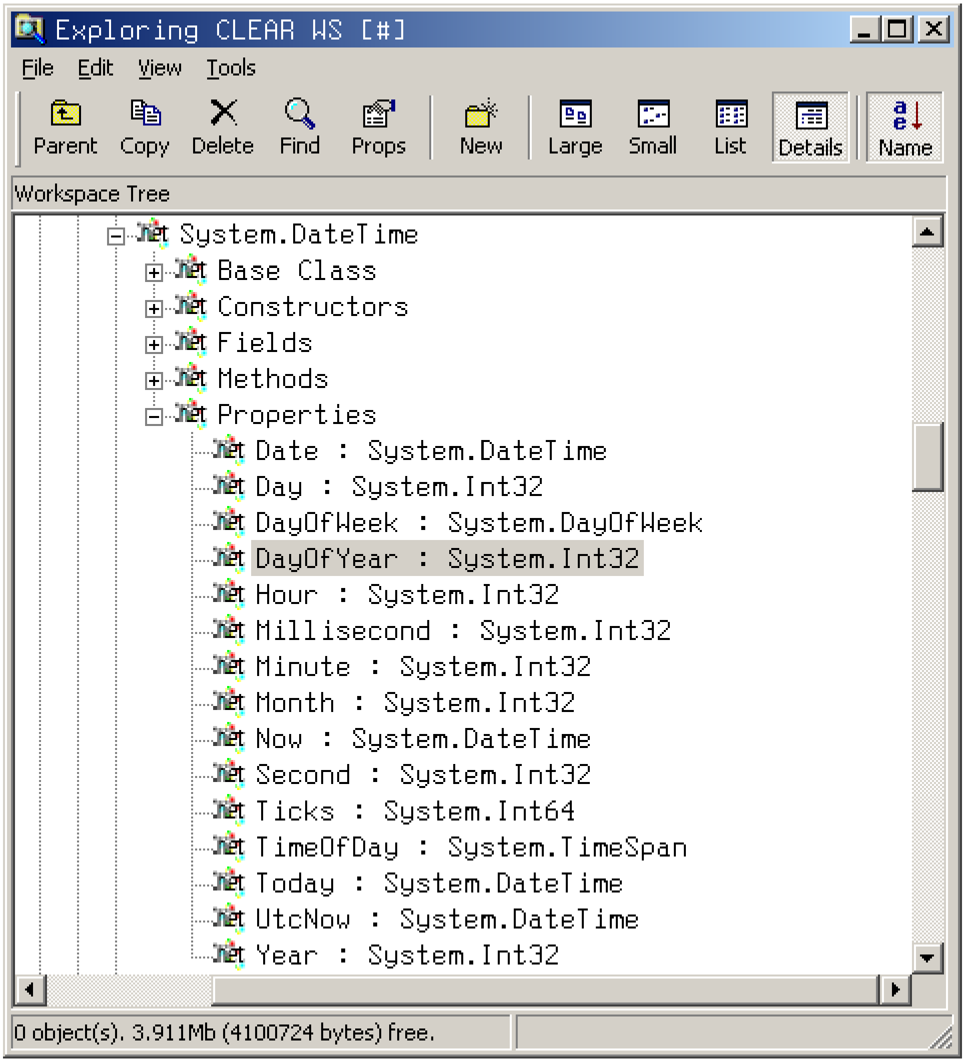
```
System.DateTime
  Base Class
  Constructors
    (Private)Void .ctor(Int64, Int32)
    Void .cctor()
    Void .ctor(Int32, Int32, Int32)
    Void .ctor(Int32, Int32, Int32, Int32, Int32, Int32)
    Void .ctor(Int32, Int32, Int32, Int32, Int32, Int32, Int32)
    Void .ctor(Int32, Int32, Int32, Int32, Int32, Int32, 
    Void .ctor(Int32, Int32, Int32, Int32, Int32, Int32, System.
    Void .ctor(Int32, Int32, Int32, System.Globalization.Calendar
    Void .ctor(Int64)
  Fields
  Methods
  Properties
```

0 object(s). 3.911Mb (4100720 bytes) free.

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←DateTime.New 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`.



```
System.DateTime
  Base Class
  Constructors
  Fields
  Methods
  Properties
    Date : System.DateTime
    Day : System.Int32
    DayOfWeek : System.DayOfWeek
    DayOfYear : System.Int32
    Hour : System.Int32
    Millisecond : System.Int32
    Minute : System.Int32
    Month : System.Int32
    Now : System.DateTime
    Second : System.Int32
    Ticks : System.Int64
    TimeOfDay : System.TimeSpan
    Today : System.DateTime
    UtcNow : System.DateTime
    Year : System.Int32
```

0 object(s). 3.911Mb (4100724 bytes) free.

You can query a property by direct reference:

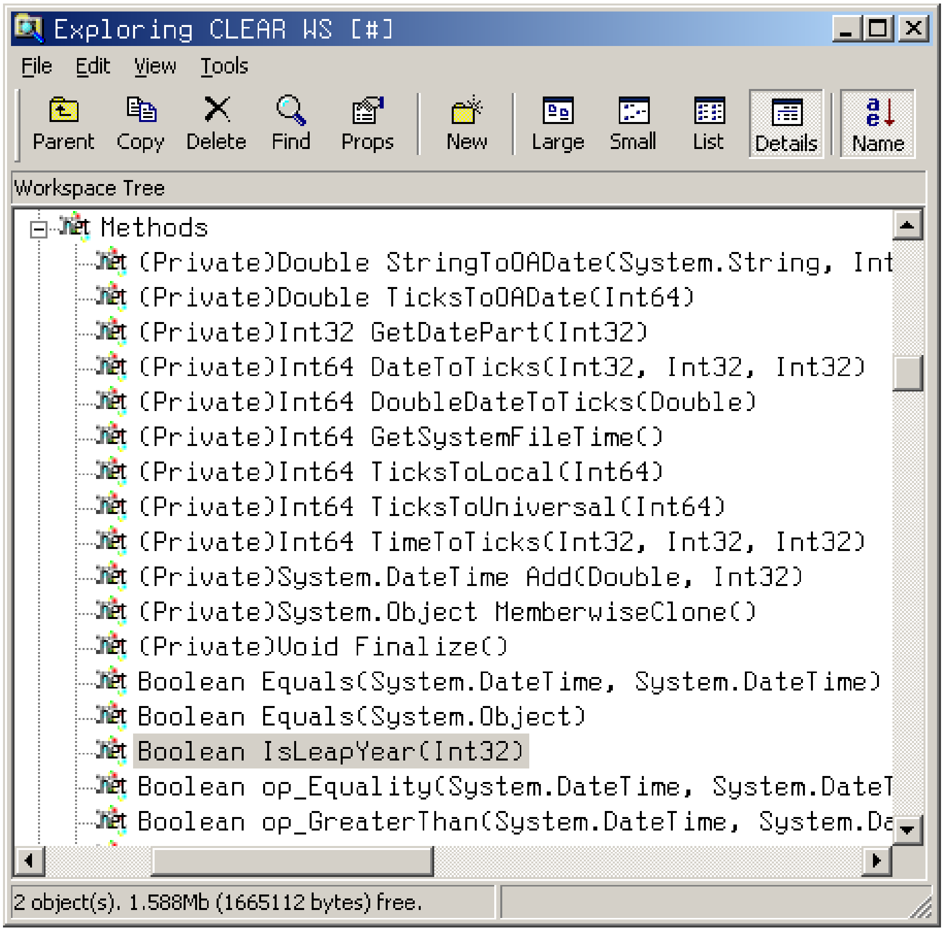
```
      mydt.DayOfWeek
1
```

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
```



```
Methods
  (Private)Double StringToOADate(System.String, Int
  (Private)Double TicksToOADate(Int64)
  (Private)Int32 GetDatePart(Int32)
  (Private)Int64 DateToTicks(Int32, Int32, Int32)
  (Private)Int64 DoubleDateToTicks(Double)
  (Private)Int64 GetSystemFileTime()
  (Private)Int64 TicksToLocal(Int64)
  (Private)Int64 TicksToUniversal(Int64)
  (Private)Int64 TimeToTicks(Int32, Int32, Int32)
  (Private)System.DateTime Add(Double, Int32)
  (Private)System.Object MemberwiseClone()
  (Private)Void Finalize()
  Boolean Equals(System.DateTime, System.DateTime)
  Boolean Equals(System.Object)
  Boolean IsLeapYear(Int32)
  Boolean op_Equality(System.DateTime, System.DateT
  Boolean op_GreaterThan(System.DateTime, System.Da
```

2 object(s). 1.588Mb (1665112 bytes) free.

Advanced Techniques

Class Methods

Certain .NET Classes provide methods, and properties, that can be called directly without the need to create an instance of the Class first.

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

```
      ⎕USING ←,⊂'System'

      DateTime.Now
07/11/2001 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←DateTime.New 2001 4 30
      DT2←DateTime.New 2001 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
07/11/2001 11:33:45

      TS←TimeSpan.New 1 1 1
      TS
01:01:01

      DateTime.op_Addition DT3 TS
07/11/2001 12:34:46
```

```
      (DateTime.op_Subtraction DT3 TS).ToString ⍬
07/11/2001 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.

.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
07/11/2001 12:34:46
      DT3-TS
07/11/2001 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
01/01/2001 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←DateTime.New 100000 0 0
EXCEPTION
      DT←DateTime.New 100000 0 0

      ⎕EN
90
      ⎕EXCEPTION.Message
Specified argument was out of the range of valid values.

Parameter name: Year, Month, and Day parameters describe
an unrepresentable 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)
```

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←DirectoryInfo.New 'C:\Dyalog10'
```

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

```
      d
C:\Dyalog10
```

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'
 dyalog.exe  dyalogrt.exe
```

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

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

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

```
      (d.GetFiles '*.exe').CreationTime
07/11/2002 14:51:35  22/11/2002 14:46:12
```

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

```
      files←d.GetFiles ⍬
```

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
 DeIsL1.isu                      07/11/2002 14:50:58
 def_us.dse                      07/11/2002 14:51:05
 def_uk.dse                      07/11/2002 14:51:06
 def_gr.dse                      07/11/2002 14:51:08
 def_fi.dse                      07/11/2002 14:51:08
 def_fr.dse                      07/11/2002 14:51:08
```

. . .

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←MailMessage.New ⍬
      m.From←'tony.blair@uk.gov'
      m.To←'sales@dyadic.com'
      m.Subject←'order'
      m.Body←'Send me 100 copies of Dyalog APL immediately'

      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@dyadic.com' 'o
rder' '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'
      ↑⎕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 `http://www.cdnow.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 'http://www.cdnow.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..

```
      PX←WebProxy.New '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 its 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←StreamReader.New 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
```

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 know 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←Form.New ⍬
      f1.BorderStyle←FormBorderStyle.FixedDialog
      FormBorderStyle.FixedDialog
FixedDialog
```

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.New 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.

Handling Pointers with Dyalog.ByRef

Certain .NET methods take parameters that are pointers.

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

An example of a .NET method that requires pointers is the `nGetVersion` method that is provided by the System.Reflection.Assembly class. This method may be used to obtain the major version, minor version, and build and revision numbers associated with a .NET Assembly. The following example illustrates how this works.

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 `(dotnet\bridge.dll)`, because (like `mscorlib.dll`) it is automatically loaded by DYALOG.EXE.

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

`System.AppDomain.CurrentDomain` represents the collection of Assemblies that is loaded into the current process. The first of these is `mscorlib.dll`.

```
      ass←AppDomain.CurrentDomain.GetAssemblies[1]
      ass.GetType ⍬
System.Reflection.Assembly
      ass
mscorlib, Version=1.0.3300.0, Culture=neutral, PublicKeyToken
=b77a5c561934e089
```

The version information is in fact shown when you display the object and could be derived programmatically by parsing the character vector returned by format (⍕). However, the `nGetVersion` method, which produces this information in numerical form, is more convenient. The problem is that it takes four parameters, each of which is a *pointer* to an `Int32`. When you call `nGetVersion`, it enters the major, minor, build and revision numbers into the addresses that you have provided as its parameters.

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 a empty pointer:

```
      ptr1←ByRef.New ⍬
```

Or, to create pointers to specific values,

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

Notice that the parameter can be any scalar APL array, so you must enclose it. 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 `nGetVersion` method takes as its parameters, 4 pointers into which it will insert `Int32` values. Using `ByRef`, we can call it as follows:

```
      args←ByRef.New¨4⍴⊂⍬
      args.Value
VALUE ERROR
      args.Value
      ^
      ass.nGetVersion args
1
      args.Value
1 0 3300 0
```

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←FileStream.New'c:\tmp\jd.txt'FileMode.Open
      fs.Length
25
      fs.Read(arg←ByRef.New⊂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
```

CHAPTER 3

Using Windows.Forms

Introduction

`System.Windows.Forms` is a .NET namespace that provides a set of classes for creating the Graphical User Interface for Windows applications. For languages such as C# and Visual Basic, this mechanism replaces the Windows API as the means to write the GUI. For Dyalog APL developers, `System.Windows.Forms` is an *alternative* to the Dyalog APL built-in GUI, which will continue to be maintained for the foreseeable future.

The main advantage of using `System.Windows.Forms` is that it provides immediate access to the latest Microsoft GUI components. Whenever Microsoft develops a new `Windows.Forms` component, it can immediately be incorporated into a Dyalog APL application; you do not need to wait for Dyadic to *cover* it. The same applies to GUI components developed by third parties.

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

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 APL, 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.

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 (i.e. without a parent) and then associated with a container, such as a `Form`, by calling the `Add` method. Notice too that `Windows.Forms` objects are associated with APL symbols that are namespace references, but `Windows.Forms` objects do not have implicit names.

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.

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`.

`EG[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` function is called with an argument of zilde(⍬).

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

`EG[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←Form.New ⍬
[7]  ⍝ Create two buttons to use as the accept and cancel
      buttons.
[8]    button1←Button.New ⍬
[9]    button2←Button.New ⍬
[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←Point.New 10 10
[15]   ⍝ Set the text of button2 to "Cancel".
[16]   button2.Text←'Cancel'
[17] ⍝ Set the position of the button based on the
      location of button1.
[18]   button2.Location←Point.New button1.Left
      button1.(Height+Top+10)
[19]
```

`EG[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, `EG[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 `StartPostion` 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 center 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`.

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

`EG[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
     ∇
```

Example 2

Functions `EG2` and `EG2A` illustrate how the Each operator (¨) and the extended namespace reference syntax in Dyalog APL 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←Form.New ⍬
[7]
[8]    textBox1←TextBox.New ⍬
[9]    label1←Label.New ⍬
[10]
[11]   ⍝ Initialize the controls and their bounds.
[12]   label1.Text←'First Name'
[13]   label1.Location←Point.New 48 48
[14]   label1.Size←Size.New 104 16
[15]   textBox1.Text←''
[16]   textBox1.Location←Point.New 48 64
[17]   textBox1.Size←Size.New 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 `(Form TextBox Label).New⊂⍬` runs the `New` function in 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←(Form TextBox Label).New⊂⍬
[8]
[9]    ⍝ Initialize the controls and their bounds.
[10]   (label1 textBox1).Text←'First Name' ''
[11]   (label1 textBox1).Location←Point.New¨(48 48)(48 64)
[12]   (label1 textBox1).Size←Size.New¨(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 ⍬
     ∇
```

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 SDK Beta2.

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\FrameworkSDK\Samples\...`
`...QuickStart\winforms\samples\VB\Data\Grid\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 is 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 is 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.

CHAPTER 4

Writing .NET Classes in Dyalog APL

Introduction

Dyalog APL allows you to build new .NET Classes, components and controls. 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 starts off 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.

Assemblies, Namespaces and Classes

To create a .NET class in Dyalog APL, you start with an APL workspace. This should contain a single top-level namespace whose name represents the name of the .NET Namespace that will be stored in your assembly. Under this namespace, you create one or more objects (namespaces) of Type `'NetType'` each of which represents a .NET class. When you create a *`NetType`* object, you specify the name of the base class from which it inherits (the default is `System.Object`).

Within each of the *`NetType`* namespaces, you define functions and variables as usual. If a function is to be exported as a method, you must define the number of parameters, their data types, and the data type of the result. You may also specify names for these items. This information can be entered through the *Net Properties* tab of the *Properties* dialog box, or established programmatically using *`SetMethodInfo`* and *`SetPropertyInfo`*. The information is necessary to allow a client application to use your classes.

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. Note that the workspace itself is not saved at the same time. Your .NET class is now ready for use.

When an APL .NET class is invoked by a client application, it automatically loads `dyalog10.dll` or `dyalog10rt.dll`, the developer/debug or run-time dynamic link library version of Dyalog APL. 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. See User Guide for further details.

Example 1

This example builds an Assembly called `APLClasses1.dll` in the sub-directory `samples\aplclasses`, which contains a .NET Namespace called `APLClasses`.

`APLClasses` contains a single .NET Class called `Primitives` that exports a single method called `IndexGen`.

First we create a container namespace `#.APLClasses` that will represent the .NET Namespace in the assembly:

```
clear ws
      )NS APLClasses
#.APLClasses
```

Next, using `⎕WC`, we create a NetType object called `#.APLClasses.Primitives`. Note that the default BaseClass for a `NetType` object is `System.Object`.

```
      )CS APLClasses
#.APLClasses
     'Primitives'⎕WC'NetType'
```

Then, inside the `Primitives` namespace, we set `⎕USING` so that the code inside this object can reference the .NET base types such as `Int32`.

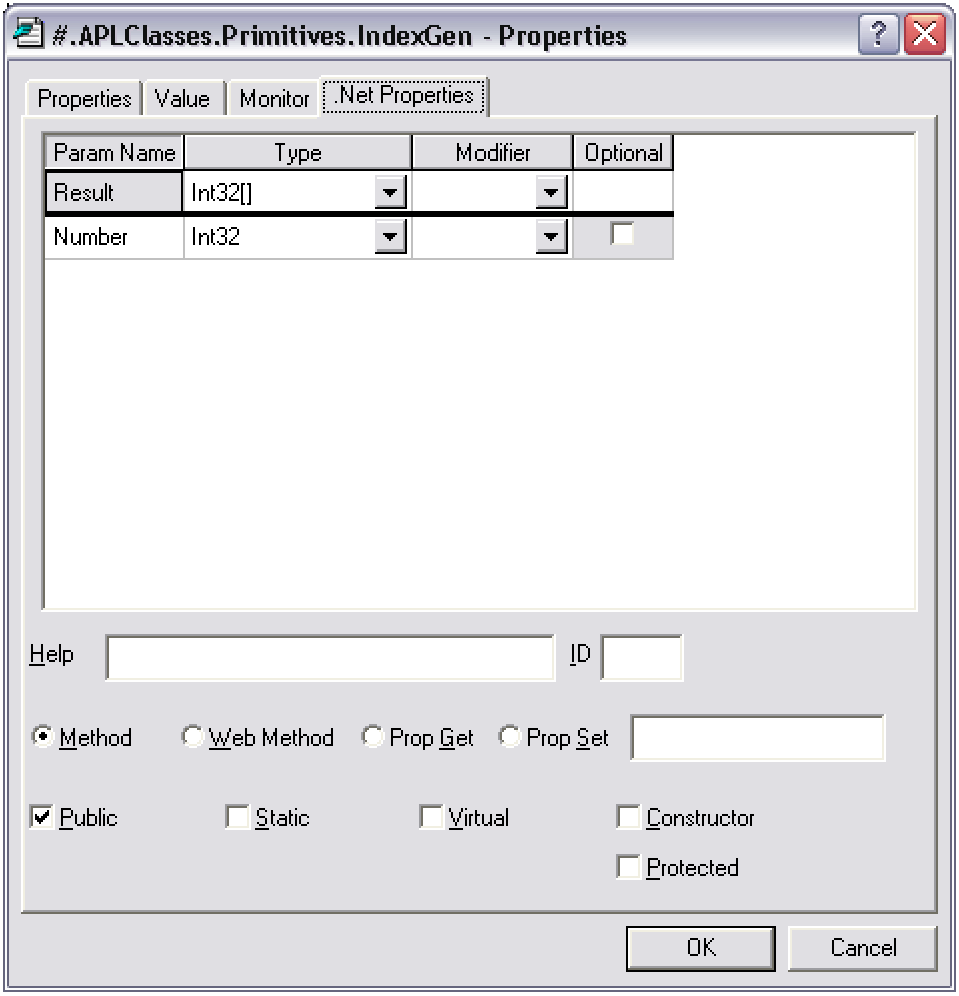
```
      )CS Primitives
#.APLClasses.Primitives
      ⎕USING←'System'
```

Next, we write the `Primitives.IndexGen` function. As we will see later, it is not necessary for a function to have the same name as the exported method that it implements, but it is the default.

```
     ∇ R←IndexGen N
[1]    R←⍳N
     ∇
```

The next step is to define the public characteristics for the exported method . This is done using the *.Net Properties* page of the *Properties* dialog box for the `IndexGen` function as shown below.

1. To make the function available to a client application, check the *Public* check box.
2. To export the function as a method (as opposed to a Web Method, or a Property Get/Set function), select the *Method* radio button.
3. Enter `Int32`[] in the box for *Result Type*. This says that IndexGen returns an array of integers.
4. Enter `Int32` in the box for the *Param1 Type* and (optionally) rename the parameter Param1, in this case, to "number".
5. Click *OK*.



Param Name	Type	Modifier	Optional
Result	Int32[]		
Number	Int32		☐

● Method ○ Web Method ○ Prop Get ○ Prop Set

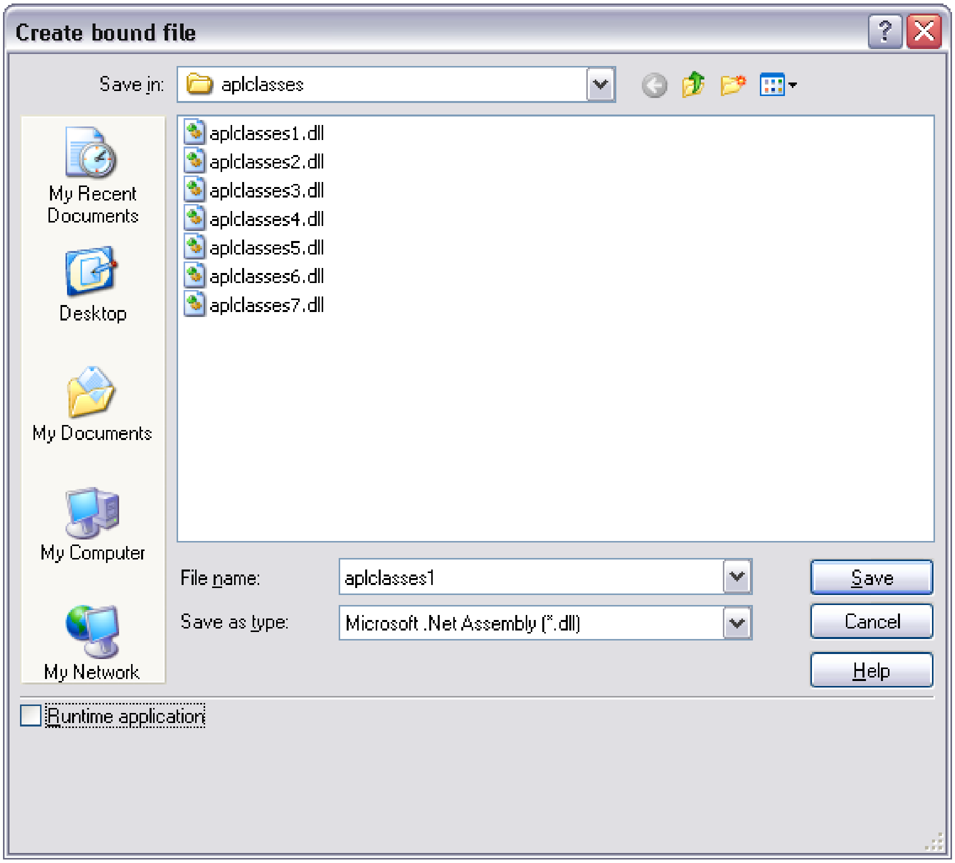
☑ Public ☐ Static ☐ Virtual ☐ Constructor ☐ Protected

Note that APL will at this stage check the data types you have specified for the result and for the method's parameters. If one or more of the data types are not recognised as available .NET Types (Classes), you will be informed by a message box. If you see such a warning you have either entered an incorrect data type, or you have not set `⎕USING` correctly. In this case, the only data type used is `Int32`, which is a Type, defined in the .NET Namespace `System`, and `⎕USING` is set to `'System'`, so all will be well.

The next step is not strictly necessary, but it does make good sense to `)SAVE` the workspace at this stage. The name you choose for the workspace will be the default name for the assembly

```
      )CS
#
      )WSID samples\APLClasses\aplclasses1
was CLEAR WS
      )SAVE
samples\APLclasses\aplclasses1 saved Wed Nov 21 12:30:38 2001
```

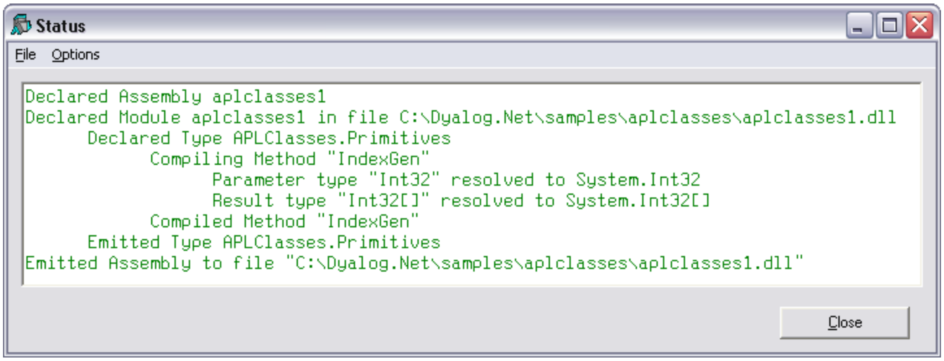
Now you are ready to create the assembly. This is done by selecting *Export…* from the Session *File* menu. This displays the following dialog box.



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 APL dynamic link library the assembly will be bound. See User Guide for further details.

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.



```
Declared Assembly aplclasses1
Declared Module aplclasses1 in file C:\Dyalog.Net\samples\aplclasses\aplclasses1.dll
      Declared Type APLClasses.Primitives
            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 "C:\Dyalog.Net\samples\aplclasses\aplclasses1.dll"
```

Note that when APL makes a .NET Assembly, it does **not** save the workspace at the same time.

aplfns1.cs

The following C# source, called `samples\APLClasses\aplfns1.cs`, may be used to call your APL .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]);
            }
      }
```

Then, to compile and run the program from a DOS command shell, change directory to the `samples\aplclasses` sub-directory, and then type the following commands shown in bold type. The first command is required to set up environment variables and your PATH. Note that all this assumes that you have Visual Studio.NET installed.

```
APLClasses>setpath.bat
Setting environment for using Microsoft Visual C++.NET
7.0 tools.
(If you also have Visual C++ 6.0 installed and wish to
use its tools
from the command line, run vcvars32.bat for Visual C++
6.0.)
APLClasses>csc /r:APLClasses1.dll aplfns1.cs
Microsoft (R)Visual C# Compiler Version 7.00.9254 [CLR
version 1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights
reserved.

APLClasses>aplfns1
1
2
3
4
5
6
7
8
9
10
```

Calling IndexGen from Dyalog APL

```
      ⎕USING←'APLClasses,samples\APLclasses\aplclasses1.dll'
      PR←Primitives.New ⍬
      PR.IndexGen 10
1 2 3 4 5 6 7 8 9 10
```

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`.

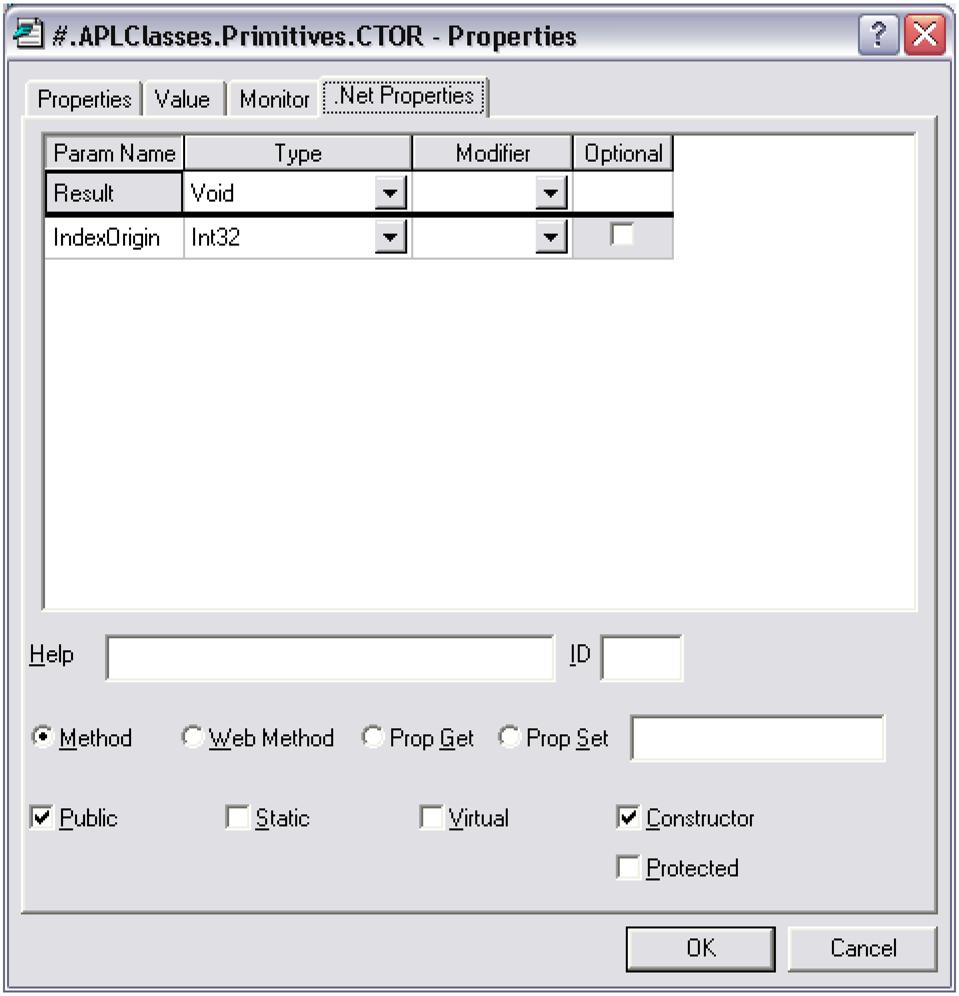
First, we will `)LOAD` the `aplclasses1` workspace we saved in Example 1, and change to the `APLClasses.Primitives` namespace.

```
      )LOAD samples\APLClasses\aplclasses1
samples\APLClasses\aplclasses1 saved Wed Nov 21 12:30:38 2001
      )CS APLClasses.Primitives
#.APLClasses.Primitives
```

Next, we will define a function called `CTOR` that simply sets `⎕IO` to the value of its argument. The name of this function is purely arbitrary.

```
      ∇ CTOR IO
[1]     ⎕IO←IO
      ∇
```

Then we will export this function as a *constructor*. This is done using the *.Net Properties* page of the *Properties* dialog box for the `CTOR` function as shown below. Note that in addition to checking *Public* and selecting *Method*, the *Constructor* box is also checked. The data type of the result is defined to be `Void` (no result) and that of its parameter to be `Int32`.



Param Name	Type	Modifier	Optional
Result	Void		
IndexOrigin	Int32		☐

● Method ○ Web Method ○ Prop Get ○ Prop Set

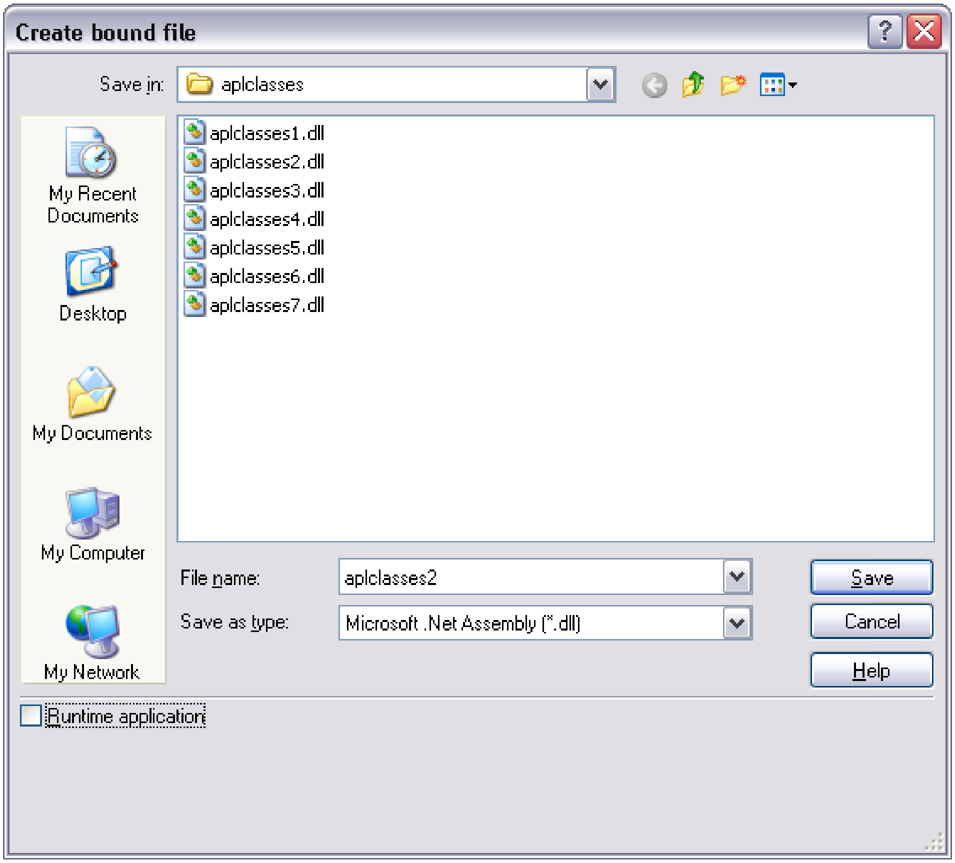
☑ Public ☐ Static ☐ Virtual ☑ Constructor

☐ Protected

Then we rename and save the workspace:

```
      )WSID  samples\APLClasses\aplclasses2
was samples\APLClasses\aplclasses1
      )SAVE
samples\APLClasses\aplclasses2 saved Wed Nov 21 12:39:10 2001
```

Finally, we can build a new .NET Assembly using *File/Export…* as before.



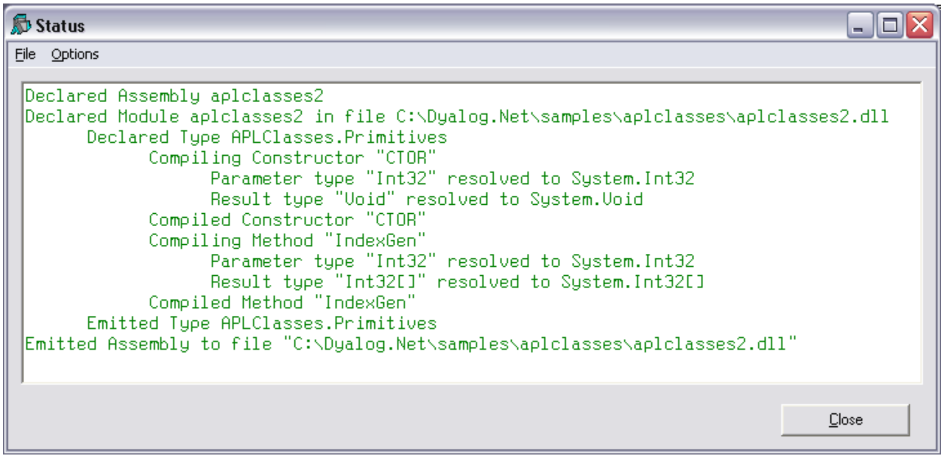
Save in: aplclasses

File name: aplclasses2

Save as type: Microsoft .Net Assembly (*.dll)

☐ Runtime application

Please note that, in this case, it is **essential** (for Example 2a) that the *Build runtime assembly* checkbox is not checked. We will need the development version for debugging purposes.



```
Declared Assembly aplclasses2
Declared Module aplclasses2 in file C:\Dyalog.Net\samples\aplclasses\aplclasses2.dll
      Declared Type APLClasses.Primitives
            Compiling Constructor "CTOR"
                  Parameter type "Int32" resolved to System.Int32
                  Result type "Void" 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 "C:\Dyalog.Net\samples\aplclasses\aplclasses2.dll"
```

aplfns2.cs

The following C# source, called `samples\APLClasses\aplfns2.cs`, may be used to call your APL .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);
```

When the code is compiled, this call is matched with the various constructors available in the `Primitives` class, namely the default constructor (which takes no arguments) and the *`CTOR`* constructor, which takes a single integer argument. The latter matches, so the program compiles successfully with this line compiled to call *`CTOR`* with a parameter of 0. When the program runs, the output is 0-9 as expected.

```
APLClasses>setpath.bat
...
APLClasses>csc /r:APLClasses2.dll aplfns2.cs
Microsoft (R)Visual C# Compiler Version 7.00.9254 [CLR
version 1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights
reserved.

APLClasses>aplfns2
0
1
2
3
4
5
6
7
8
9
```

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*.

aplfns2a.cs

The following C# source, called `samples\APLClasses\aplfns2a.cs`, may be used to demonstrate what happens.

```
using System;
using APLClasses;
public class MainClass
      {
      public static void Main()
            {
            Primitives apl = new Primitives(2);
            int[] rslt = apl.IndexGen(10);

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

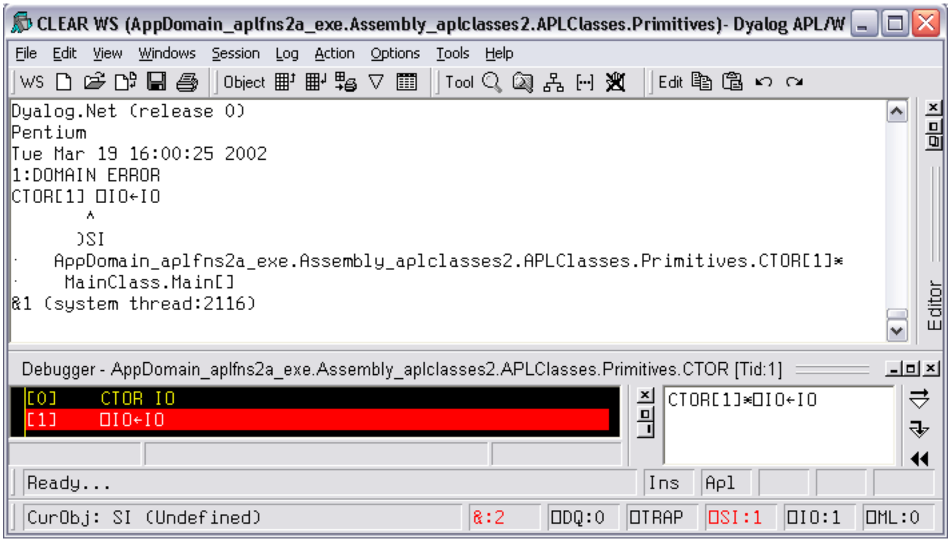
The code is the same as in the previous example, except that the line that creates an instance of the `Primitives` class specifies an inappropriate argument; in this case 2.

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

Then, when the program is compiled and run …

```
APLClasses>setpath.bat
...
APLClasses>csc /r:APLClasses2.dll aplfns2a.cs
Microsoft (R)Visual C# Compiler Version 7.00.9254 [CLR
version 1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights
reserved.

APLClasses>aplfns2a
```

… the APL Session appears, and the Tracer may 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 `APLClasses.Primitives`) to finish.


CLEAR WS (AppDomain_aplfns2a_exe.Assembly_aplclasses2.APLClasses.Primitives)- Dyalog APL/W
File Edit View Windows Session Log Action Options Tools Help
WS Object Tool Edit
Editor
Debugger - AppDomain_aplfns2a_exe.Assembly_aplclasses2.APLClasses.Primitives.CTOR [Tid:1]

```
Dyalog.Net (release 0)
Pentium
Tue Mar 19 16:00:25 2002
1:DOMAIN ERROR
CTOR[1] ⎕IO←IO
       ∧
      )SI
·   AppDomain_aplfns2a_exe.Assembly_aplclasses2.APLClasses.Primitives.CTOR[1]*
·    MainClass.Main[]
&1 (system thread:2116)
```

```
[0]     CTOR IO
[1]     ⎕IO←IO
```

```
CTOR[1]*⎕IO←IO
```

```
Ready...                     Ins  Apl
CurObj: SI (Undefined)   &:2  ⎕DQ:0  ⎕TRAP  ⎕SI:1  ⎕IO:1  ⎕ML:0
```

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

```
IO←1
→⎕LC
```

This causes the APL Session to disappear; the `aplfns2a` program continues successfully and the output is displayed.

```
1
2
3
4
5
6
7
8
9
10
```

Notice that in Dyalog APL, 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 2116. See Chapter 12 for further information on debugging APL classes.

Example 3

The correct .NET behaviour when an APL function fails with an error is to *throw an exception*, and his 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` has been extended to allow you 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`. (Other options for the left argument may be implemented later).

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.

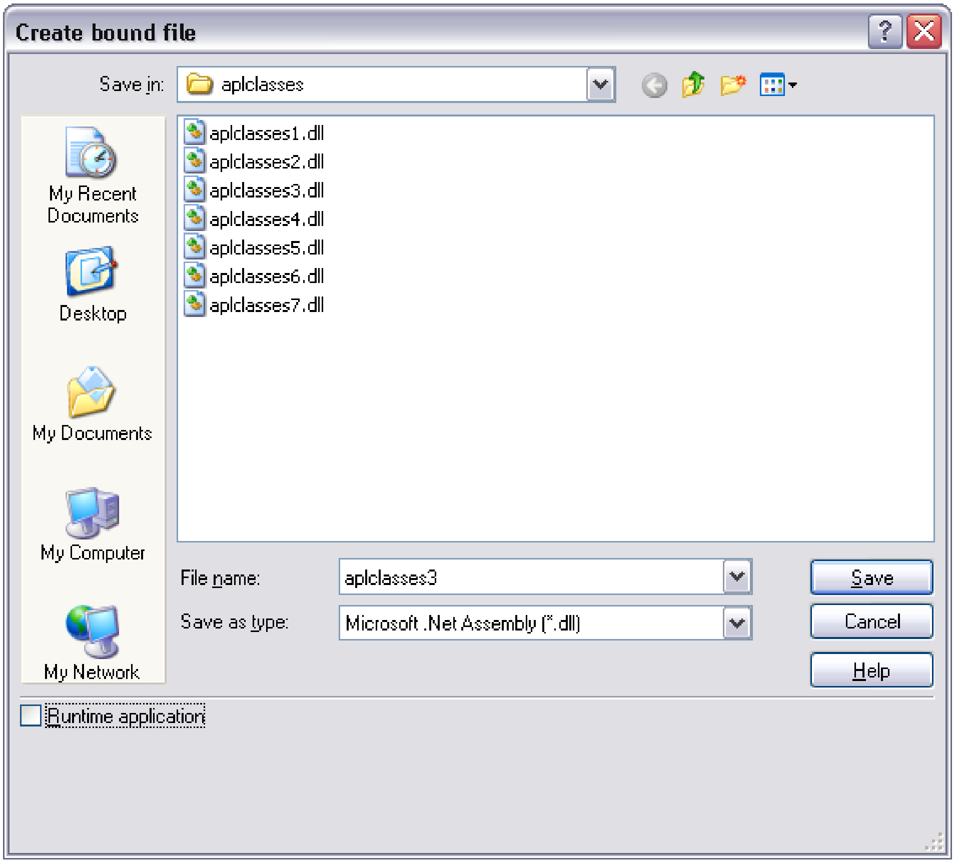
Starting with the APLCLASSES2.DWS workspace, the following changes add exception handling to the `CTOR` function.

```
      )LOAD samples\APLClasses\aplclasses2
samples\APLClasses\aplclasses2 saved Wed Nov 21 12:39:10 2001
      )CS APLClasses.Primitives
#.APLClasses.Primitives
```

Then modify the `CTOR` function to perform exception handling in the approved manner.

```
     ∇ CTOR IO;EX
[1]    :If IO∊0 1
[2]        ⎕IO←IO
[3]    :Else
[4]        EX←ArgumentException.New'IndexOrigin must be 0 or
1'
[5]        EX ⎕SIGNAL 90
[6]    :EndIf
     ∇

        )WSID  samples\APLClasses\aplclasses3
was samples\APLClasses\aplclasses2
      )SAVE
samples\APLClasses\aplclasses3 saved Wed Nov 21 13:32:14 2001
```

and make a new .NET Assembly called `aplclasses3.dll`.



Create bound file

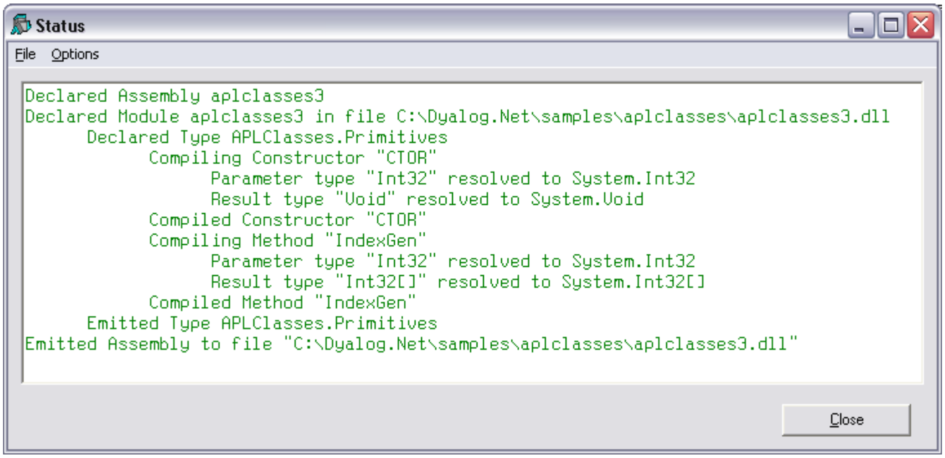
Save in: aplclasses

aplclasses1.dll
aplclasses2.dll
aplclasses3.dll
aplclasses4.dll
aplclasses5.dll
aplclasses6.dll
aplclasses7.dll

File name: aplclasses3

Save as type: Microsoft .Net Assembly (*.dll)

☐ Runtime application



```
Declared Assembly aplclasses3
Declared Module aplclasses3 in file C:\Dyalog.Net\samples\aplclasses\aplclasses3.dll
      Declared Type APLClasses.Primitives
            Compiling Constructor "CTOR"
                  Parameter type "Int32" resolved to System.Int32
                  Result type "Void" 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 "C:\Dyalog.Net\samples\aplclasses\aplclasses3.dll"
```

aplfns3.cs

The following C# source, called `samples\APLClasses\aplfns3.cs`, may be used to invoke the new *CTOR* function. `aplfns3.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);
            }
      }
```

Then, when the program is compiled and run …

```
APLClasses>setpath.bat
APLClasses>csc /r:APLClasses3.dll aplfns3.cs
Microsoft (R)Visual C# Compiler Version 7.00.9254 [CLR
version 1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights ...
APLClasses>aplfns3
IndexOrigin must be 0 or 1
```

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`.

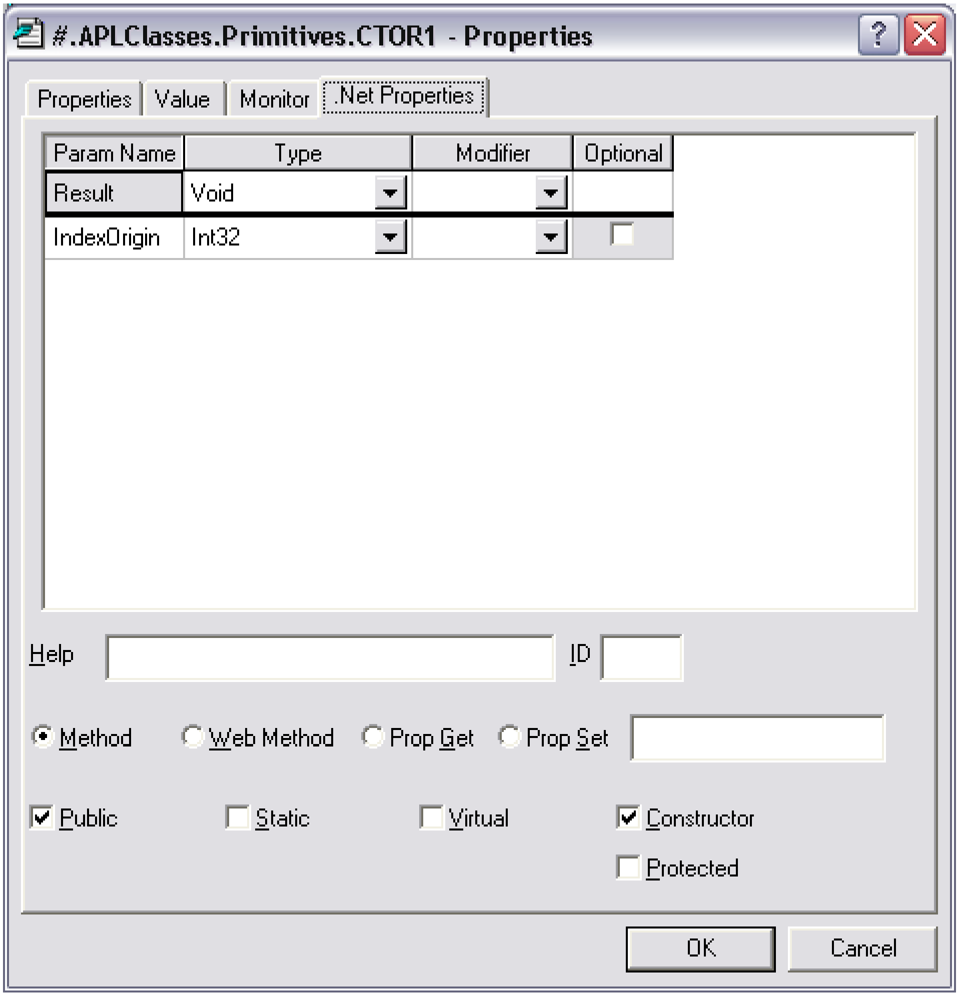
The simplest way to implement this is to have two public constructor functions `CTOR1` and `CTOR2`, which call a private constructor function `CTOR` as listed below.

```
      )LOAD  samples\APLClasses\aplclasses3
samples\APLClasses\aplclasses3 saved Wed Nov 21 13:32:14 2001
      )CS APLClasses.Primitives
#.APLClasses.Primitives

     ∇ CTOR1 IO
[1]    CTOR IO 0
     ∇

     ∇ CTOR2 IOML
[1]    CTOR IOML
     ∇

     ∇ CTOR IOML;IO;ML;EX
[1]    IO ML←IOML
[2]    :If ~IO∊0 1
[3]       EX←ArgumentException.New'IO must be 0 or 1'
[4]       EX ⎕SIGNAL 90
[5]    :EndIf
[6]    :If ~ML∊0 1 2 3
[7]       EX←ArgumentException.New'MigrationLevel must be
                                 0, 1, 2 or 3'
[8]       EX ⎕SIGNAL 90
[9]    :EndIf
[10]   ⎕IO ⎕ML←IO ML
     ∇
```

The *.Net Properties* 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 strictly a *Private Constructor*; it is simply an internal function that is invisible to the outside world.

.NET Properties for *CTOR1*



Param Name	Type	Modifier	Optional
Result	Void		
IndexOrigin	Int32		☐

Help

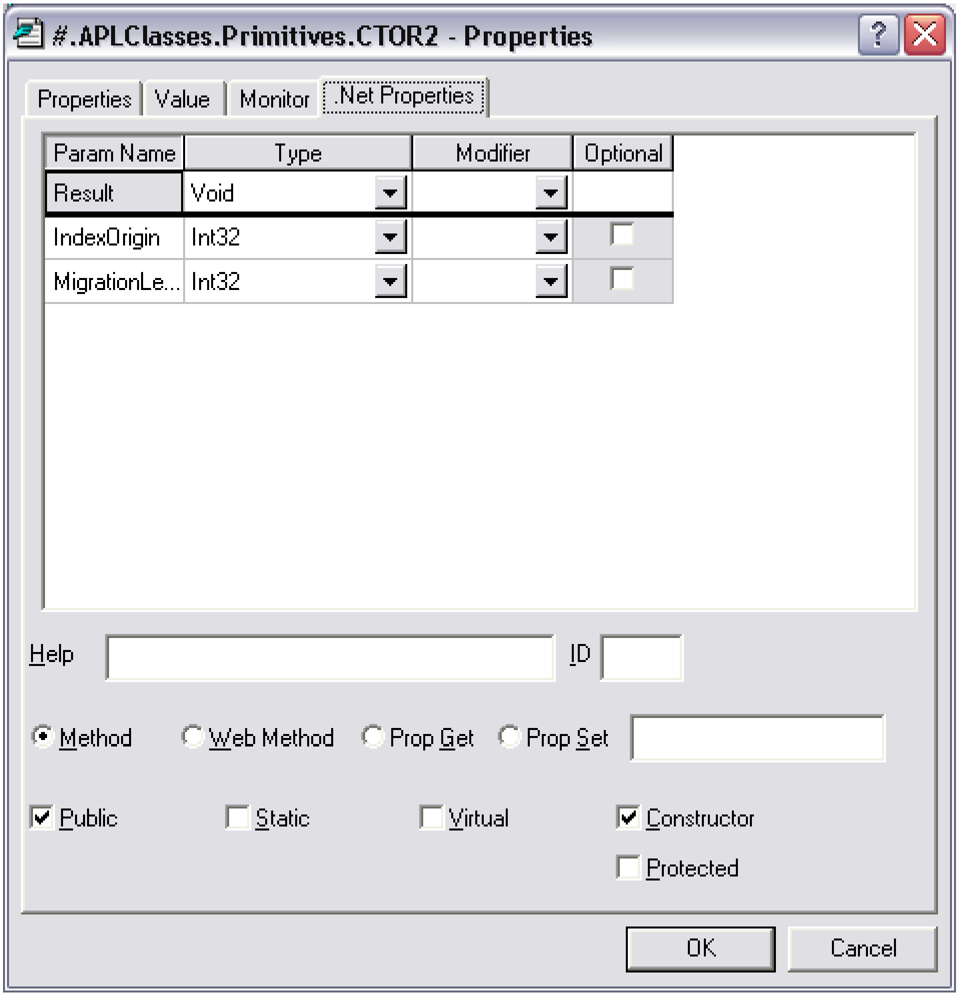
ID

● Method ○ Web Method ○ Prop Get ○ Prop Set

☑ Public ☐ Static ☐ Virtual ☑ Constructor

☐ Protected

.NET Properties for *CTOR2*



#.APLClasses.Primitives.CTOR2 - Properties

Param Name	Type	Modifier	Optional
Result	Void		
IndexOrigin	Int32		☐
MigrationLe...	Int32		☐

Help

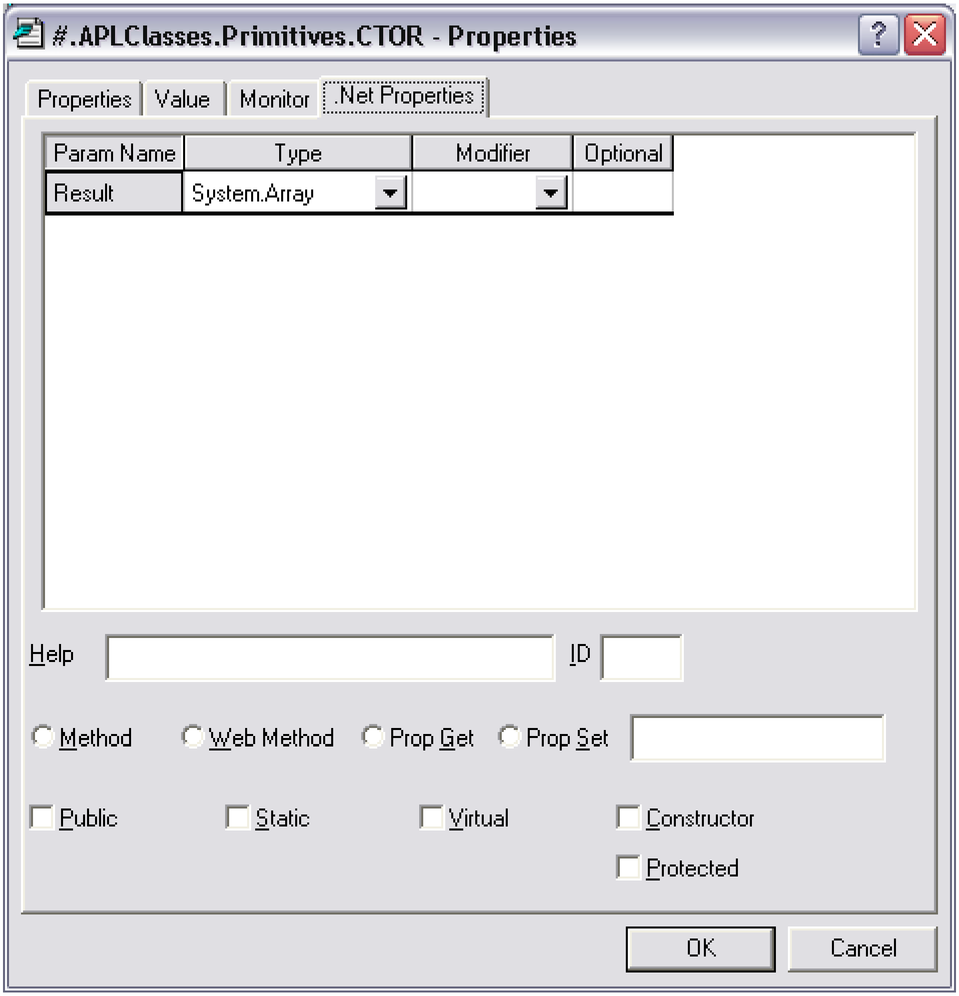
ID

● Method ○ Web Method ○ Prop Get ○ Prop Set

☑ Public ☐ Static ☐ Virtual ☑ Constructor

☐ Protected

.NET Properties for *CTOR*



Param Name	Type	Modifier	Optional
Result	System.Array		

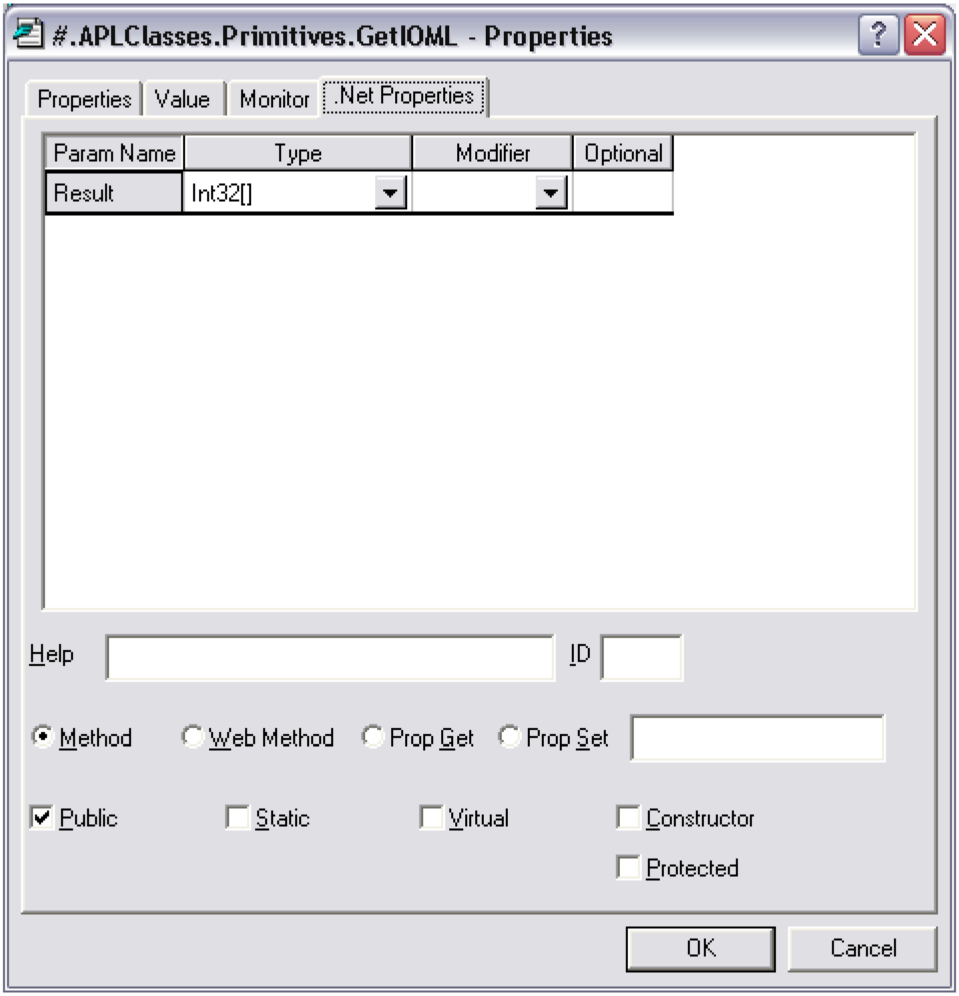
○ Method ○ Web Method ○ Prop Get ○ Prop Set

☐ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

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]     R←⎕IO ⎕ML
      ∇
```



Param Name	Type	Modifier	Optional
Result	Int32[]		

Help

ID

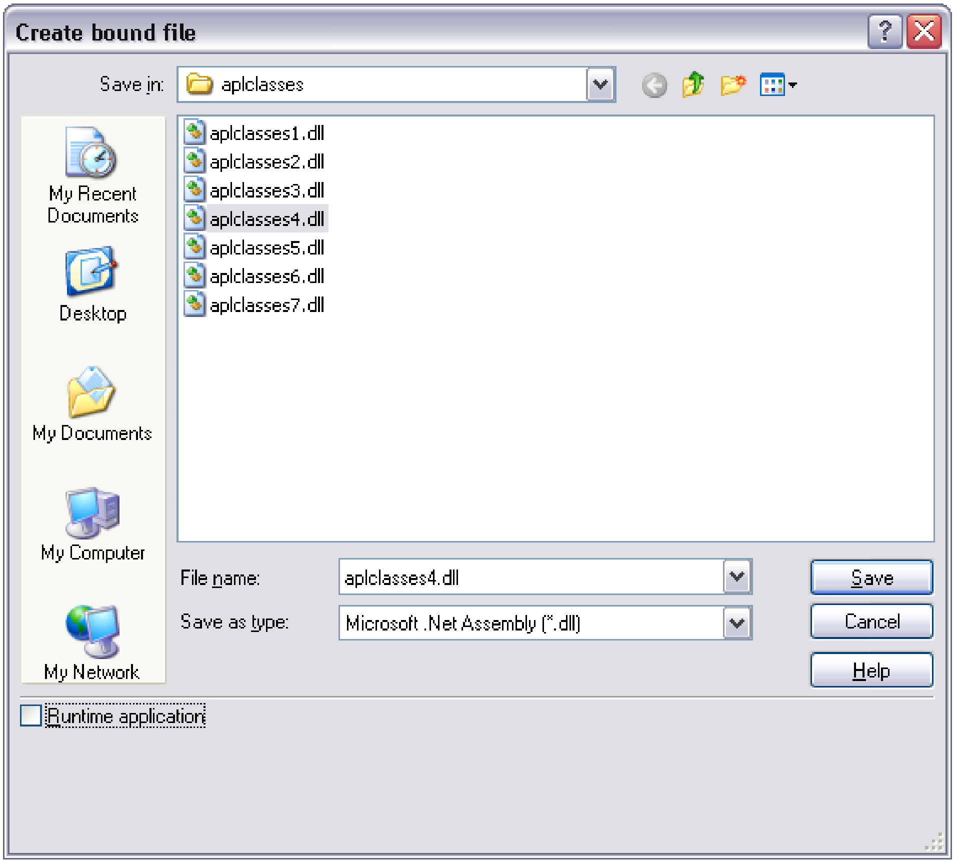
● Method ○ Web Method ○ Prop Get ○ Prop Set

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

Having done this, the workspace is renamed aplclasses4.dws, and saved, and a new Assembly `aplclasses4.dll` is built.

```
      )WSID  samples\APLClasses\aplclasses4
was samples\APLClasses\aplclasses4
     )SAVE
samples\APLClasses\aplclasses4 saved Wed Nov 21 13:51:22 2001
```

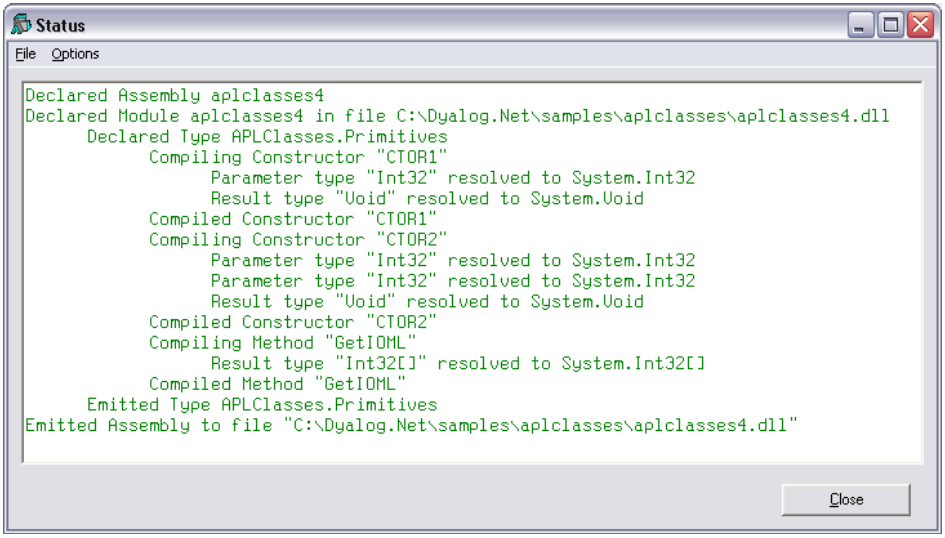


Save in: aplclasses

File name: aplclasses4.dll

Save as type: Microsoft .Net Assembly (*.dll)

☐ Runtime application



```
Declared Assembly aplclasses4
Declared Module aplclasses4 in file C:\Dyalog.Net\samples\aplclasses\aplclasses4.dll
      Declared Type APLClasses.Primitives
            Compiling Constructor "CTOR1"
                  Parameter type "Int32" resolved to System.Int32
                  Result type "Void" resolved to System.Void
            Compiled Constructor "CTOR1"
            Compiling Constructor "CTOR2"
                  Parameter type "Int32" resolved to System.Int32
                  Parameter type "Int32" resolved to System.Int32
                  Result type "Void" resolved to System.Void
            Compiled Constructor "CTOR2"
            Compiling Method "GetIOML"
                  Result type "Int32[]" resolved to System.Int32[]
            Compiled Method "GetIOML"
      Emitted Type APLClasses.Primitives
Emitted Assembly to file "C:\Dyalog.Net\samples\aplclasses\aplclasses4.dll"
```

aplfns4.cs

The following C# source, called `samples\APLClasses\aplfns4.cs`, may be used to invoke the two different constructor functions *`CTOR1`* and *`CTOR2`* in the new aplclasses4.dll Assembly.

```
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]);
            }
      }
```

In this example, 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

Then, when the program is compiled and run …

```
APLClasses>setpath.bat
...
APLClasses>csc /r:APLClasses4.dll aplfns4.cs
Microsoft (R)Visual C# Compiler Version 7.00.9254 [CLR
version 1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights
reserved.

APLClasses>aplfns4
1
0
0
3
```

Example 5

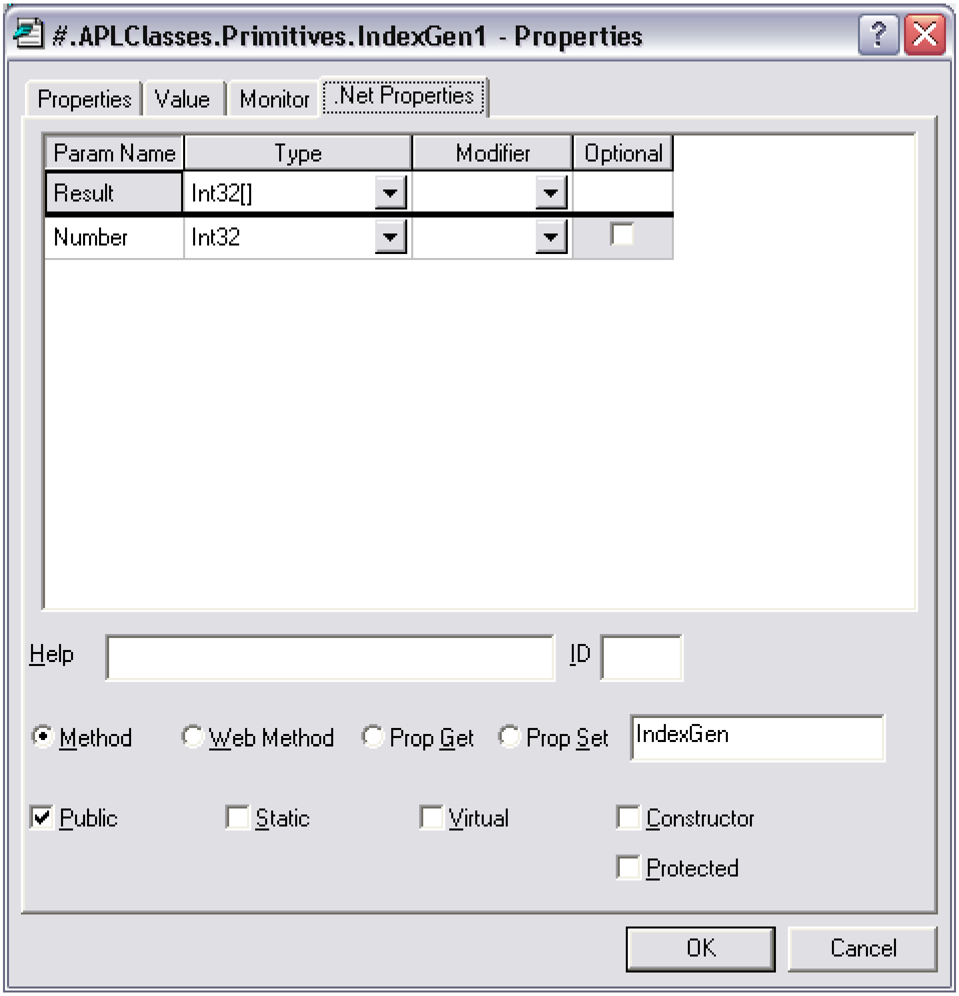
This example takes things a stage further and illustrates how you can implement *method overloading*.

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 *.NET Properties*, 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.

```
      )LOAD  samples\APLClasses\aplclasses5
samples\APLClasses\aplclasses5 saved Wed Nov 21 12:23:18 2001
      )CS APLClasses.Primitives
#.APLClasses.Primitives
      )FNS
CTOR    CTOR1   CTOR2   IndexGen1    IndexGen2    IndexGen3

     ∇ R←IndexGen1 N
[1]    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`.



Param Name	Type	Modifier	Optional
Result	Int32[]		
Number	Int32		☐

Help

ID

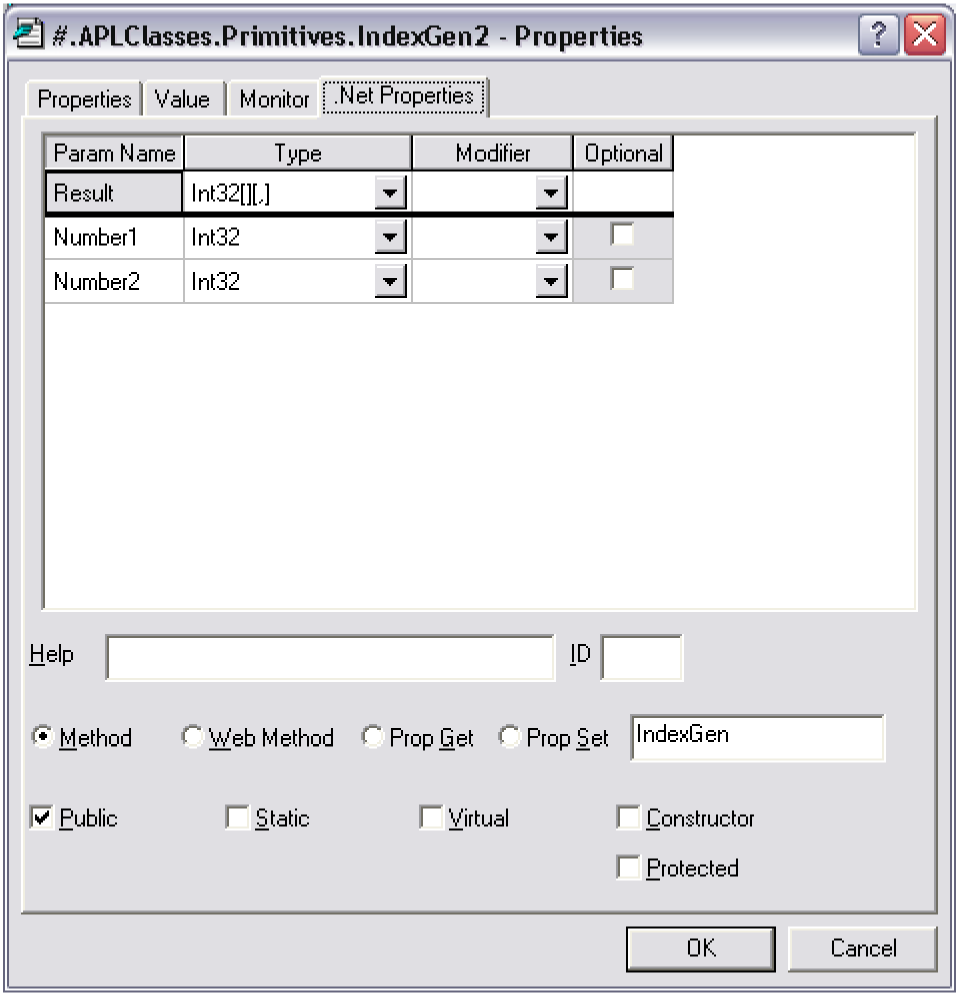
● Method ○ Web Method ○ Prop Get ○ Prop Set IndexGen

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

```
      ∇ R←IndexGen2 N
[1]     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`.



#.APLClasses.Primitives.IndexGen2 - Properties

Param Name	Type	Modifier	Optional
Result	Int32[][,]		
Number1	Int32		☐
Number2	Int32		☐

Help

ID

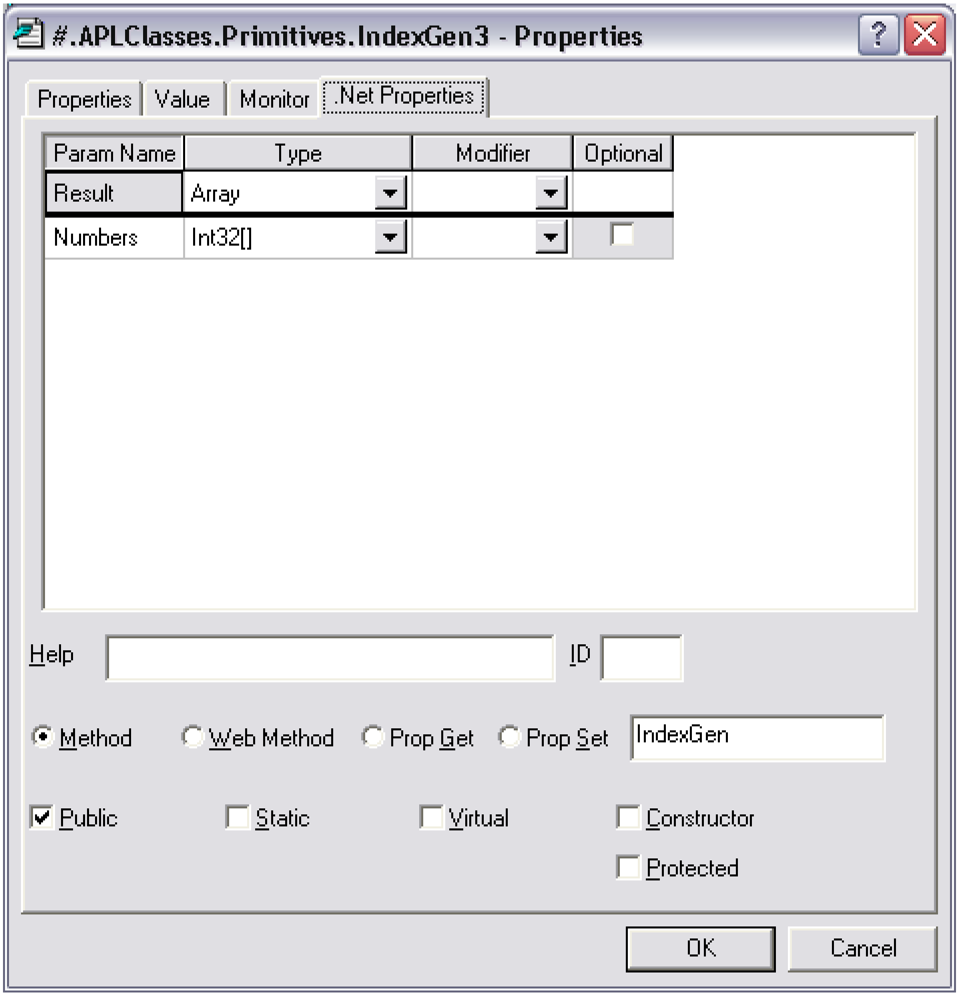
● Method ○ Web Method ○ Prop Get ○ Prop Set IndexGen

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

```
      ∇ R←IndexGen3 N
[1]     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.



#.APLClasses.Primitives.IndexGen3 - Properties

Param Name	Type	Modifier	Optional
Result	Array		
Numbers	Int32[]		☐

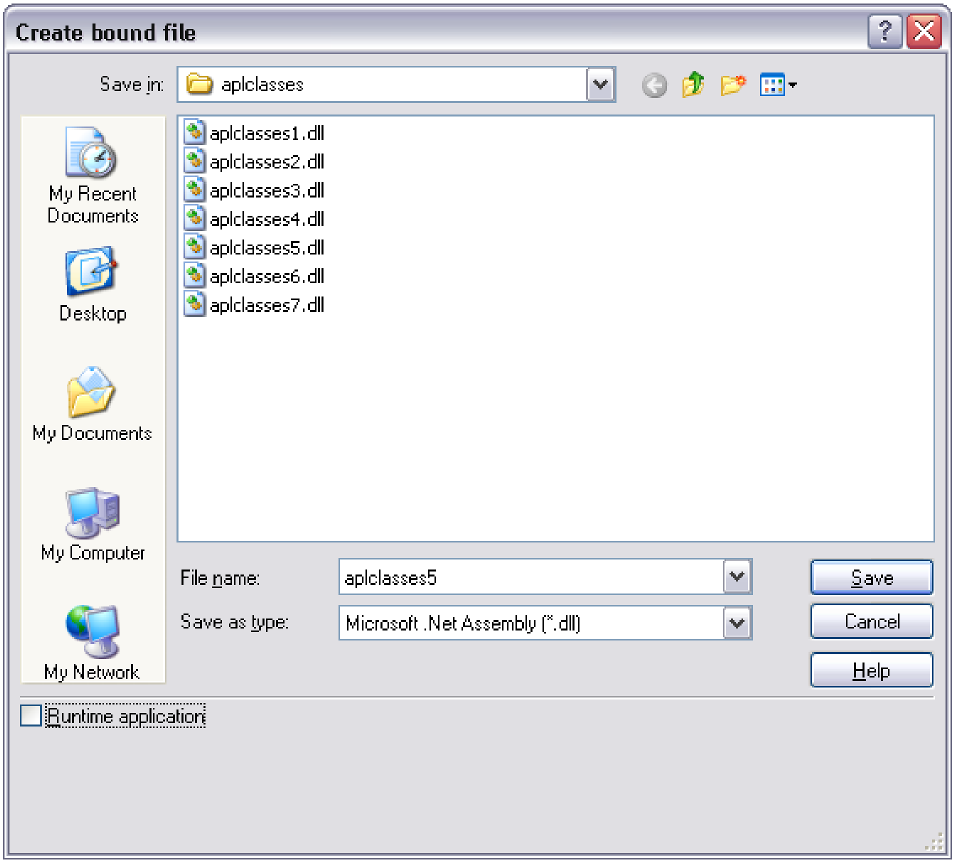
Help

ID

● Method ○ Web Method ○ Prop Get ○ Prop Set IndexGen

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected



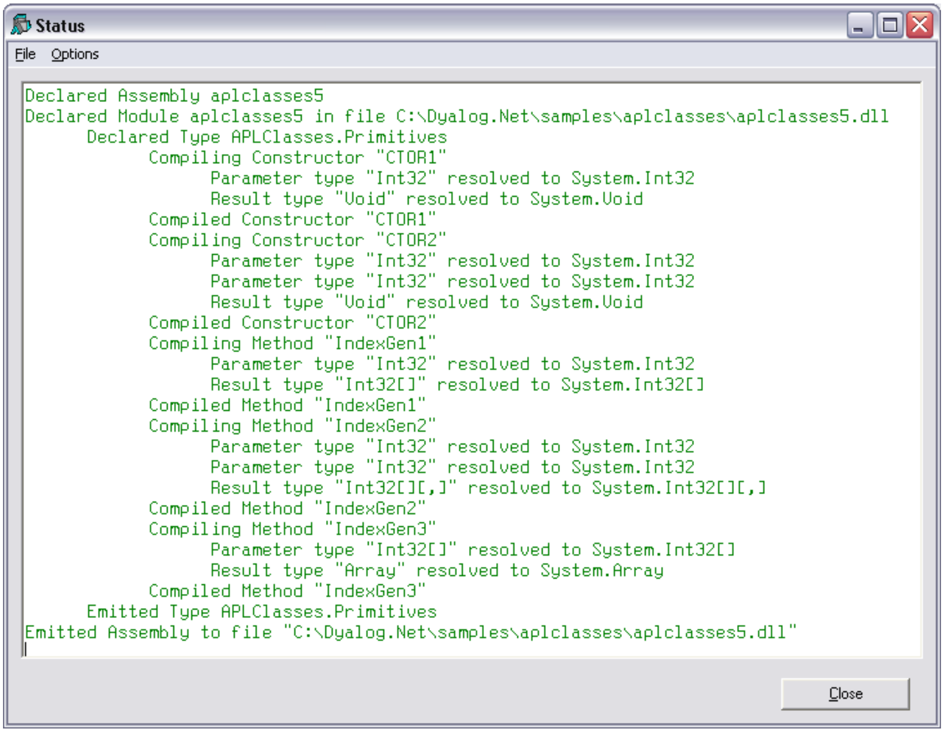
Save in: aplclasses

aplclasses1.dll
aplclasses2.dll
aplclasses3.dll
aplclasses4.dll
aplclasses5.dll
aplclasses6.dll
aplclasses7.dll

File name: aplclasses5

Save as type: Microsoft .Net Assembly (*.dll)

☐ Runtime application



```
Declared Assembly aplclasses5
Declared Module aplclasses5 in file C:\Dyalog.Net\samples\aplclasses\aplclasses5.dll
      Declared Type APLClasses.Primitives
            Compiling Constructor "CTOR1"
                  Parameter type "Int32" resolved to System.Int32
                  Result type "Void" resolved to System.Void
            Compiled Constructor "CTOR1"
            Compiling Constructor "CTOR2"
                  Parameter type "Int32" resolved to System.Int32
                  Parameter type "Int32" resolved to System.Int32
                  Result type "Void" resolved to System.Void
            Compiled Constructor "CTOR2"
            Compiling Method "IndexGen1"
                  Parameter type "Int32" resolved to System.Int32
                  Result type "Int32[]" resolved to System.Int32[]
            Compiled Method "IndexGen1"
            Compiling Method "IndexGen2"
                  Parameter type "Int32" resolved to System.Int32
                  Parameter type "Int32" resolved to System.Int32
                  Result type "Int32[][,]" resolved to System.Int32[][,]
            Compiled Method "IndexGen2"
            Compiling Method "IndexGen3"
                  Parameter type "Int32[]" resolved to System.Int32[]
                  Result type "Array" resolved to System.Array
            Compiled Method "IndexGen3"
      Emitted Type APLClasses.Primitives
Emitted Assembly to file "C:\Dyalog.Net\samples\aplclasses\aplclasses5.dll"
```

aplfns5.cs

The following C# source, called `samples\APLClasses\aplfns5.cs`, may be used to invoke the three different variants of `IndexGen`, in the new aplclasses5.dll Assembly .

```
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);

}
```

Then, when the program is compiled and run …

```
APLClasses>setpath.bat
...
APLClasses>csc /r:APLClasses5.dll aplfns5.cs
Microsoft (R)Visual C# Compiler Version 7.00.9254 [CLR
version 1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights
reserved.

APLClasses>aplfns5
0,1,2,3,4,5,6,7,8,9
(0,0)(0,1)(0,2)
(1,0)(1,1)(1,2)
System.Object[,,]
```

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 static or virtual methods, static 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 create a *`NetType`* object using *`⎕WC`* you may specify which interfaces it provides by specifying a value for the *`Interfaces`* property. You must do this when you create the object; you cannot subsequently change the value of the *`Interfaces`* property.

The value of *`Interfaces`* is a character vector containing a comma-separated list of `Interface` names. Each of these must be the full name of an existing Interface, which is defined in one of the .NET Namespaces installed on your computer. Note that *`⎕USING`* is not used to locate an Interface name; you must specify it fully. However, the .NET Assembly that defines the Interface must be included in *`⎕USING`*.

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 `TemperatureControlCtl1` custom control described in Chapter 10. This is built on top of the `System.Web.UI.Control` class, which 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.

The `TemperatureControlCtl2` custom control example described in Chapter 10 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.

See these examples in Chapter 10 for further details.

CHAPTER 5

Dyalog APL and IIS

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 comes with NT XP Professional and Windows 2000; 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 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 COM 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*.

IIS Applications and Virtual Directories

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 `Dyalog10\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 named `apl.net`, `apl.net/Golf`, and so forth. When you want to run the Web Services and Web Page examples, you do so by specifying the URL `http://localhost/apl.net/…`

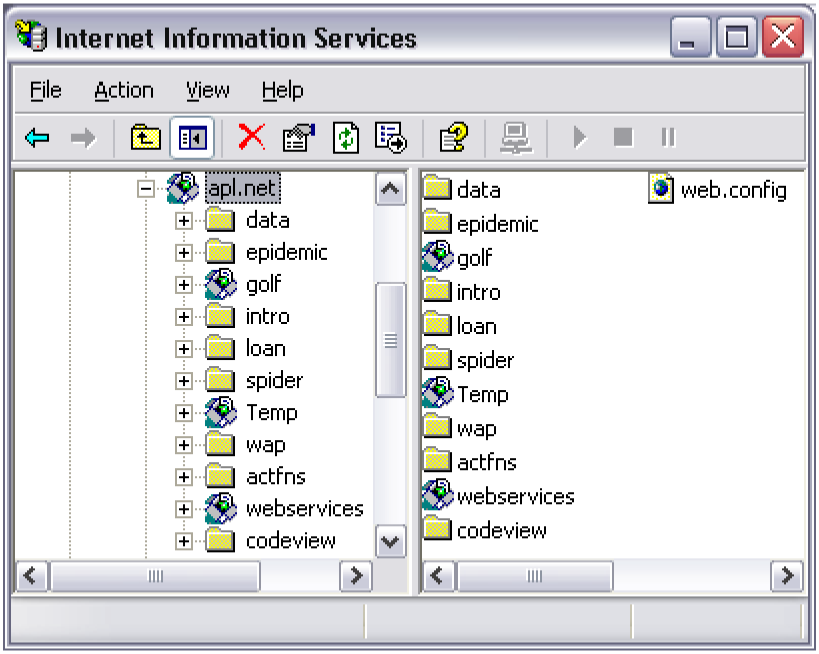
Internet Services Manager

As its name suggests, Internet Services Manager is a tool for managing IIS. If you are going to develop 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 Services Manager*, and select *Send To Desktop (create shortcut)*.

The *apl.net* Virtual Directory

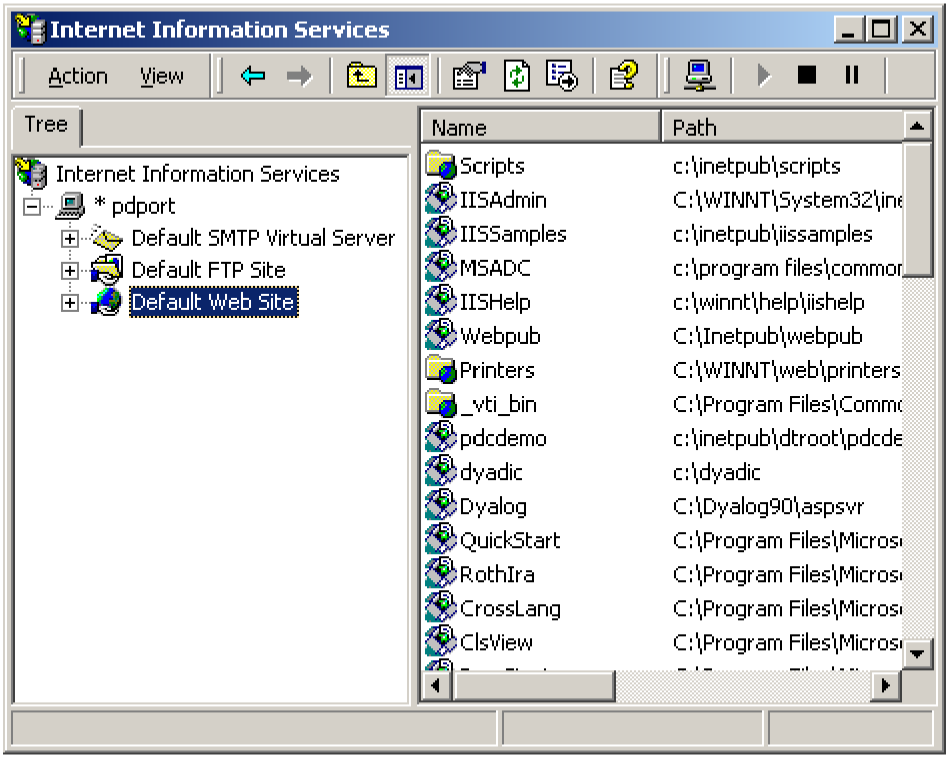
Following a successful installation of Dyalog APL, the apl.net Virtual Directory should appear in Internet Services Manager as shown below.



In case you need to set up your own IIS Virtual Directories yourself, the procedure is described below.

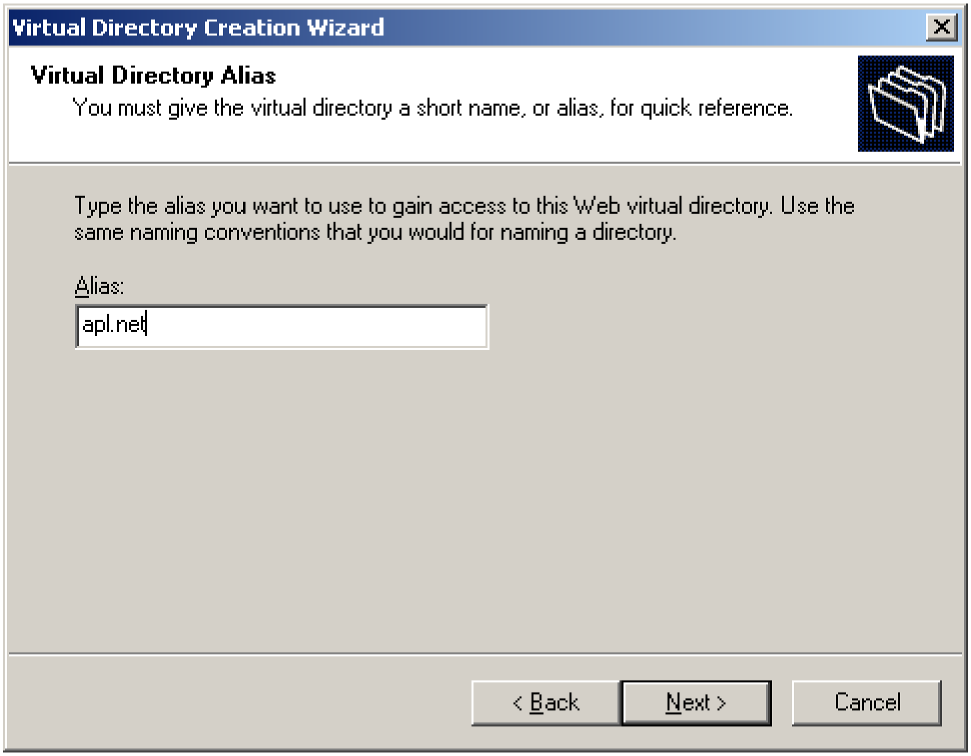
Creating the *apl.net* Virtual Directory

Start *Internet Services Manager*, open the icon associated with your computer (in this case, *pdport*) and select *Default Web Site* (or whatever it is called).


Internet Information Services
Action
View
Tree
Internet Information Services
* pdport
Default SMTP Virtual Server
Default FTP Site
Default Web Site

Name	Path
Scripts	c:\inetpub\scripts
IISAdmin	C:\WINNT\System32\ine
IISSamples	c:\inetpub\iissamples
MSADC	c:\program files\commor
IISHelp	c:\winnt\help\iishelp
Webpub	C:\Inetpub\webpub
Printers	C:\WINNT\web\printers
_vti_bin	C:\Program Files\Commo
pdcdemo	c:\inetpub\dtroot\pdcde
dyadic	c:\dyadic
Dyalog	C:\Dyalog90\aspsvr
QuickStart	C:\Program Files\Micros
RothIra	C:\Program Files\Micros
CrossLang	C:\Program Files\Micros
ClsView	C:\Program Files\Micros

Select *New Virtual Directory* from the *Action* menu or from the item's context menu. This brings up the *Virtual Directory Creation Wizard*. Click *Next* to bring up the first page and enter **apl.net** into the *Alias* field.



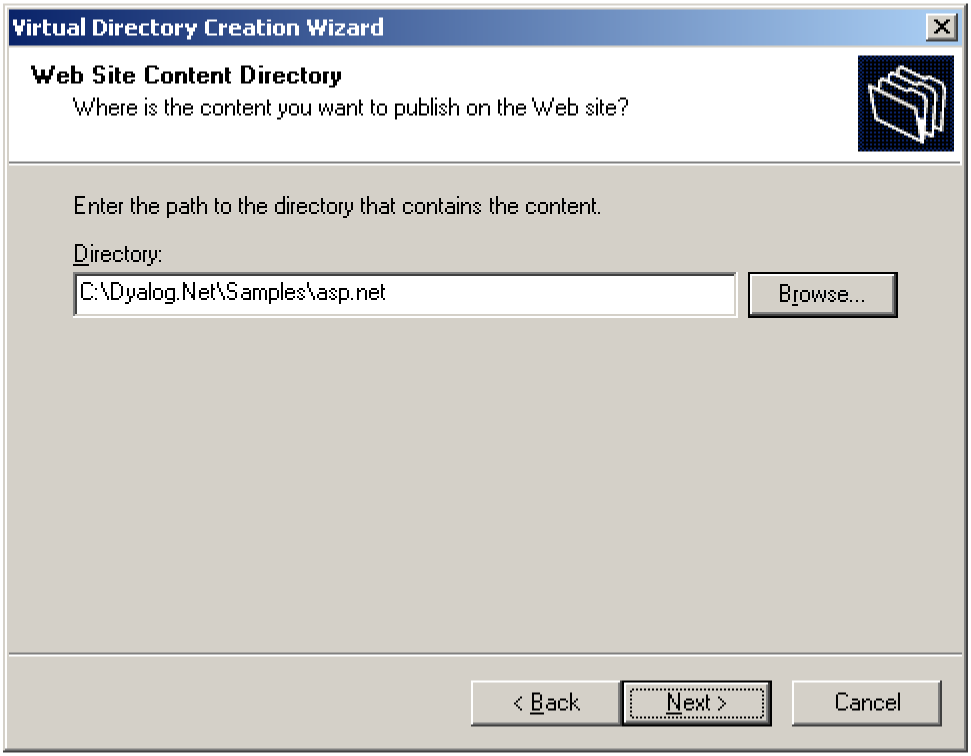
Virtual Directory Alias

You must give the virtual directory a short name, or alias, for quick reference.

Type the alias you want to use to gain access to this Web virtual directory. Use the same naming conventions that you would for naming a directory.

Alias: apl.net

Click Next, then enter the full pathname to the `Dyalog10\samples\asp.net` directory as shown below.



Virtual Directory Creation Wizard

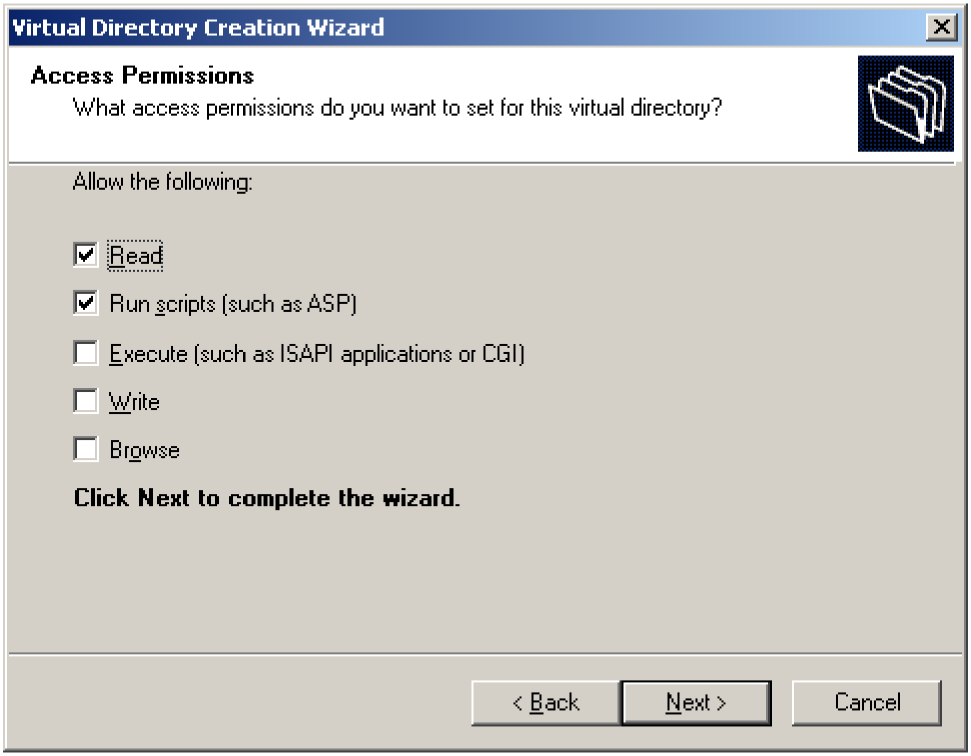
Web Site Content Directory

Where is the content you want to publish on the Web site?

Enter the path to the directory that contains the content.

Directory: C:\Dyalog.Net\Samples\asp.net

Accept the default Access Permissions, as shown below, and click *Next*.



Access Permissions

What access permissions do you want to set for this virtual directory?

Allow the following:

☑ Read

☑ Run scripts (such as ASP)

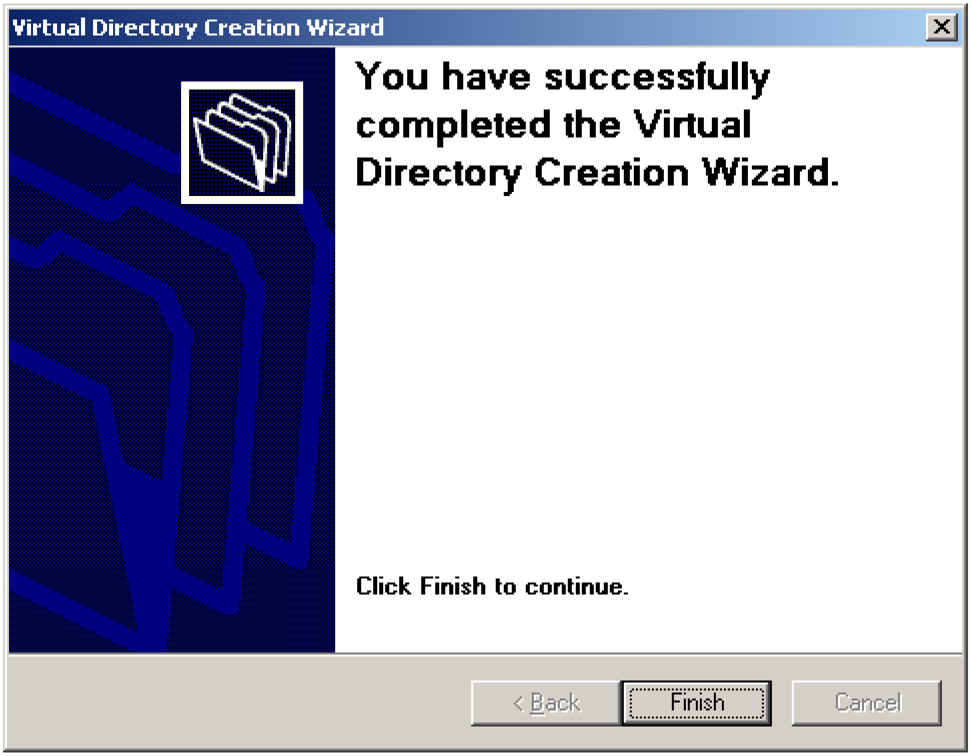
☐ Execute (such as ISAPI applications or CGI)

☐ Write

☐ Browse

Click Next to complete the wizard.

Then finally, click *Finish*.



You have successfully completed the Virtual Directory Creation Wizard.

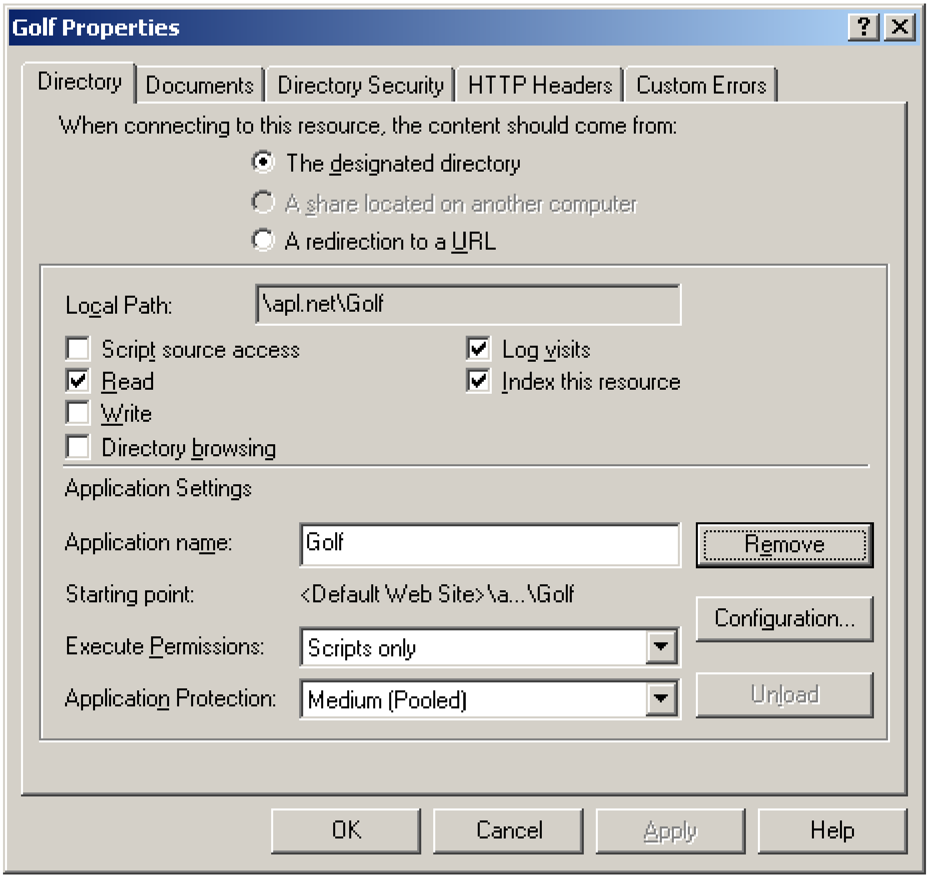
Click Finish to continue.

Creating the *apl.net* Virtual Sub-Directories

The `golf`, `temp` and `webservices` sub-directories in `apl.net` represent separate IIS Applications, so these need to be registered as IIS Virtual Directories too.

Open the newly created *apl.net* item shown in the left pane of *Internet Services Manager*, bring up the context menu of the *Golf* sub-directory, and select *Properties*.

Click the *Create* button; this turns the sub-directory into an IIS Virtual Directory (a separate IIS application) named *Golf*.



Golf Properties

When connecting to this resource, the content should come from:

● The designated directory
○ A share located on another computer
○ A redirection to a URL

Local Path: \apl.net\Golf

☐ Script source access
☑ Read
☐ Write
☐ Directory browsing
☑ Log visits
☑ Index this resource

Application Settings

Application name: Golf

Starting point: <Default Web Site>\a...\Golf

Execute Permissions: Scripts only

Application Protection: Medium (Pooled)

Note that the *Application Protection* entry dictates whether your application is loaded into the IIS process (Low), a shared DLLHost task (Medium) or its own DLLHost task (High). The last choice isolates your application from all other IIS applications, and is the safest option.

Follow the same procedure to define `Temp` and `webservices` as Virtual Directories (as sub-directories of `apl.net`).

It is not necessary to do this now, but you will need to do this during development and it won’t hurt now.

Restart IIS. You can do this from the context menu of the item associated with your computer at the top of the tree. Restarting IIS causes it to unload all the assemblies associated with your Applications.

CHAPTER 6

Writing Web Services

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 simple 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 around the Internet without the need to specially configure firewalls.

A Web Service is a programmable 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 (i.e. executed) by ASP.NET running under Microsoft IIS. Any one Web Services 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).

Web Service (.asmx) Scripts

Web Services may be written in a variety of languages, including `APLScript`, the scripting version of Dyalog APL (see Chapter 10).

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="apl" Class="GolfService" %>
```

Note that `Language="apl"` is specifically connected to the Dyalog APL script compiler through the ASP.NET system file `Machine.config`.

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.

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 APL 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 APL Script compiler creates a `DLL` containing a workspace, which itself contains a single namespace that represents the Web Service class. The class namespace 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 result in a namespace running with `⎕ML←2` containing a single function `FOO` and a variable `X`.

```
:Class
⎕ML←2
X←10
∇Z←FOO Y
Z←Y+X
∇
:EndClass
```

Note that all expressions are executed by the script compiler when it creates the class namespace. 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 namespace. This may be used to import functions from other workspaces and obviate the need to include them in the `.asmx` file.

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.

`:Returns type`

This statement declares the data type of the result of the method where *type* may specify any valid .NET type that is supported by Web Services.

`:ParameterList type1 name1, type2 name2, ...`

This statement declares one or more input parameters to the method with specified *type* and *name*. The declaration of each parameter is separated from the next by a comma (,). Each *type* may specify any valid .NET type that is supported by Web Services. Each *name* may be any ASCII character string. Note that names are optional.

`:Implements type name`

This statement supports *method overloading* and *Property Get/Put* functions. The type may be one of the terms `Method`, `PropGet` or `PropSet`. The *name* specifies the public name of the method that this function is exported as, or the public name of the property that this function *gets* or *sets*.

Add1

```
∇ R←Add1 args
:Access WebMethod
:ParameterList Int32 arg1,Int32 arg2
:Returns Int32
R←+/args
∇
```

The `Add1` 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`. Armed with this definition, that 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
[1]    :Access WebMethod
[2]    :ParameterList Double[] arg1
[3]    :Returns:Double
[4]     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`.

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:

1. Multi-dimensional arrays are not supported; only vectors.
2. 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.

Execution

When your Web Service 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 APL Web Service or Web Page, the Dyalog APL dynamic link library (see Chapter 12) is loaded into the ASP.NET host process and the namespace corresponding to your Web Service class is *)COPYed* from the Assembly. the Dyalog APL dynamic link library then delivers an instance of this namespace to the client (calling) process.

In general, every call on a method in a Web Service causes a new instance of the Web Server class namespace to be created. If you need to maintain/update variables between calls, you have to put the variables in the parent namespace (##) of the class.

If a client invokes a different Dyalog APL Web Service or Web Page, its class namespace is *)COPY*ed from its Assembly into the workspace managed by the Dyalog APL dynamic link library. All Dyalog APL Web Services (and Web Pages) share the same workspace when they are invoked. This has implications for the way that you access and manage global resources such as component files.

Global.asax and 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, i.e. 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 APL 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.

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/apl.net/webservices/eg1.asmx`

```
<%@ WebService Language="apl" Class="APLExample" %>

:Class APLEXample:System.Web.Services.WebService

     ∇ R←Add args
[1]    :Access WebMethod
[2]    :ParameterList Int32 arg1, Int32 arg2
[3]    :Returns Int32
[4]    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 [4] could in fact be coded as:

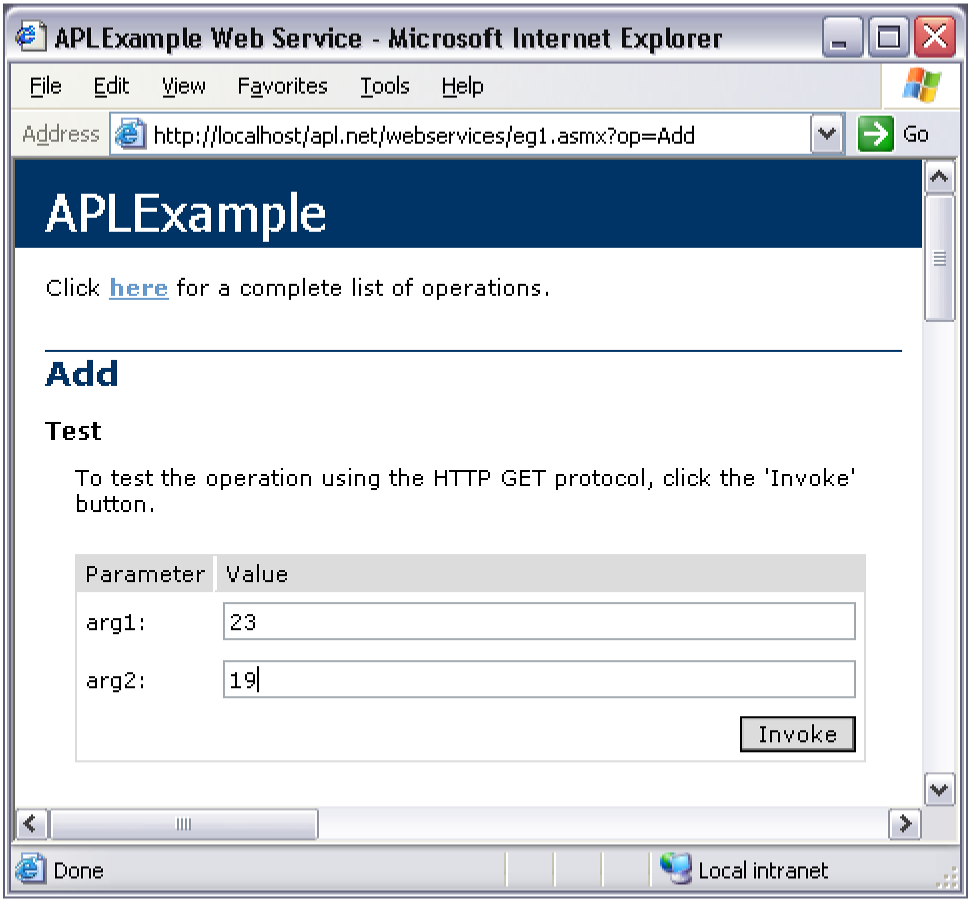
```
[4]    R←args[1]+args[2]
```

because .NET guarantees that a client can only call the method by providing two 32-bit integers as parameters.

Testing APLExample from IE5

If you connect, using Internet Explorer 5, to a URL that represents a Web Service, it displays a page that displays 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 IE5 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*.



http://localhost/apl.net/webservices/eg1.asmx?op=Add

APLExample

Click here for a complete list of operations.

Add

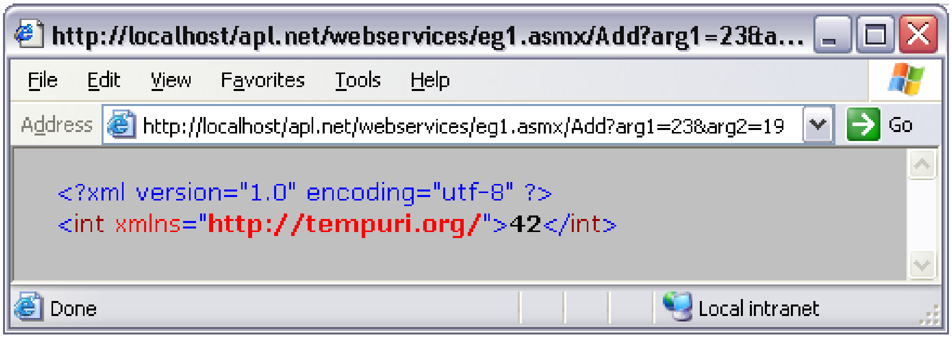
Test

To test the operation using the HTTP GET protocol, click the 'Invoke' button.

Parameter	Value
arg1:	23
arg2:	19

The following screen shot shows the result of entering the values 23 and 19 into the form fields and then pressing the Invoke method.

In this case, the method returns an `int` value 42.


http://localhost/apl.net/webservices/eg1.asmx/Add?arg1=23&a...
File Edit View Favorites Tools Help
Address http://localhost/apl.net/webservices/eg1.asmx/Add?arg1=23&arg2=19
Go
Done
Local intranet

```
<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://tempuri.org/">42</int>
```

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.

Sample Web Service: LoanService

The `LoanService` sample is supplied in `Dyalog10\Samples\asp.net\Loan\Loan.asmx`, which is mapped via an IIS Virtual Directory to the URL `http://localhost/apl.net/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="apl" Class="LoanService" %>

:Class LoanService:System.Web.Services.WebService

    :Class LoanResult
        :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]   :ParameterList Int32 LoanAmt, Int32 LenMax,
                    Int32 LenMin, Int32 IntrMax,
                    Int32 IntrMin
[3]   :Returns LoanResult
[4]
[5]  ⍝ Calculates loan repayments
[6]  ⍝ Argument X specifies:
[7]  ⍝   LoanAmt     Loan amount
[8]  ⍝   LenMax      Maximum loan period
[9]  ⍝   LenMin      Minimum loan period
[10] ⍝   IntrMax     Maximum interest rate
[11] ⍝   IntrMin     Minimum interest rate
[12]
[13]   LoanAmt LenMax LenMin IntrMax IntrMin←X
[14]   R←LoanResult.New ⍬
[15]   R.Periods←¯1+LenMin+⍳1+LenMax-LenMin
[16]   R.InterestRates←0.5×¯1+(2×IntrMin)+⍳1+2×
                      IntrMax-IntrMin
[17]   NI←⍴INTEREST←R.InterestRates÷100×12
[18]   NM←⍴PERIODS←R.Periods×12
[19]   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)`, but 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 are not initialised. In this case, that is sufficient, as all the fields will be filled in explicitly later.

```
:Class LoanResult
    :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.

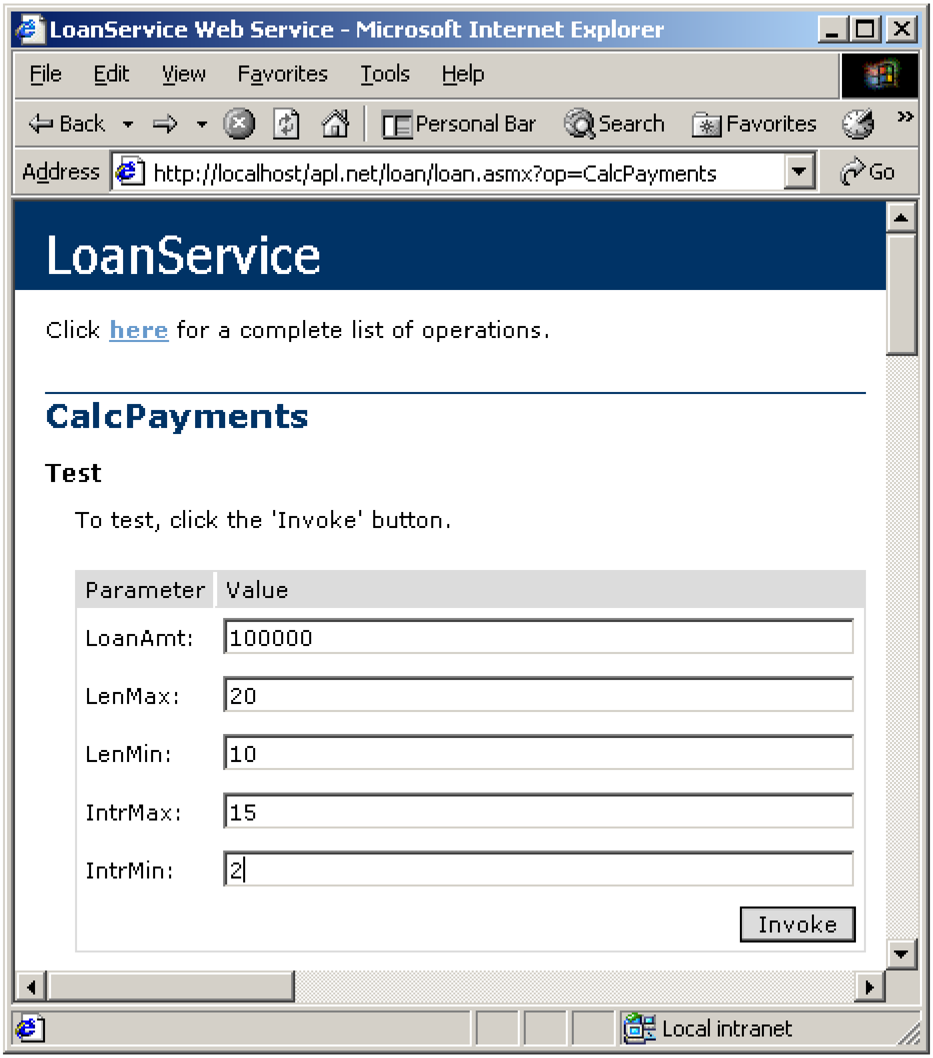
The `Periods` field is defined to be an array of integers; the `InterestRates` field an array of `Double`. Both these arrays are 1-diemsional, i.e. vectors. These will contain the numbers of years, and the different interest rates, to which the repayments matrix applies.

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[14]` gets a new instance of the `LoanResult` class by calling `LoanResult.New`. It then assigns values to each of the three fields in lines `[15]`, `[16]` and `[19]`.

Testing LoanService from IE5

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 IE5 at it is shown below.


LoanService Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost/apl.net/loan/loan.asmx?op=CalcPayments
Go

LoanService

Click here for a complete list of operations.

CalcPayments

Test

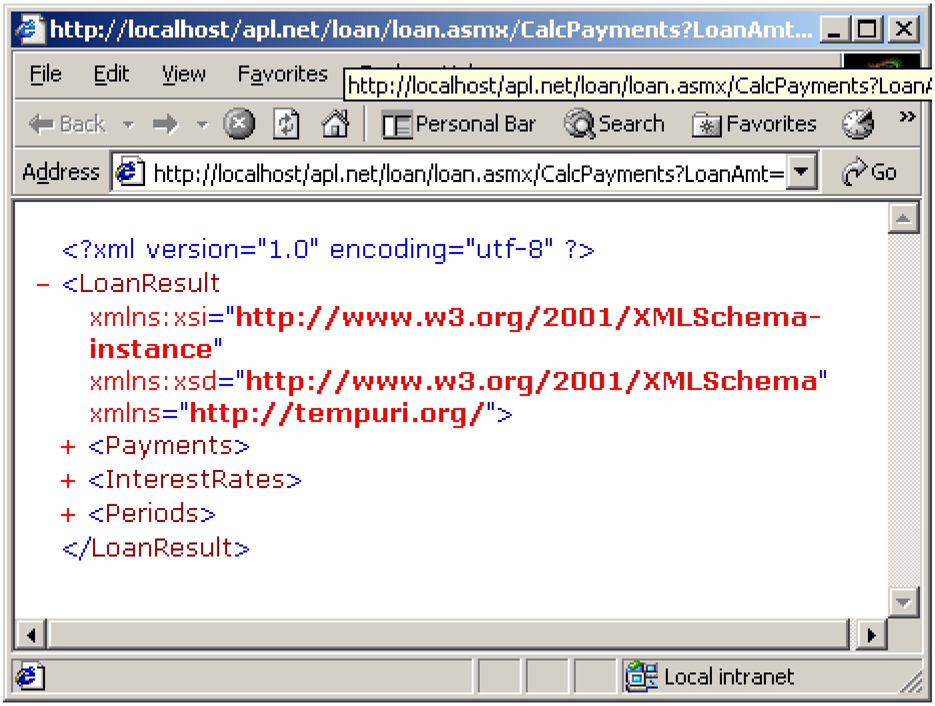
To test, click the 'Invoke' button.

Parameter	Value
LoanAmt:	100000
LenMax:	20
LenMin:	10
IntrMax:	15
IntrMin:	2

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.

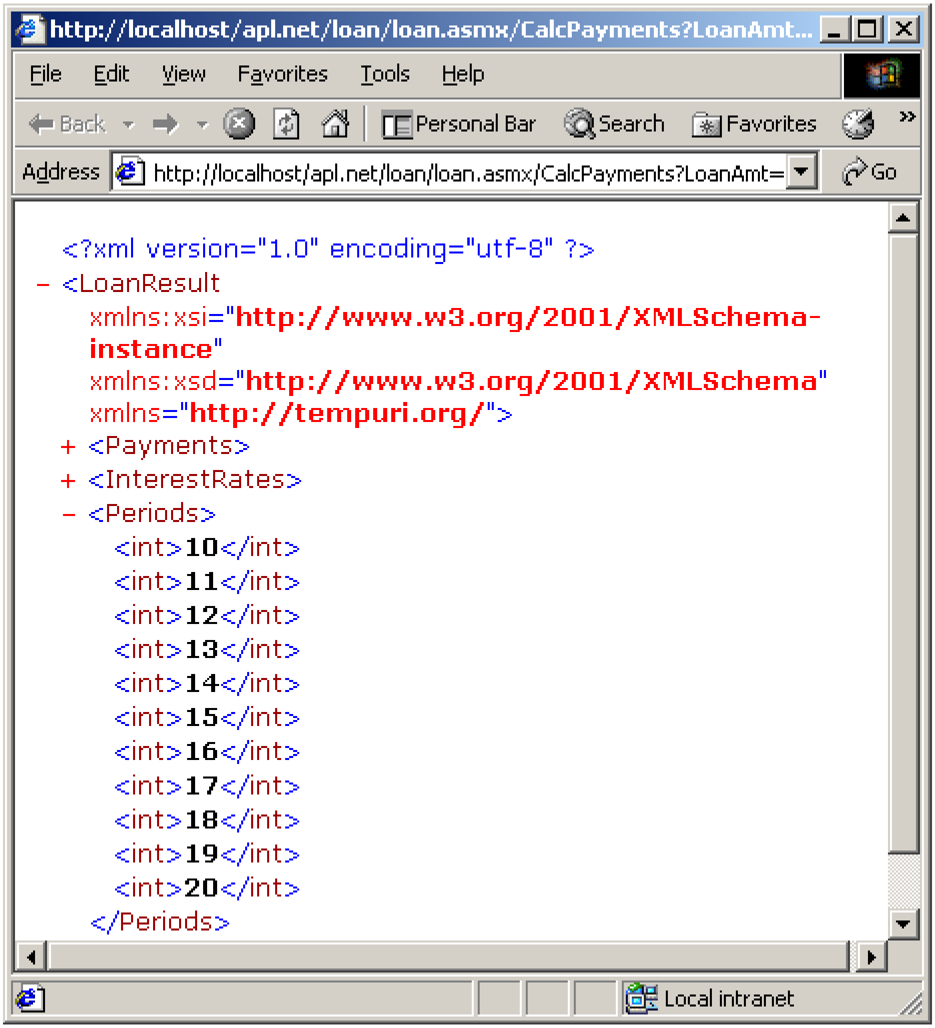
Notice that the result is described using XML, which is in fact the very language used to invoke a Web Service and return its result.

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.


http://localhost/apl.net/loan/loan.asmx/CalcPayments?LoanAmt...
File
Edit
View
Favorites
http://localhost/apl.net/loan/loan.asmx/CalcPayments?LoanA
Back
Personal Bar
Search
Favorites
Address
http://localhost/apl.net/loan/loan.asmx/CalcPayments?LoanAmt=
Go
Local intranet

```
  <?xml version="1.0" encoding="utf-8" ?>
- <LoanResult
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://tempuri.org/">
  + <Payments>
  + <InterestRates>
  + <Periods>
  </LoanResult>
```

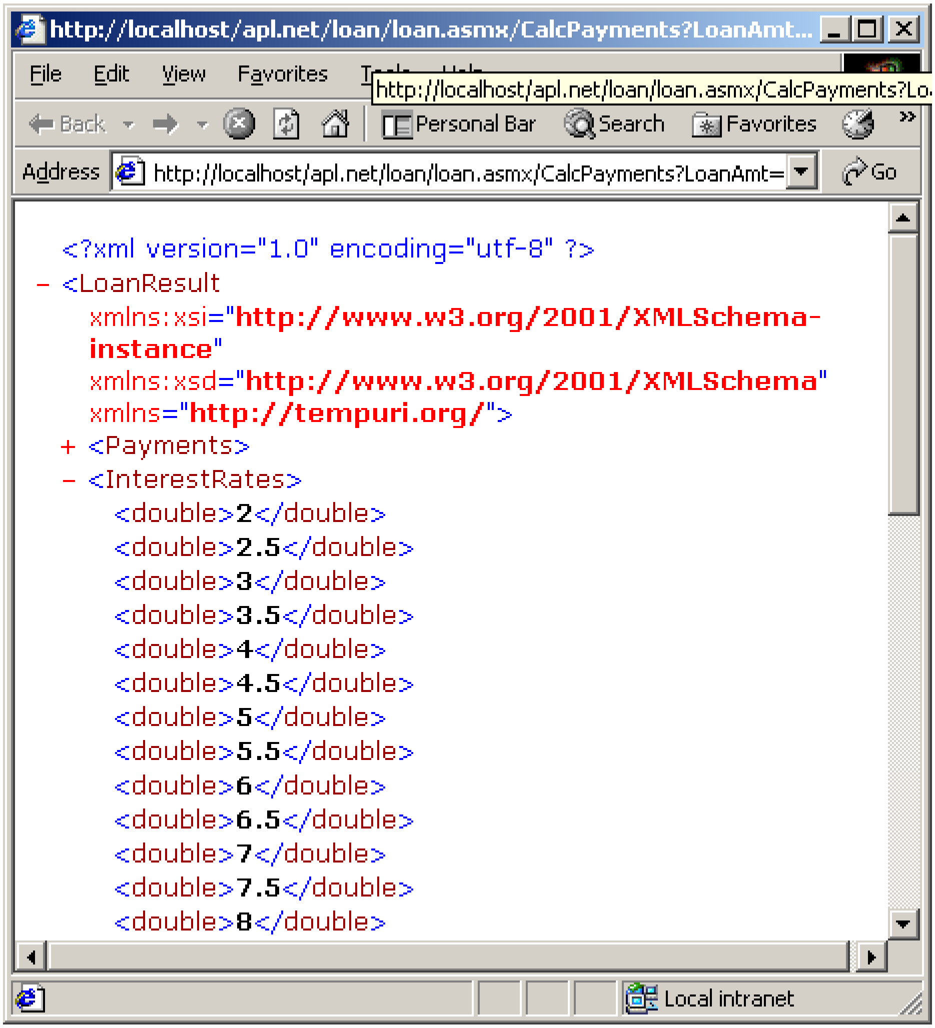
Opening the `Periods` field causes the page to display its value. As you can see, it contains a vector of integers from the minimum period to the maximum period that we specified on the input form, in increments of 1.



http://localhost/apl.net/loan/loan.asmx/CalcPayments?LoanAmt...
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost/apl.net/loan/loan.asmx/CalcPayments?LoanAmt= Go
Local intranet

```
  <?xml version="1.0" encoding="utf-8" ?>
- <LoanResult
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://tempuri.org/">
  + <Payments>
  + <InterestRates>
  - <Periods>
      <int>10</int>
      <int>11</int>
      <int>12</int>
      <int>13</int>
      <int>14</int>
      <int>15</int>
      <int>16</int>
      <int>17</int>
      <int>18</int>
      <int>19</int>
      <int>20</int>
    </Periods>
```

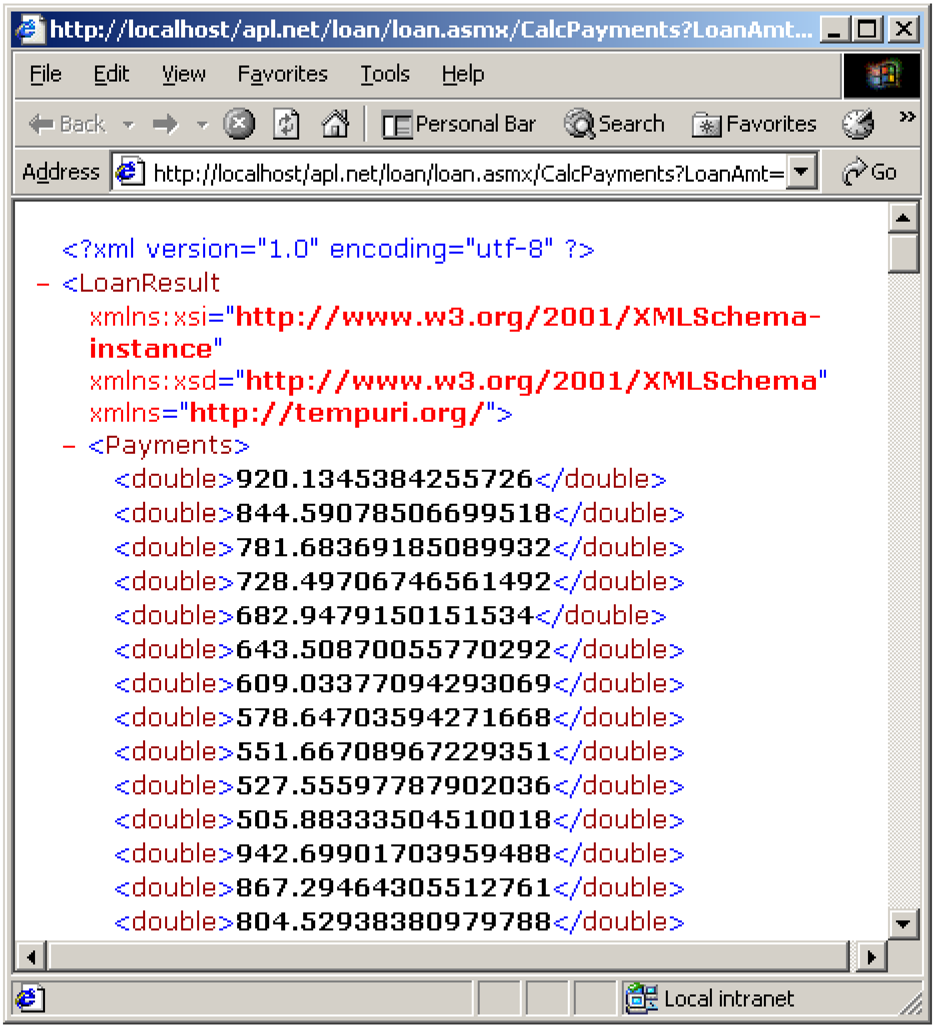
Similarly, opening the `InterestRates` field shows that it contains a vector of floating-point values (`double`) from the minimum rate to the maximum rate that we specified on the input form. This time, the increment is 0.5.


http://localhost/apl.net/loan/loan.asmx/CalcPayments?LoanAmt...
File
Edit
View
Favorites
Tools
Help
http://localhost/apl.net/loan/loan.asmx/CalcPayments?Lo
Back
Personal Bar
Search
Favorites
Address
Go
Local intranet

Address: http://localhost/apl.net/loan/loan.asmx/CalcPayments?LoanAmt=

```
  <?xml version="1.0" encoding="utf-8" ?>
- <LoanResult
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://tempuri.org/">
  + <Payments>
  - <InterestRates>
      <double>2</double>
      <double>2.5</double>
      <double>3</double>
      <double>3.5</double>
      <double>4</double>
      <double>4.5</double>
      <double>5</double>
      <double>5.5</double>
      <double>6</double>
      <double>6.5</double>
      <double>7</double>
      <double>7.5</double>
      <double>8</double>
```

Finally, the `Payments` field contains the calculated repayment values.



Address http://localhost/apl.net/loan/loan.asmx/CalcPayments?LoanAmt=

```
  <?xml version="1.0" encoding="utf-8" ?>
- <LoanResult
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://tempuri.org/">
  - <Payments>
      <double>920.1345384255726</double>
      <double>844.59078506699518</double>
      <double>781.68369185089932</double>
      <double>728.49706746561492</double>
      <double>682.9479150151534</double>
      <double>643.50870055770292</double>
      <double>609.03377094293069</double>
      <double>578.64703594271668</double>
      <double>551.66708967229351</double>
      <double>527.55597787902036</double>
      <double>505.88333504510018</double>
      <double>942.69901703959488</double>
      <double>867.29464305512761</double>
      <double>804.52938380979788</double>
```

Sample Web Service: GolfService

`GolfService` is an example Web Service that resides in the directory `Dyalog10\samples\asp.net\Golf` and is associated with the IIS Virtual Directory `apl.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="apl" runat=server>

     ∇ Application_Start;GOLFID
[1]    :Access Public
[2]    GOLFID←'c:\Dyalog10\samples\asp.net\golf\GolfData'
             ⎕FTIE 0
[3]    Application.Item 'GOLFID' GOLFID
     ∇

     ∇ Application_End;GOLFID
[1]    :Access Public
[2]    :Trap 6
[3]        GOLFID←Application.Item'GOLFID'
[4]        ⎕FUNTIE GOLFID
[5]    :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, and 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. You may need to alter the file path first.

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
    :Field Public Int32 Code
    :Field Public String Name

        ∇ ctor args
    [1]  :Access Constructor
    [2]  :ParameterList Int32, String
    [3]   Code Name←args
        ∇
        ∇ ctor_def
    [1]  :Access Constructor
    [2]   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←GolfCourse.New ⍬
      GC.(Code Name)←1 'St Andrews'
```

Or, more simply

```
      GC←GolfCourse.New 1 'St Andrews'
```

Note that the names of the constructor functions are not visible outside the class. Constructors are identified by their signatures (basically, their parameters) 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
    :Field Public DateTime Time
    :Field Public String[] Players

        ∇ ctor1 arg
    [1]  :Access Constructor
    [2]  :ParameterList DateTime
    [3]   Time←arg
    [4]    Players← 0ρ⊂''
        ∇
        ∇ ctor2 args
    [1]  :Access Constructor
    [2]  :ParameterList DateTime, String[]
    [3]   Time Players←args
        ∇
        ∇ ctor_def
    [1]  :Access Constructor
        ∇
: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
    :Field Public Boolean OK
    :Field Public GolfCourse Course
    :Field Public Slot TeeTime
    :Field Public String Message

        ∇ ctor args
    [1]  :Access Constructor
    [2]  :ParameterList Boolean, GolfCourse, Slot, String
    [3]   OK Course TeeTime Message←args
        ∇
        ∇ ctor_def
    [1]  :Access 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
    :Field Public Boolean OK
    :Field Public GolfCourse Course
    :Field Public DateTime Date
    :Field Public Slot[] Slots
    :Field Public String Message

        ∇ ctor args
    [1]  :Access Constructor
    [2]  :ParameterList Boolean, GolfCourse, DateTime
    [3]   OK Course Date←args
        ∇

        ∇ ctor_def
    [1]  :Access Constructor
        ∇
: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]   :Returns GolfCourse[]
[4]
[5]    GOLFID←Application.Item'GOLFID'
[6]    COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
[7]    R←GolfCourse.New¨↓⍉↑COURSECODES COURSES
     ∇
```

The Get*Courses* 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 [11] creates an instance of a `StartingSheet` object and uses it to initialise the result *`R`*. The value of the `OK` field is set to 0 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[16] extracts the `Year`, `Month` and `Day` properties from the requested tee time, a `DateTime` object, and converts 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]   :ParameterList Int32 CourseCode, DateTime Date
[4]   :Returns StartingSheet
[5]
[6]     CODE DATE←ARGS
[7]     GOLFID←Application.Item'GOLFID'
[8]     COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
[9]     COURSEI←COURSECODES⍳CODE
[10]   COURSE←GolfCourse.New CODE (COURSEI⊃COURSES,⊂'')
[11]   R←StartingSheet.New 0 COURSE DATE
[12]   :If COURSEI>⍴COURSECODES
[13]       R.Message←'Invalid course code'
[14]       :Return
[15]   :EndIf
[16]   IDN←2 ⎕NQ'.' 'DateToIDN',DATE.(Year Month Day)
[17]   DATES COMPS←⎕FREAD GOLFID,COURSEI⊃INDEX
[18]   IDATE←DATES⍳IDN
[19]   :If IDATE>⍴DATES
[20]       R.Message←'No Starting Sheet available'
[21]       :Return
[22]   :EndIf
[23]   TEETIMES GOLFERS←⎕FREAD GOLFID,IDATE⊃COMPS
[24]   R.OK←1
[25]   T←DateTime.New¨(⊂DATE.(Year Month Day)),¨
                      ↓[1]24 60 1⊤TEETIMES
[26]   R.Slots←Slot.New¨ T,∘⊂¨↓GOLFERS
```

∇

Line[24] sets the `OK` field of the result to 1 (success).

Line[25] converts the stored tee times (in minutes) to `DateTime` objects.

Line[26] 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 24] 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 + TimeSpan.New 0 0 30
```

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]    :ParameterList Int32 CourseCode, DateTime TeeTime,
                      Boolean GimmeNearest, String Name1,
                      String Name2, String Name3, String Name4
[4]    :Returns Booking
[5]
[6]   ⍝ If GimmeNearest is 0, books (or fails) for
         specified time
[7]   ⍝ If GimmeNearest is 1, books (or fails) for
         nearest to specified time
[8]
[9]     CODE TEETIME NEAREST←3↑ARGS
[10]    GOLFID←Application.Item'GOLFID'
[11]    COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
[12]    COURSEI←COURSECODES⍳CODE
[13]    COURSE←GolfCourse.New CODE (COURSEI⊃COURSES,⊂'')
[14]    SLOT←Slot.New TEETIME
[15]    R←Booking.New 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←(24 60⊥7 0)+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
```

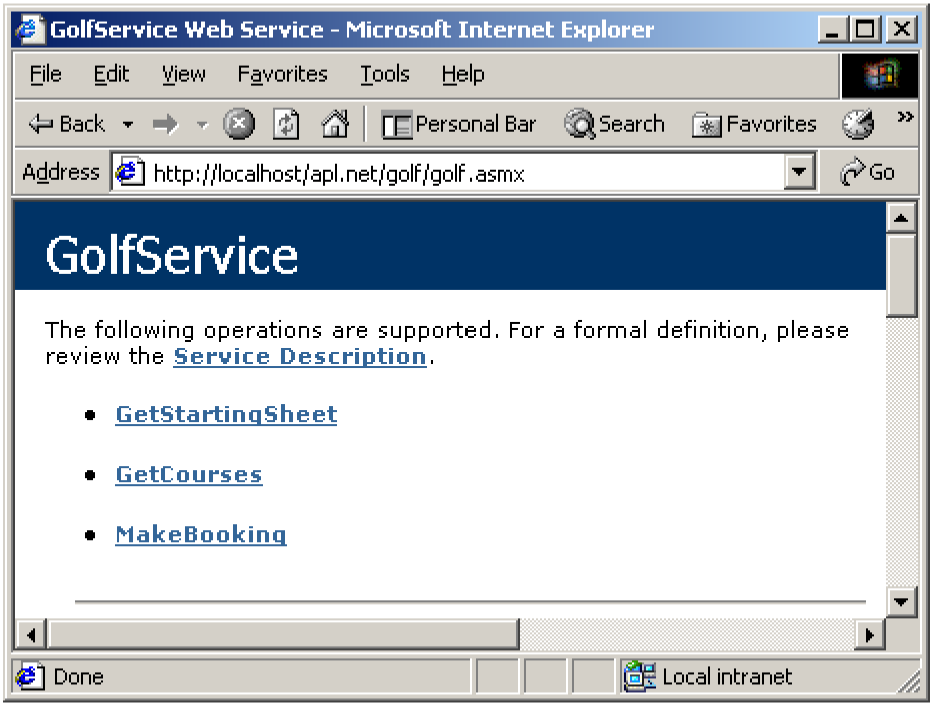
```
[34]      :If 0=OLD←⊃(DATES<2 ⎕NQ'.'
                                 'DateToIDN',3↑⎕TS)/⍳⍴DATES
[35]          COMP←(TEETIMES GOLFERS)⎕FAPPEND GOLFID
[36]          DATES,←IDN
[37]          COMPS,←COMP
[38]          (DATES COMPS)⎕FREPLACE GOLFID,COURSEI⊃INDEX
```

```
[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)~θ''
[50]    TIME←24 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←DateTime.New 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 IE5

If you point your browser at the URL `http://localhost/apl.net/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.


GolfService Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Personal Bar
Search
Favorites
Address http://localhost/apl.net/golf/golf.asmx
Go
Done
Local intranet

GolfService

The following operations are supported. For a formal definition, please review the Service Description.

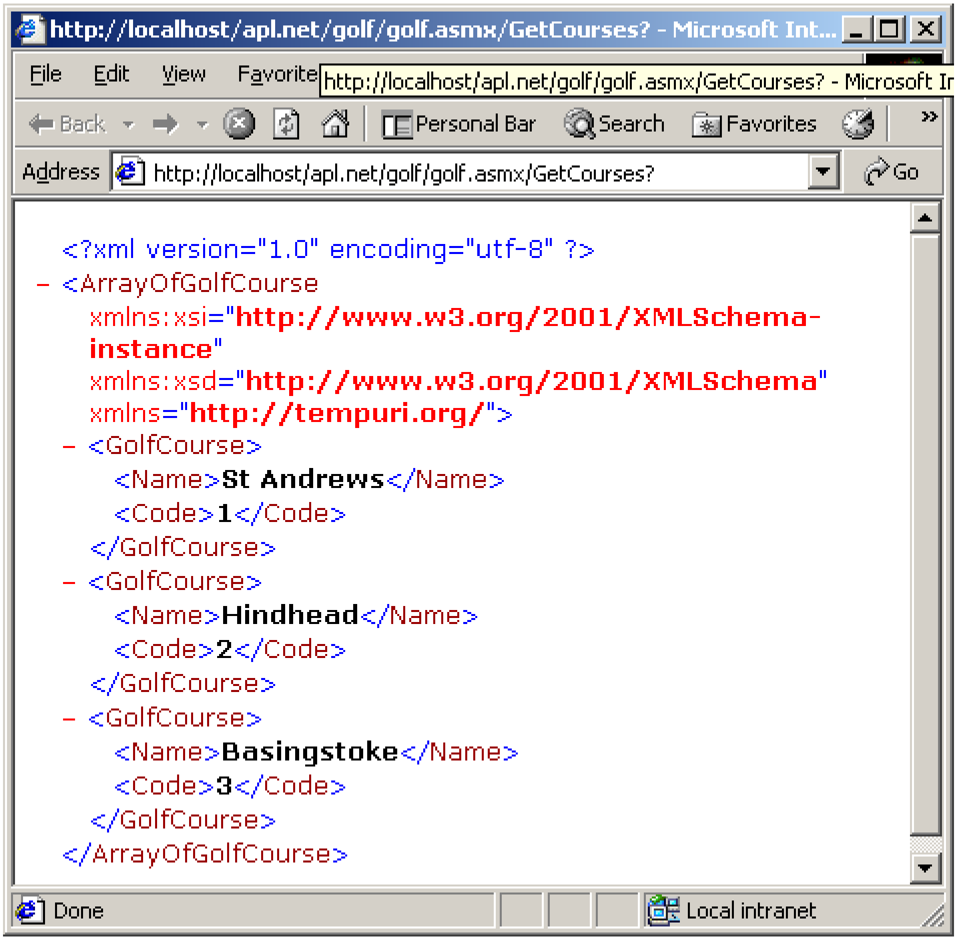
- GetStartingSheet
- GetCourses
- MakeBooking

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.

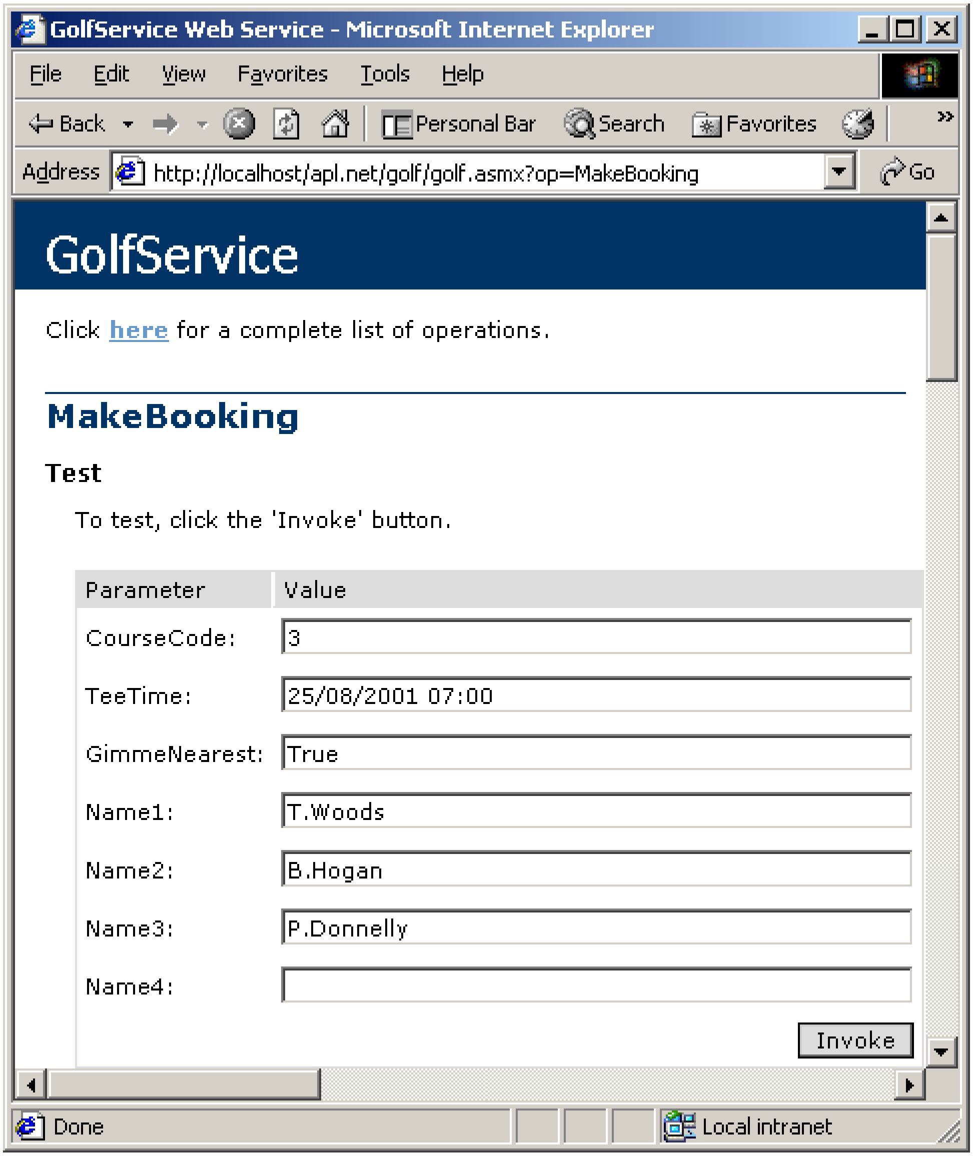

http://localhost/apl.net/golf/golf.asmx/GetCourses? - Microsoft Int...
File
Edit
View
Favorite
http://localhost/apl.net/golf/golf.asmx/GetCourses? - Microsoft Ir
Back
Personal Bar
Search
Favorites
Address
http://localhost/apl.net/golf/golf.asmx/GetCourses?
Go
Done
Local intranet

```
  <?xml version="1.0" encoding="utf-8" ?>
- <ArrayOfGolfCourse
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://tempuri.org/">
  - <GolfCourse>
      <Name>St Andrews</Name>
      <Code>1</Code>
    </GolfCourse>
  - <GolfCourse>
      <Name>Hindhead</Name>
      <Code>2</Code>
    </GolfCourse>
  - <GolfCourse>
      <Name>Basingstoke</Name>
      <Code>3</Code>
    </GolfCourse>
  </ArrayOfGolfCourse>
```

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.



http://localhost/apl.net/golf/golf.asmx?op=MakeBooking

GolfService

Click here for a complete list of operations.

MakeBooking

Test

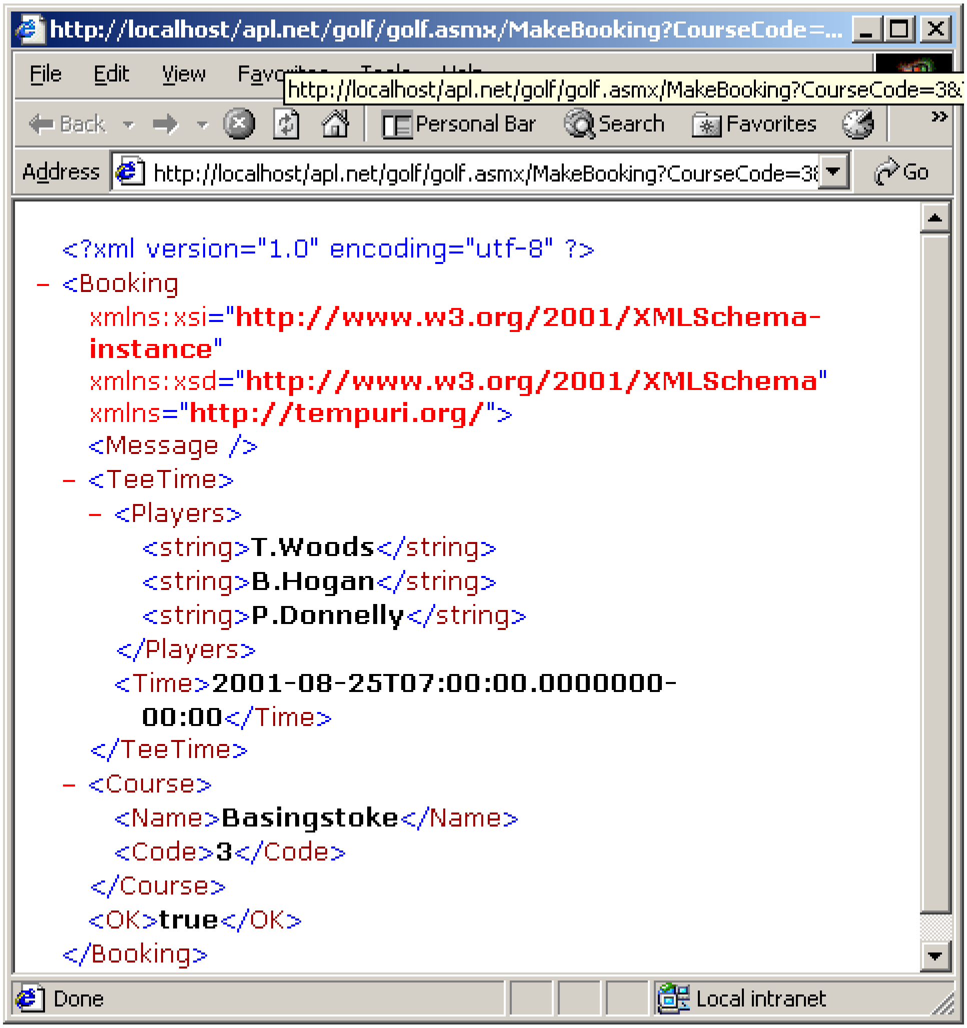
To test, click the 'Invoke' button.

Parameter	Value
CourseCode:	3
TeeTime:	25/08/2001 07:00
GimmeNearest:	True
Name1:	T.Woods
Name2:	B.Hogan
Name3:	P.Donnelly
Name4:	

Invoke

The result of invoking `MakeBooking` with this data is shown below.

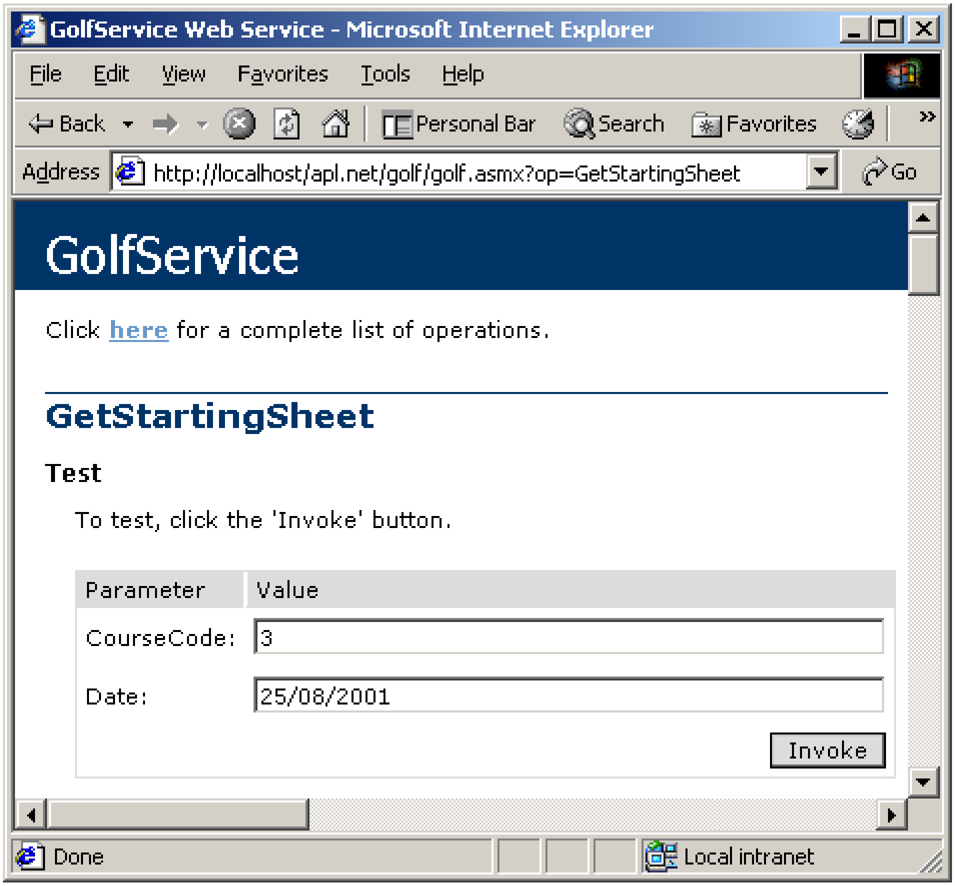
Notice how all the information about the `Booking` object structure, including the structure of the sub-objects, is provided.


http://localhost/apl.net/golf/golf.asmx/MakeBooking?CourseCode=...
File Edit View Favorites Tools Help
http://localhost/apl.net/golf/golf.asmx/MakeBooking?CourseCode=3&
Back
Personal Bar
Search
Favorites
Address
http://localhost/apl.net/golf/golf.asmx/MakeBooking?CourseCode=3
Go
Done
Local intranet

```
  <?xml version="1.0" encoding="utf-8" ?>
- <Booking
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://tempuri.org/">
    <Message />
  - <TeeTime>
    - <Players>
        <string>T.Woods</string>
        <string>B.Hogan</string>
        <string>P.Donnelly</string>
      </Players>
      <Time>2001-08-25T07:00:00.0000000-
        00:00</Time>
    </TeeTime>
  - <Course>
      <Name>Basingstoke</Name>
      <Code>3</Code>
    </Course>
    <OK>true</OK>
  </Booking>
```

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.


GolfService Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost/apl.net/golf/golf.asmx?op=GetStartingSheet
Go
Done
Local intranet

GolfService

Click here for a complete list of operations.

GetStartingSheet

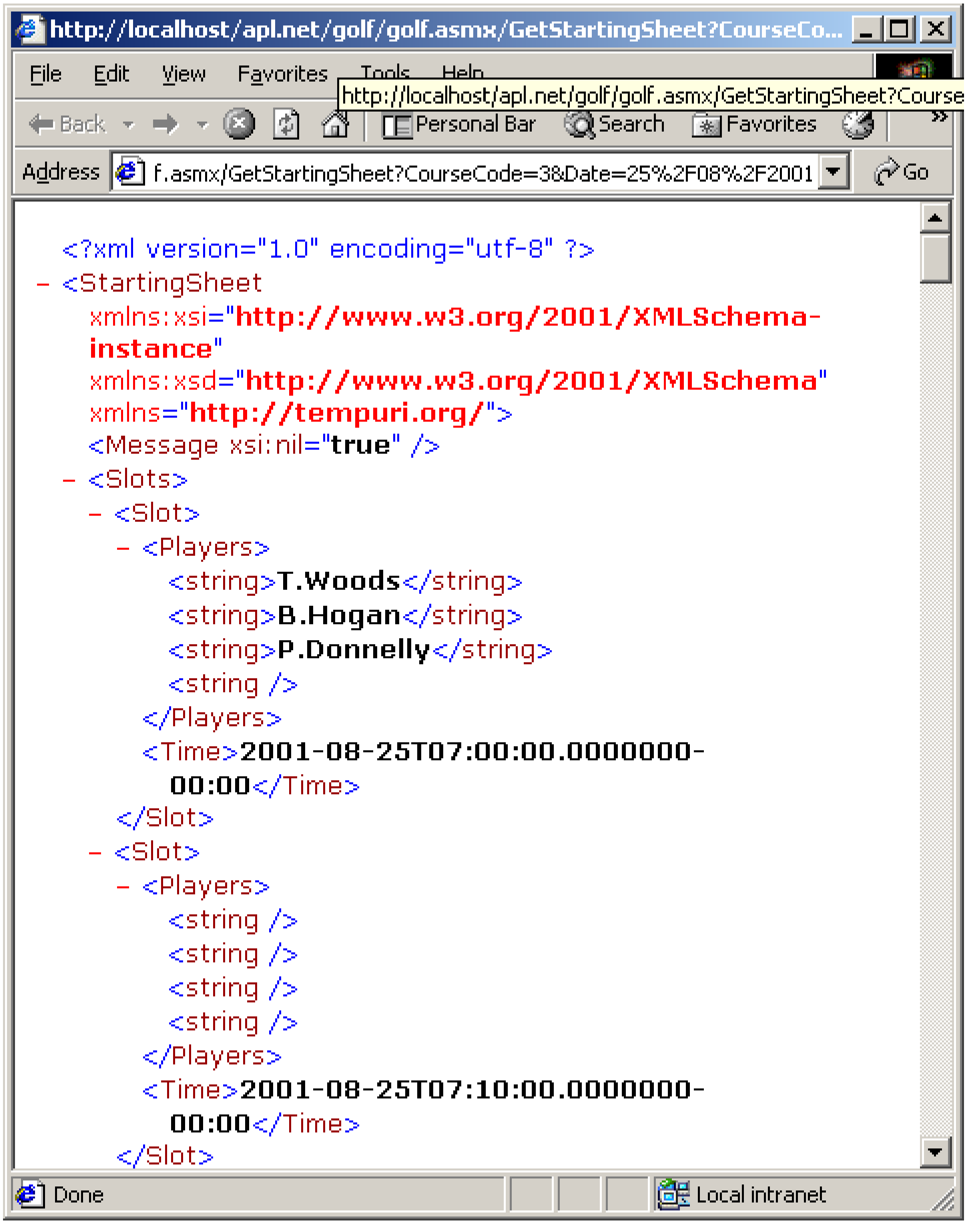
Test

To test, click the 'Invoke' button.

Parameter	Value
CourseCode:	3
Date:	25/08/2001

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.


http://localhost/apl.net/golf/golf.asmx/GetStartingSheet?CourseCo...
File
Edit
View
Favorites
Tools
Help
http://localhost/apl.net/golf/golf.asmx/GetStartingSheet?Course
Back
Personal Bar
Search
Favorites
Address
f.asmx/GetStartingSheet?CourseCode=3&Date=25%2F08%2F2001
Go
Done
Local intranet

```
  <?xml version="1.0" encoding="utf-8" ?>
- <StartingSheet
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://tempuri.org/">
    <Message xsi:nil="true" />
  - <Slots>
    - <Slot>
      - <Players>
          <string>T.Woods</string>
          <string>B.Hogan</string>
          <string>P.Donnelly</string>
          <string />
        </Players>
        <Time>2001-08-25T07:00:00.0000000-
          00:00</Time>
      </Slot>
    - <Slot>
      - <Players>
          <string />
          <string />
          <string />
          <string />
        </Players>
        <Time>2001-08-25T07:10:00.0000000-
          00:00</Time>
      </Slot>
```

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 `csharp` program `golf.exe` from a DOS prompt. Please remember to specify a reasonable date and time rather than the one used in this example.

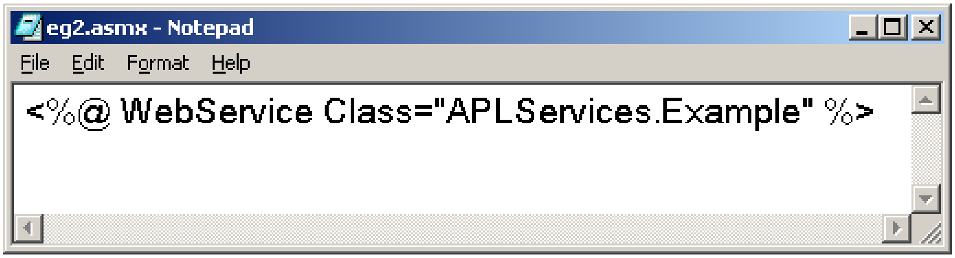
```
csharp>golf 2001-08-07T08:00:00 T.Woods A.Palmer P.Donnelly
True
25/08/2001 08:00:00
T.Woods
A.Palmer
P.Donnelly

csharp>
```

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/apl.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.



```
<%@ WebService Class="APLServices.Example" %>
```

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\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 Chapter 3. Making a WebService class is done in exactly the same way.

Starting with a *`CLEAR WS`*, we first create a namespace called *`APLServices`*. This will act as the container corresponding to a .NET Namespace in the assembly.

```
      )NS APLServices
#.APLServices
```

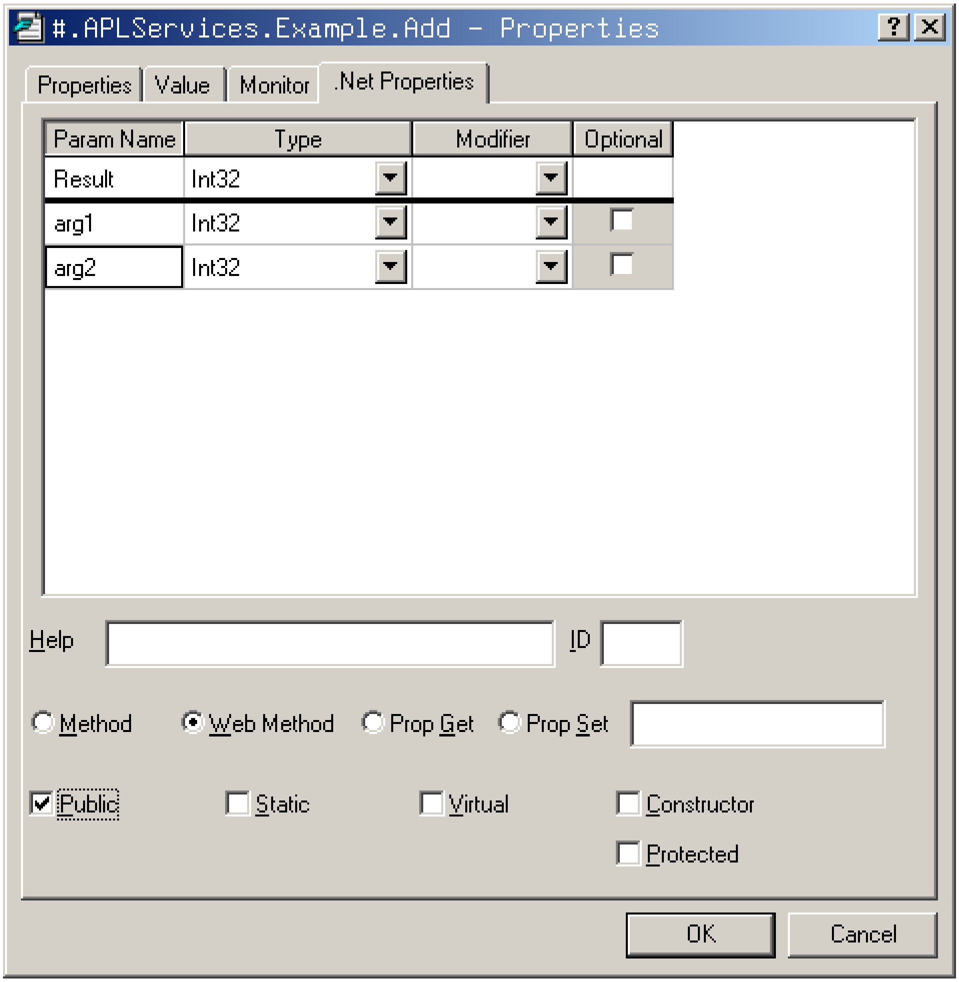
Within *`APLServices`*, we next create a *`NetType`* object called *`Example`* that inherits from **System.Web.Services.WebService**. This is the Web Service class.

```
      )CS APLServices
#.APLServices
      ⎕USING←,⊂'System'
      ⎕USING,←⊂'System.Web.Services,System.Web.Services.dll'
      'Example' ⍕WC 'NetType' 'WebService'
```

Then, within `APLServices.Example`, we can write a function called `Add` that will represent the single method to be exported by this Web Service.

```
      )CS Example
#.APLServices.Example

     ∇ R←Add arg
[1]    R←+/arg
     ∇
```

The next step is to define the *.NET Properties* for the `Add` function as shown below.



#.APLServices.Example.Add - Properties

Param Name	Type	Modifier	Optional
Result	Int32		
arg1	Int32		☐
arg2	Int32		☐

Help

ID

○ Method ● Web Method ○ Prop Get ○ Prop Set

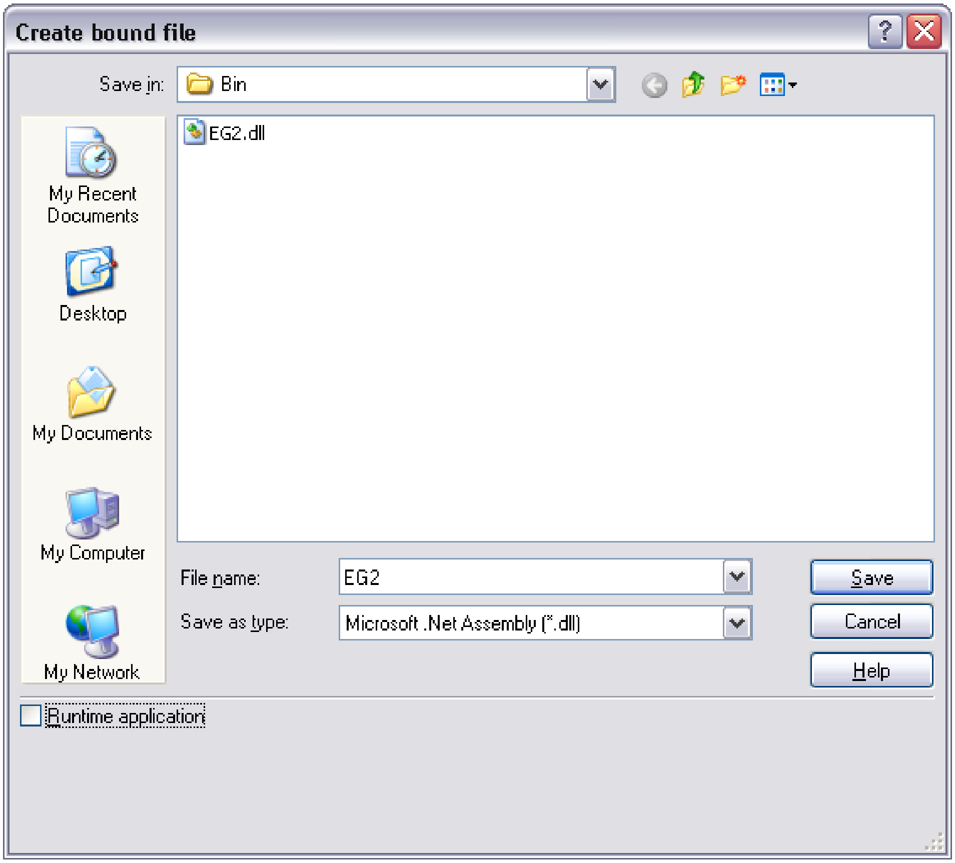
☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

It is a good idea to)SAVE the workspace, although this is not absolutely essential.

```
      )CS
#
      )WSID Samples\asp.net\webservices\Bin\EG2
was CLEAR WS
      )SAVE
Samples\asp.net\webservices\Bin\EG2 saved Tue May 22 14:...
```

Then, select the *Export…* item from the Session *File* menu, and save the assembly in `samples\asp.net\Bin`. The name of the assembly is unimportant.



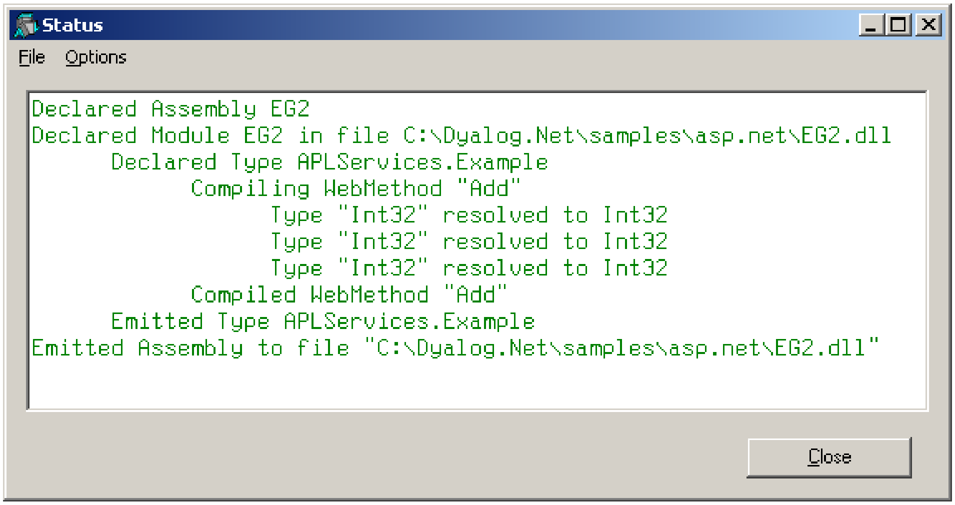
Save in: Bin

File name: EG2

Save as type: Microsoft .Net Assembly (*.dll)

☐ Runtime application

When you click *Save*, the Status Windows displays the following information to confirm that the assembly has been created correctly.

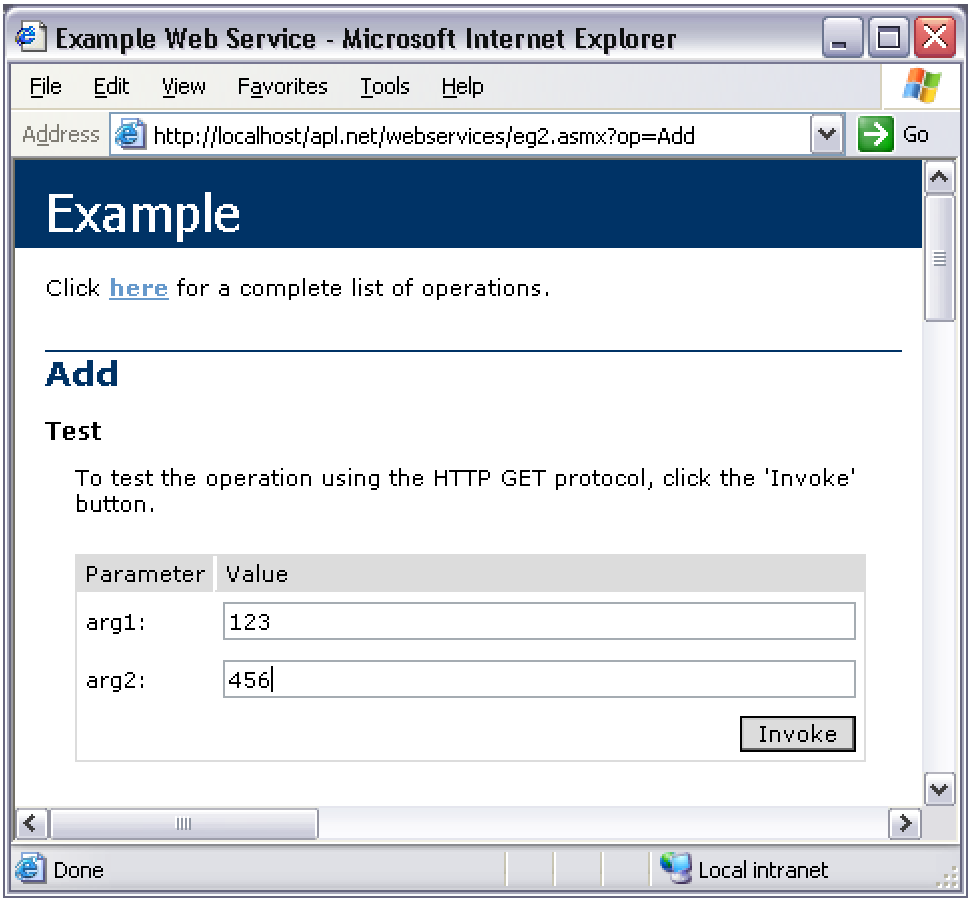


```
Declared Assembly EG2
Declared Module EG2 in file C:\Dyalog.Net\samples\asp.net\EG2.dll
      Declared Type APLServices.Example
            Compiling WebMethod "Add"
                  Type "Int32" resolved to Int32
                  Type "Int32" resolved to Int32
                  Type "Int32" resolved to Int32
            Compiled WebMethod "Add"
      Emitted Type APLServices.Example
Emitted Assembly to file "C:\Dyalog.Net\samples\asp.net\EG2.dll"
```

Testing EG2 from IE5

If you point your browser at the URL `http://localhost/apl.net/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.


Example Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Go
Invoke
Done
Local intranet

http://localhost/apl.net/webservices/eg2.asmx?op=Add

Example

Click here for a complete list of operations.

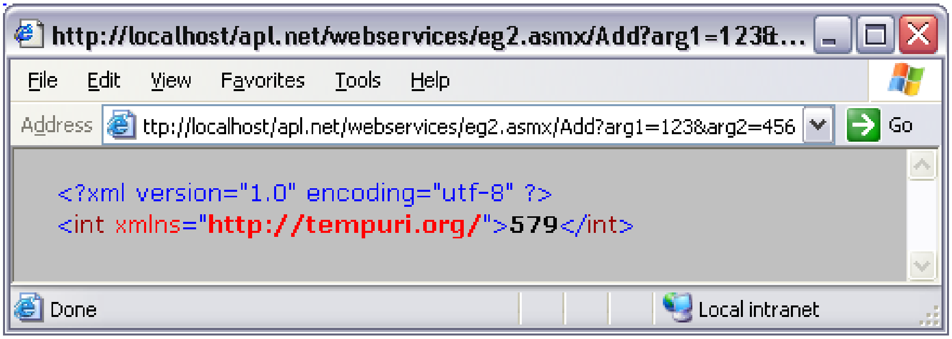
Add

Test

To test the operation using the HTTP GET protocol, click the 'Invoke' button.

Parameter	Value
arg1:	123
arg2:	456

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.


http://localhost/apl.net/webservices/eg2.asmx/Add?arg1=123&...
File Edit View Favorites Tools Help
Address
ttp://localhost/apl.net/webservices/eg2.asmx/Add?arg1=123&arg2=456
Go
Done
Local intranet

```
<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://tempuri.org/">579</int>
```

CHAPTER 7

Calling Web Services

Introduction

A Web Service communicates with its clients using TCP/IP and HTTP/XML protocols. To call a Web Service directly you need to write plumbing code to handle this communication.

Fortunately, Microsoft provides a utility called `WSDL.EXE` that creates a stub or proxy class for a Web Service. The client can then call the Web Service by creating an instance of the proxy class and calling its methods. The methods exposed by the proxy have exactly the same syntax as those of the Web Service itself.

`WSDL.EXE` actually produces the source code for the proxy class, which must then be compiled.

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/apl.net/Loan/Loan.asmx'
```

`MakeProxy` runs the Microsoft utility `WebServiceUtil.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.

Note that the paths to `WSDL.EXE` and `CSC.EXE` are hard-coded in *`MakeProxy`*, but are checked when it runs. You will have to modify this function if the hard-coded paths are wrong.

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←LoanService.New ⍬
      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, which behave like variables:

```
      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.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. Whilst it may be inconvenient that .NET Web Services does not currently support multi-dimensional arrays, it is easy to get the data into the right shape:

```
      LR.⎕ML←2
      LR.(∊⍴¨InterestRates Periods)
27 11
      ⍴LR.((∊⍴¨InterestRates Periods)⍴Payments)
27 11
      LR.((∊⍴¨InterestRates Periods)⍴Payments)
920.1345384 844.5907851 781.6836919 728.4970675 682.947 ...
```

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 several seconds; regardless of how much time it actually takes to execute on its server.

Using GolfService from Dyalog APL

The workspace `samples\asp.net\webservices\webservices` contains functions that present a GUI interface to the `GolfService` web service.

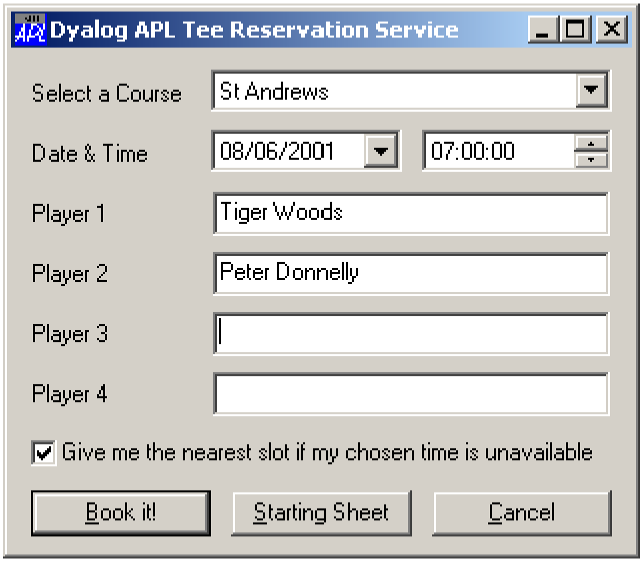
The *GOLF* function accesses `GolfService` through a proxy class. *GOLF* is called with an argument of 0 or 1. Use 1 to force GOLF to create or rebuild the proxy class, which it does by calling *MakeProxy*. You must use an argument of 1 the first time you call *GOLF*, or if you ever change the `GolfService` APL code.

Note that there is currently a problem if you attempt to use (i.e. compile) `GolfService` for the very first time via the *GOLF* function. It is currently necessary to ensure that `GolfService` is first compiled using browser access as described previously.

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/apl.net/golf/golf.asmx'
[3]    :Else
[4]        DLL←'.\GolfService.dll'
[5]    :EndIf
[6]    ⎕USING←'System'(',',DLL)
[7]    GS←GolfService.New ⍬
[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) and then presses the *Book it!* button.



Dyalog APL Tee Reservation Service

Select a Course: St Andrews

Date & Time: 08/06/2001 07:00:00

Player 1: Tiger Woods

Player 2: Peter Donnelly

Player 3:

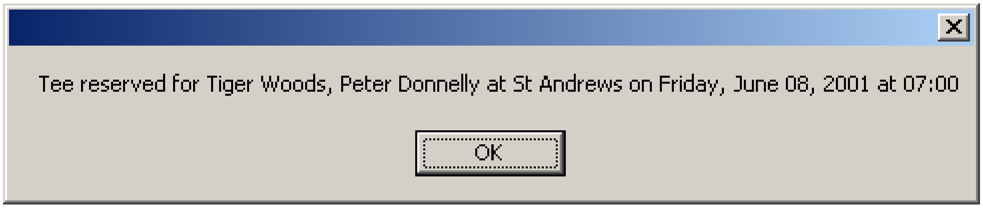
Player 4:

☑ Give me the nearest slot if my chosen time is unavailable

This action fires the `BOOK` callback function which is shown below.

```
     ∇ BOOK;CCODE;YMD;HOUR;MINUTES;FLAG;NAMES;BOOKING;M
       CCODE←⊃F.COURSE.SelItems/COURSECODES
       YMD←3↑F.DATE.(IDNToDate⊃DateTime)
       HOUR MINUTES←2↑1↓F.TIME.DateTime
       FLAG←1=F.Nearest.State
       NAMES←F.(Name1 Name2 Name3 Name4).Text
       BOOKING←GS.MakeBooking CCODE
        (DateTime.New YMD,HOUR MINUTES 0),FLAG,NAMES
       'M'⎕WC'MsgBox'
       :If BOOKING.OK
           M.Text←'Tee reserved for
             ',¯2↓⊃,/BOOKING.TeeTime.Players,¨⊂', '
           M.Text,←' at ',BOOKING.Course.Name
           M.Text,←' on ',BOOKING.TeeTime.Time.
            (ToLongDateString,' at ',ToShortTimeString)
       :Else
           M.Text←BOOKING.
            (Course.Name,' ',TeeTime.Time.(ToLongDateString,
            ' at ',ToShortTimeString),' ',Message)
       :EndIf
       ⎕DQ'M'
     ∇
```

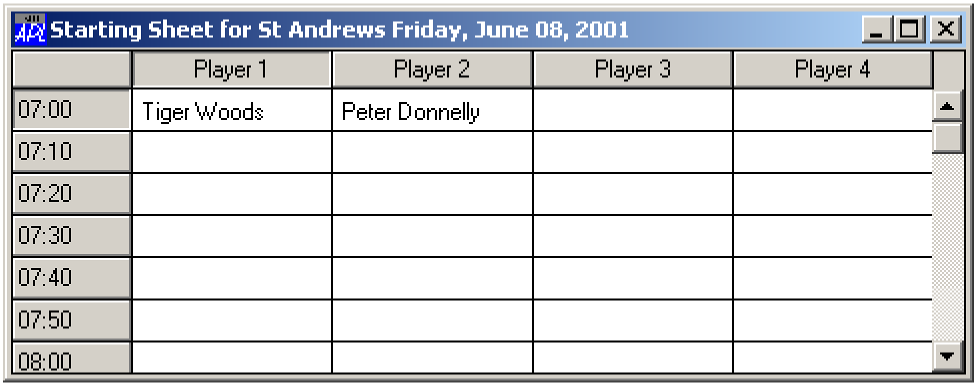
Line[6] calls the `MakeBooking` method of the *GS* object, passing it the data entered by the user. The result, a `Booking` object, is assigned to *BOOKING*. Line[8] checks its `OK` field to tell whether or not the reservation was successful. If so, lines[9-11] display the message box illustrated below. Notice how the various fields are extracted and notice how the `ToLongDateString` and `ToShortTime String` methods are employed.


Tee reserved for Tiger Woods, Peter Donnelly at St Andrews on Friday, June 08, 2001 at 07:00

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(DateTime.New 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.



Starting Sheet for St Andrews Friday, June 08, 2001

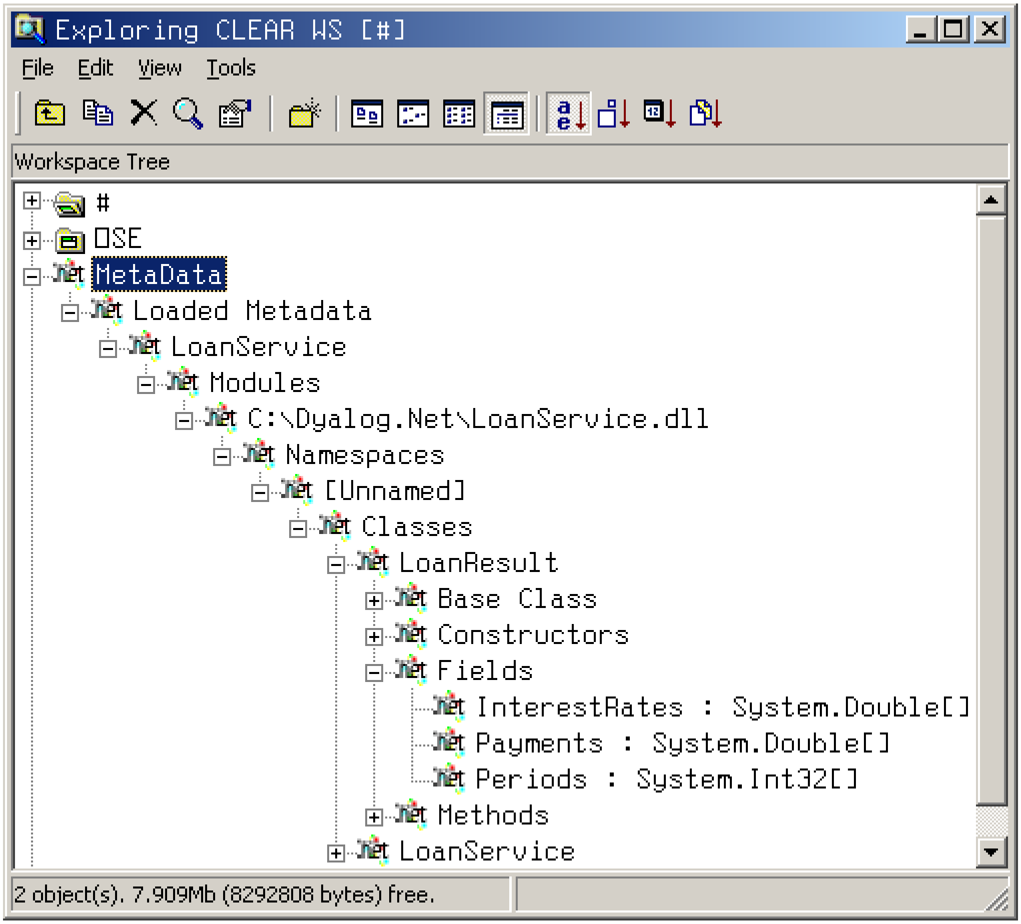
	Player 1	Player 2	Player 3	Player 4
07:00	Tiger Woods	Peter Donnelly		
07:10				
07:20				
07:30				
07:40				
07:50				
08:00				

Exploring Web Services

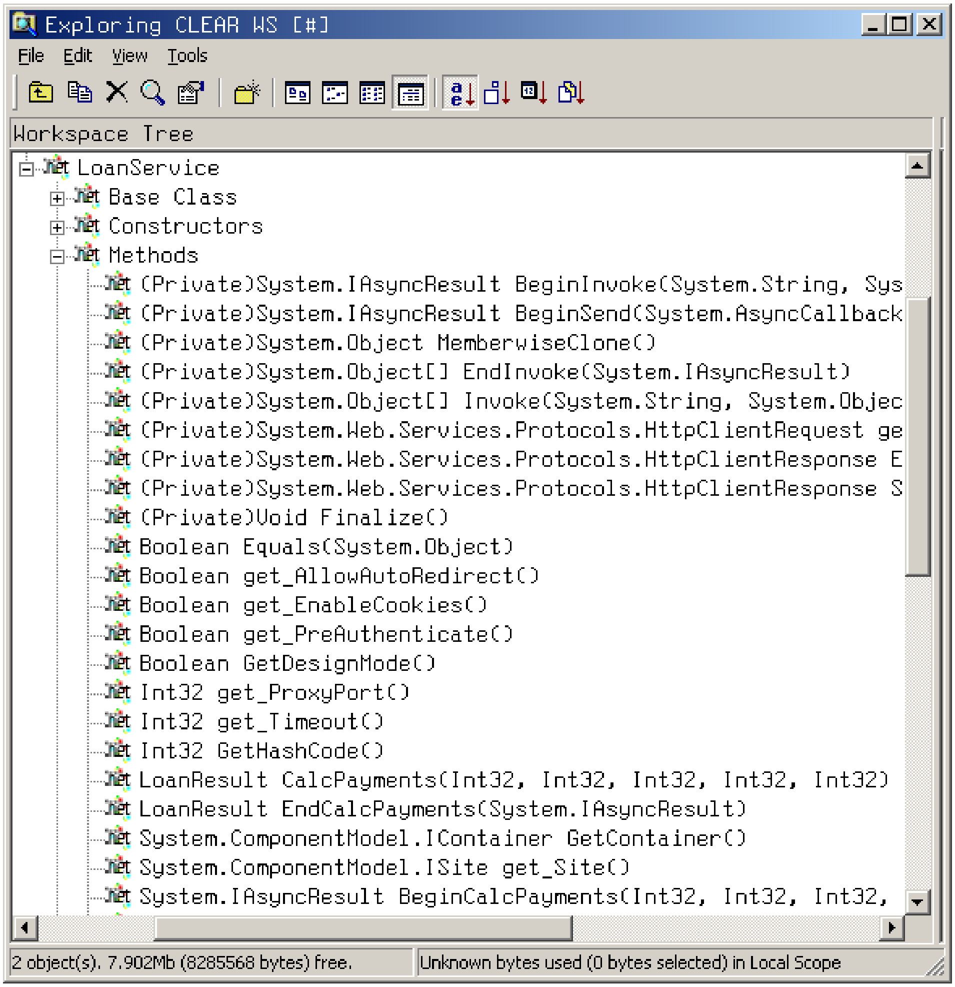
You can use the Workspace Explorer to browse the proxy class associated with a Web Service, in exactly the same way that you can browse any other .NET Assembly. The following screen shots show the *Metadata* for `LoanService`, loaded from the `LoanService.dll` proxy.

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.



```
LoanService
    Base Class
    Constructors
    Methods
        (Private)System.IAsyncResult BeginInvoke(System.String, Sys
        (Private)System.IAsyncResult BeginSend(System.AsyncCallback
        (Private)System.Object MemberwiseClone()
        (Private)System.Object[] EndInvoke(System.IAsyncResult)
        (Private)System.Object[] Invoke(System.String, System.Objec
        (Private)System.Web.Services.Protocols.HttpClientRequest ge
        (Private)System.Web.Services.Protocols.HttpClientResponse E
        (Private)System.Web.Services.Protocols.HttpClientResponse S
        (Private)Void Finalize()
        Boolean Equals(System.Object)
        Boolean get_AllowAutoRedirect()
        Boolean get_EnableCookies()
        Boolean get_PreAuthenticate()
        Boolean GetDesignMode()
        Int32 get_ProxyPort()
        Int32 get_Timeout()
        Int32 GetHashCode()
        LoanResult CalcPayments(Int32, Int32, Int32, Int32, Int32)
        LoanResult EndCalcPayments(System.IAsyncResult)
        System.ComponentModel.IContainer GetContainer()
        System.ComponentModel.ISite get_Site()
        System.IAsyncResult BeginCalcPayments(Int32, Int32, Int32,
```

2 object(s). 7.902Mb (8285568 bytes) free.

Unknown bytes used (0 bytes selected) in Local Scope

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, i.e. 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.Web.Services.Protocols.WebClientAsyncResult
```

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 1=⎕NC'MAKEPROXY'
[3]        DLL←MakeProxy'http://localhost/apl.net/loan/
                              loan.asmx'
[4]    :Else
[5]        DLL←'.\LoanService.dll'
[6]    :EndIf
[7]    ⎕USING←'System'(',',DLL)
[8]    LN←LoanService.New ⍬
[9]    AS←System.AsyncCallback.New ⎕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.

CHAPTER 8

Writing ASP.NET Web Pages

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 squirts it back. 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 `apl.net`, so you may execute the examples by specifying the URL `http://localhost/apl.net/` followed by the name of the sub-directory and page.

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.

Your first APL Web Page

The first web page example is `Intro\intro1.aspx`, which is listed below. This page displays a button whose text is reversed each time you press it.

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

∇Reverse args
:Access Public
:ParameterList Object,EventArgs

(⊃args).Text←⌽(⊃args).Text
∇

</script>

<html>
<body>
<Form runat=server>
      <asp:Button id="Pressme"
      Text="Press Me"
      runat="server"
      OnClick="Reverse"
      />
</form>
</body>
</html>
```

In this example, the page language is defined in the `<script>` section to be `"apl"`. This in turn is mapped to the `APLScript` compiler via information in the 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”

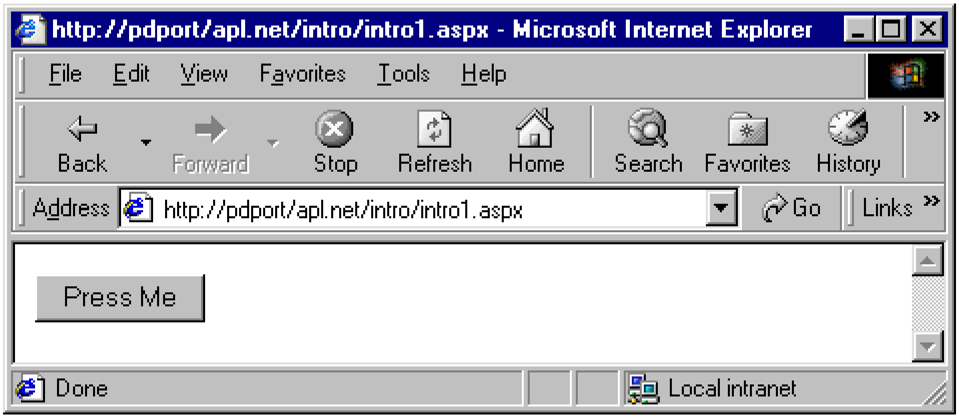
The Form and Button page elements may appear to be simple HTML, but in fact there is more to them than meets the eye and they are actually both types of ASP.NET *intrinsic controls*.

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.


http://pdport/apl.net/intro/intro1.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
Address
http://pdport/apl.net/intro/intro1.aspx
Go
Links
Press Me
Done
Local intranet

`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.

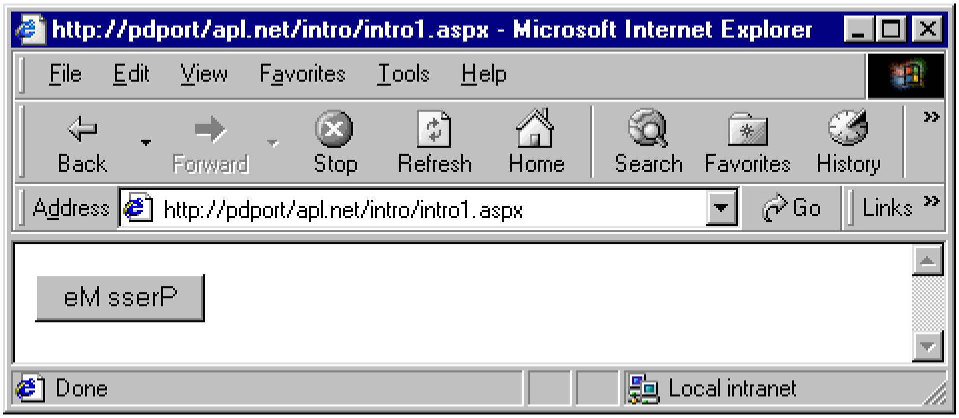
This time, because a POST back is loading the page, and because the Pressme button caused the POST, ASP.NET executes the function associated with its `onClick` attribute, namely the `APLScript` function *`Reverse`*.

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 APL callback function.

```
∇Reverse args
:Access Public
:ParameterList 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
:ParameterList 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:

```
:ParameterList 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`.

The Page_Load Event

`Intro6.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.

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

∇Page_Load
:Access Public

:if 0=IsPostBack
    list.Items.Add 'Apples'
    list.Items.Add 'Oranges'
    list.Items.Add 'Bananas'
:endif
∇
∇Select args
:Access Public
:ParameterList Object,EventArgs

out.Text←'You selected ',list.SelectedItem.Text
∇

</script>

<body>
<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>
</body>
```

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.

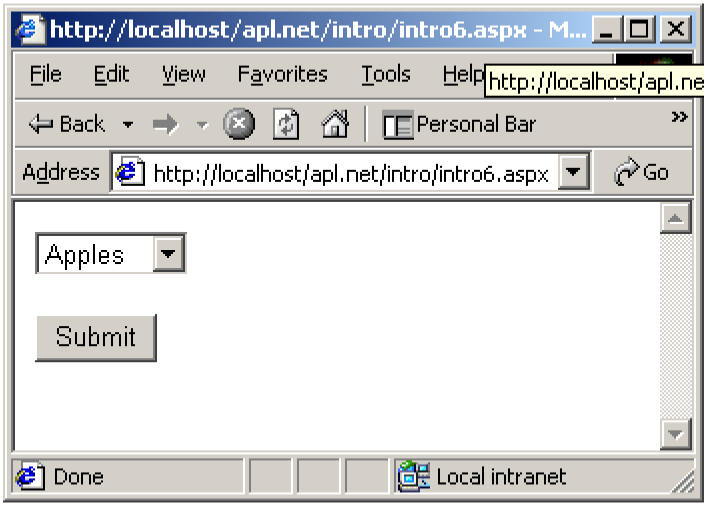
Note that, if the page employs the technique illustrated in `Intro1.aspx`, whereby the page is continually POSTed back to itself by user interaction, your `Page_Load` function will be run every time the page is loaded and you may not wish to repeat the initialisation every time. Fortunately, you can 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:

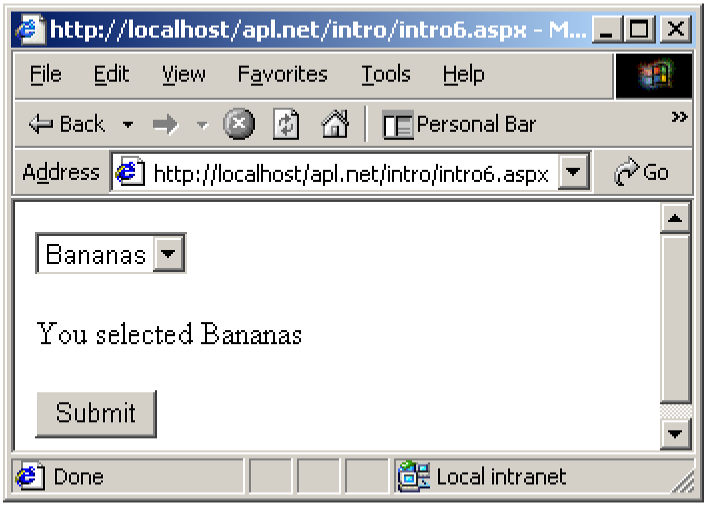
```
        list.Items.Add '...'
```

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.


http://localhost/apl.net/intro/intro6.aspx - M...
File Edit View Favorites Tools Help
Back
Personal Bar
Address http://localhost/apl.net/intro/intro6.aspx
Go
Bananas
Submit
Done
Local intranet

You selected Bananas

However, you can make this happen by adding the following attributes to the `list` object.

```
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.

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 `intro7.aspx` example illustrates this technique.

```
%@Page Language="apl" Inherits="FruitSelection"
src="fruit.apl" %>

<html>
<body>
<form runat="server" >
<asp:DropDownList id="list" runat="server"
autopostback="true"
OnSelectedIndexChanged="Select"/>
<p>
<asp:Label id=out runat="server" />
<p>
<asp:Button id="btn" Text="Pick" runat="server"
OnClick="Select" />
</form>
</body>
</html>
```

This essentially implements the same web page as `intro6a.aspx` but here *code behind* is used to separate the script implementation from the `.aspx` file.

The statement

```
%@Page Language="apl" 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 "apl" language, and its source code resides in a file called `fruit.apl`. `FruitSelection` is effectively the *base class* for the `.aspx` page.

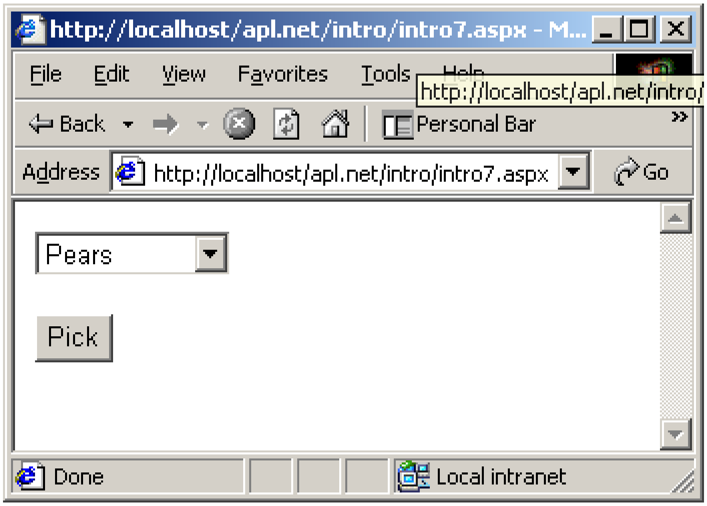
In this case, `fruit.apl` is simply another text file containing the `APLScript` code and is shown below.

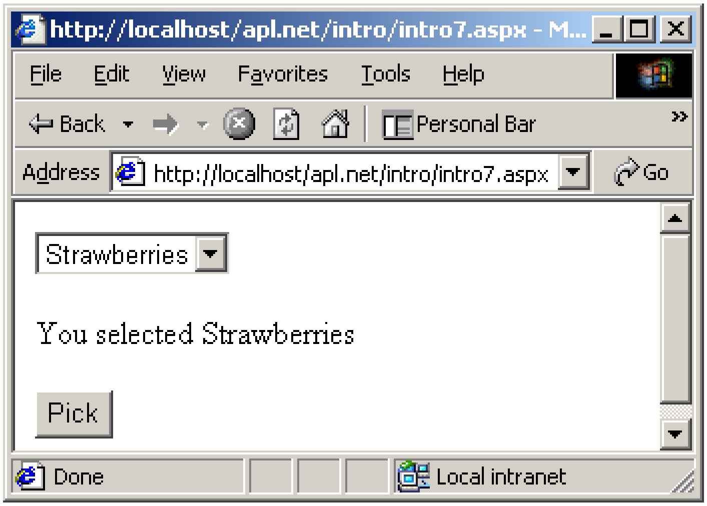
```
:class FruitSelection:System.Web.UI.Page

∇Page_Load
:Access Public

:if 0=IsPostBack
    list.Items.Add 'Pears'
    list.Items.Add 'Nectarines'
    list.Items.Add 'Strawberries'
:endif
∇

∇Select args
:Access Public
:ParameterList 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. However, this is not the case here.

The name of the class, in this case `FruitSelection`, must be the same name as is referenced in the `.aspx` web page file itself (`intro7.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.


http://localhost/apl.net/intro/intro7.aspx - M...
File
Edit
View
Favorites
Tools
Help
http://localhost/apl.net/intro/
Back
Personal Bar
Address
http://localhost/apl.net/intro/intro7.aspx
Go
Pears
Pick
Done
Local intranet


http://localhost/apl.net/intro/intro7.aspx - M...
File
Edit
View
Favorites
Tools
Help
Back
Personal Bar
Address
http://localhost/apl.net/intro/intro7.aspx
Go
Strawberries

You selected Strawberries

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.

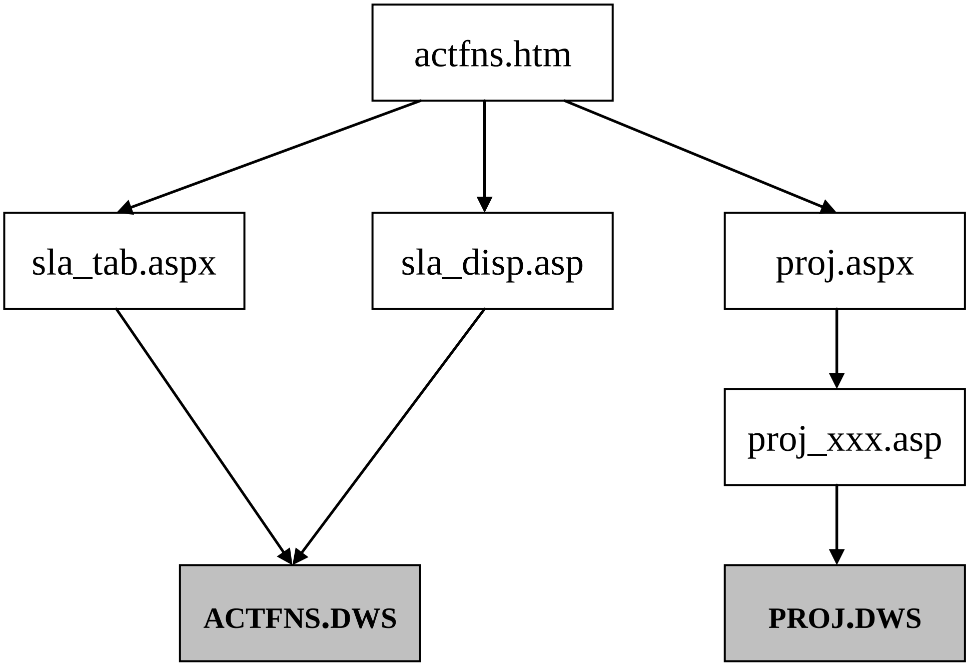
In the previous example, you saw that the `fruit.apl` file defined a new class called `FruitSelection` that inherits from `System.Web.UI.Page`. This class contains a *`Page_Load`* function that (by virtue of its name) overrides the `Page_Load` method of the underlying base class and will be called every 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 NetType object 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.

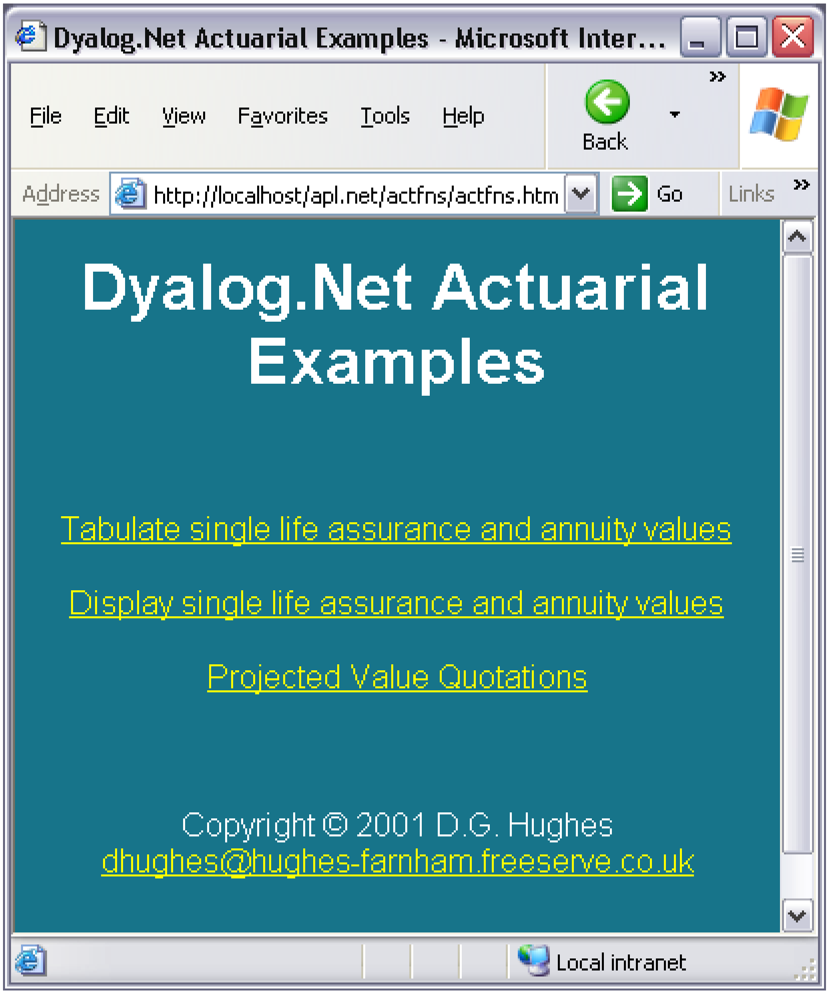
The `ACTFNS` sub-directory in `samples\asp.net` contains some examples of Dyalog APL systems that have been converted to run as Web applications using this technique.

Dyadic is grateful to David Hughes (`dhughes@hughes-farnham.freeserve.co.uk`) 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


Dyalog.Net Actuarial Examples - Microsoft Inter...
File
Edit
View
Favorites
Tools
Help
Back
Address
http://localhost/apl.net/actfns/actfns.htm
Go
Links
Local intranet

Dyalog.Net Actuarial Examples

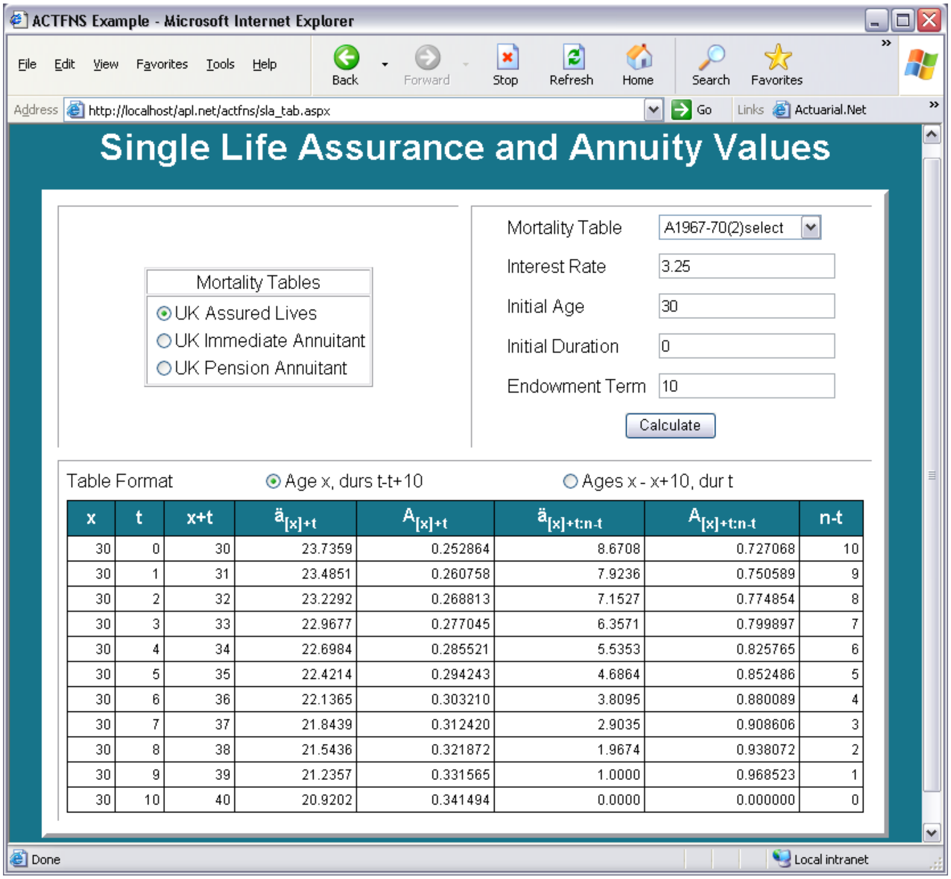
Tabulate single life assurance and annuity values

Display single life assurance and annuity values

Projected Value Quotations

Copyright © 2001 D.G. Hughes
dhughes@hughes-farnham.freeserve.co.uk

The `actfns.htm` start page offers 3 application choices


ACTFNS Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
Address
http://localhost/apl.net/actfns/sla_tab.aspx
Go
Links
Actuarial.Net
Done
Local intranet

Single Life Assurance and Annuity Values

Mortality Tables
● UK Assured Lives
○ UK Immediate Annuitant
○ UK Pension Annuitant

Mortality Table: A1967-70(2)select
Interest Rate: 3.25
Initial Age: 30
Initial Duration: 0
Endowment Term: 10

Calculate

Table Format ● Age x, durs t-t+10 ○ Ages x - x+10, dur t

x	t	x+t	$\ddot{a}_{[x]+t}$	$A_{[x]+t}$	$\ddot{a}_{[x]+t:n-t}$	$A_{[x]+t:n-t}$	n-t
30	0	30	23.7359	0.252864	8.6708	0.727068	10
30	1	31	23.4851	0.260758	7.9236	0.750589	9
30	2	32	23.2292	0.268813	7.1527	0.774854	8
30	3	33	22.9677	0.277045	6.3571	0.799897	7
30	4	34	22.6984	0.285521	5.5353	0.825765	6
30	5	35	22.4214	0.294243	4.6864	0.852486	5
30	6	36	22.1365	0.303210	3.8095	0.880089	4
30	7	37	21.8439	0.312420	2.9035	0.908606	3
30	8	38	21.5436	0.321872	1.9674	0.938072	2
30	9	39	21.2357	0.331565	1.0000	0.968523	1
30	10	40	20.9202	0.341494	0.0000	0.000000	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 `NetType` object that represents a .NET class based upon `System.Web.UI.Page`. For example, in converting the ACTFNS workspace, we started by defining `⎕USING` as follows:

   ```
         ↑⎕USING
   System
   System.Web.UI,system.web.dll
   System.Web.UI.WebControls
   System.Web.UI.HtmlControls
   System.Data,system.data.dll
   ```

   and then creating the NetType object:

   ```
         'actuarial' ⎕WC 'NetType' 'Page'
   ```

   The name you choose for this object will replace `classname` in the `Inherits="classname"` declaration in the `.aspx` web page(s) that call it.

4. Change into the newly created `NetType` object, and copy the workspace to be converted; in this case, the starting point was a workspace named `DH_ACTFNS`:

   ```
         )CS actuarial'
   #.actuarial
         )COPY DH_ACTFNS
   DHACTFNS saved ...
   ```

5. Modify the code as appropriate, inserting a `Page_Load` function and whatever callbacks functions are required.

The Page_Load function

Using its .NET Properties page, the *`Page_Load`* function must be declared as a *Public Method*. You may either call the function *`Page_Load`*, or export an arbitrary function as `Page_Load` by entering the name `Page_Load` in the edit box on this property page. Note that, either way, `Page_Load` must be spelled correctly as it is this name that causes the function to *override* 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
           ;NTY;RUN_OPTION;OPT
 ⍝ Overrides Page_Load method of Page class
 ⍝ Gets called when Page is loaded or re-loaded
   after postback
 ⍝ Initialise fields and calculate initial results 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 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←⍕TERM←10
          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 of the container NetType object 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>
```

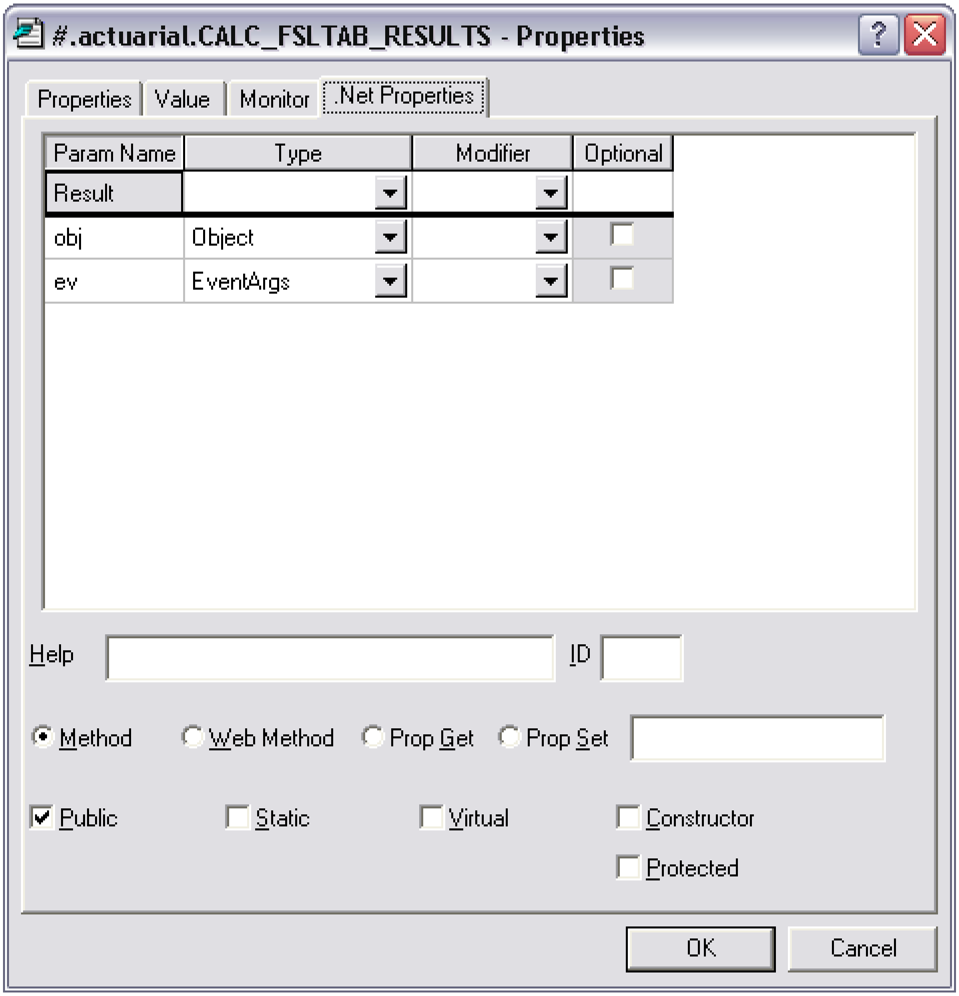
The third callback, `CHANGE_TABLES`, is called by `sla_tab.aspx` when the user selects a different set of Mortality Tables from the three provided. `CHANGE_TABLE_FORMAT` is called when the user clicks either of the two radio buttons that select how the output is to be displayed.

Like the `Page_Load` function, callback functions must be declared as being *Public Methods*. This is most easily achieved using their .NET Property page.

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 .NET property page for the function *CALC_FSLTAB_RESULTS* is shown below:



Param Name	Type	Modifier	Optional
Result			
obj	Object		☐
ev	EventArgs		☐

Help

ID

● Method ○ Web Method ○ Prop Get ○ Prop Set

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

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]    source args←MSG
[3]    :Trap 0
[4]        INT←Convert.ToDouble args.Value
[5]    :Else
[6]        args.IsValid←0
[7]        :Return
[8]    :EndTrap
[9]    args.IsValid←(0≤INT)∧20≥INT
     ∇
```

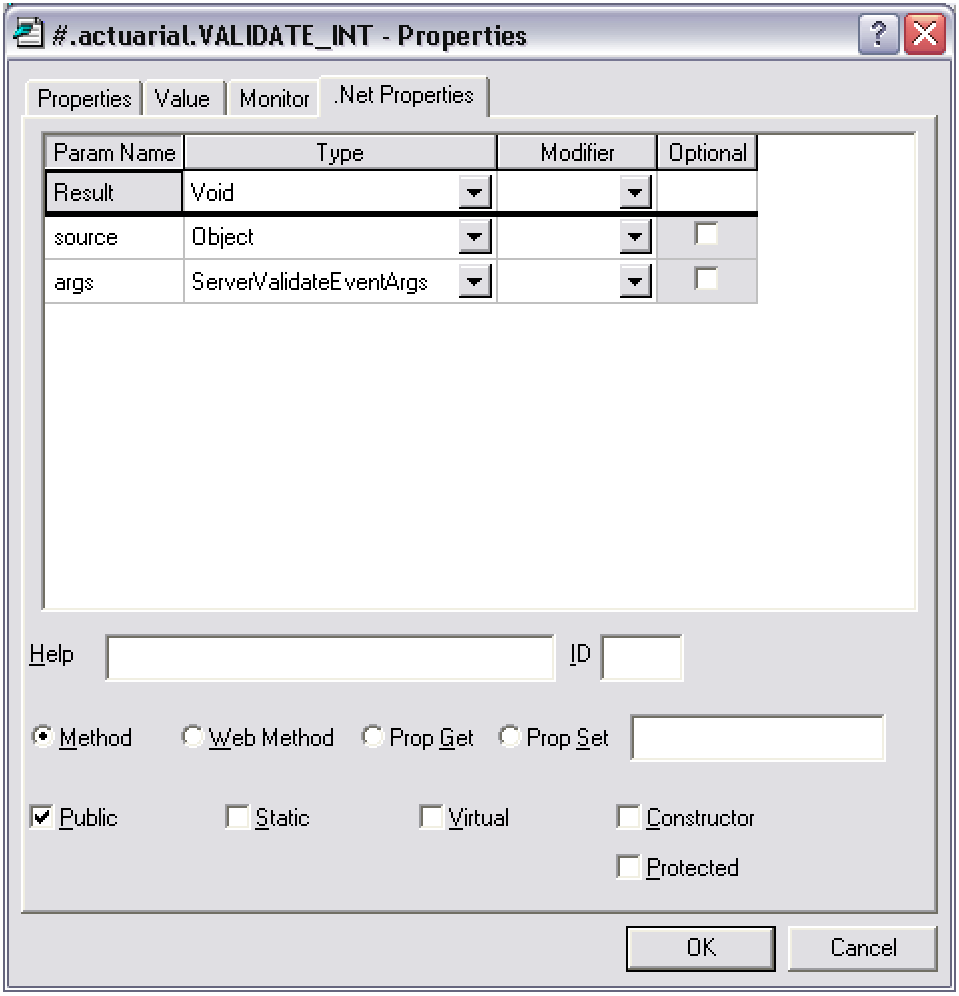
To make the *`VALIDATE_INT`* function available to the calling web page, it is exported as a method. Its *calling signature*, namely that it takes two parameters of type `System.Object` and `System.Web.UI.WebControls.ServerValidateEventArgs` respectively, identifies it as a validation function. All these factors are essential in making it recognizable and callable.

`VALIDATE_INT``[2]` 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``[4]` 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``[6 9]` 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.



Param Name	Type	Modifier	Optional
Result	Void		
source	Object		☐
args	ServerValidateEventArgs		☐

Help

ID

● Method ○ Web Method ○ Prop Get ○ Prop Set

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

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]     source args←MSG
[3]     :Trap 0
[4]         AGE←Convert.ToInt32 args.Value
[5]     :Else
[6]         args.IsValid←0
[7]         :Return
[8]     :EndTrap
[9]     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 `[8-13]` 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]     source args←MSG
[3]     :If ∧/(RFVAGE CustomValidator_AGE).IsValid
[4]         source.ErrorMessage←'Endowment Term must be an
             integer between 10 and ',(⍕90-AGE),' (90-Age)'
[5]     :Else
[6]         source.ErrorMessage←'Endowment Term must be an
             integer between 10 and (90-Age)'
[7]     :EndIf
[8]     :Trap 0
[9]         TERM←Convert.ToInt32 args.Value
[10]    :Else
[11]        args.IsValid←0
[12]        :Return
[13]    :EndTrap
[14]    :If ∧/(RFVAGE CustomValidator_AGE).IsValid
[15]        args.IsValid←(TERM≥10)∧TERM≤90-AGE
[16]    :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.

a) The page, including all the contents of its fields, is sent back to the ASP.NET server using an http POST command.

b) The postback causes the creation of a new instance of the page; which is represented by a new clone of the `actuarial` namespace.

c) 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.

d) 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.

e) The control that caused the postback raises an appropriate event, which in turn fires the associated callback function.

f) 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.

g) 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 [14] 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 [3] 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 [6] 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]     source args←MSG
[3]     :If 2=GET_RUN_OPTION
[4]         DT←1
[5]     :Else
[6]         DT←+/10 1×(TA TB).Checked
[7]     :EndIf
[8]
[9]     :If ∧/(RFVTRM CustomValidator_TERM).IsValid
[10]        source.ErrorMessage←'Initial Duration must be an
             integer between 0 and ',(⍕TERM-DT),'
                                        (TERM-',(⍕DT),')'
[11]    :Else
[12]        source.ErrorMessage←'Initial Duration must be an
             integer between 0 and (Term-',(⍕DT),')'
[13]    :EndIf
[14]    :Trap 0
[15]        DUR←Convert.ToInt32 args.Value
[16]    :Else
[17]        args.IsValid←0
[18]        :Return
[19]    :EndTrap
[20]    :If ∧/(RFVTRM CustomValidator_TERM).IsValid
[21]        args.IsValid←(0≤DUR)∧DUR≤TERM-DT
```

```
[22]    :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]    RUN_OPTION←GET_RUN_OPTION
[2]    MORT_OPTION←1+MT.SelectedIndex
[3]    OPTSMORT←MORT_OPTION⊃OPTSMORT_ASS OPTSMORT_ANNI
                            OPTSMORT_ANNP
[4]    TableNames←⊃OPTSMORT       ⍝ Assured lives/term
                                    assurance tables
[5]    TableNames←↓(2=⎕NC 0 1↓3⊃OPTSMORT)⌿TableNames
[6]    TableNames←TableNames~¨' '
[7]    CMTAB.Items.Clear
[8]    :For TableName :In TableNames
[9]        CMTAB.Items.Add TableName
[10]   :EndFor
[11]   Page.Validate ⍝ Force page validation
[12]   :Select RUN_OPTION
[13]   :Case 1
[14]       CALC_FSLTAB_RESULTS ⍬
[15]   :Case 2
[16]       CALC_FSL_RESULTS ⍬
[17]   :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←DataTable.New ⍬
[44]            cols←Columns.Add¨##.FSL_HEADER
[45]            {
[46]                row←NewRow ⍬
[47]                row.ItemArray←⍵
[48]                Rows.Add row
[49]            }¨↓##.FSLT
[50]        :EndWith
[51]        fsl.DataSource←DataView.New 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`*.

CHAPTER 9

Writing Custom Controls for ASP.NET

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, i.e. 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`.

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 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.

The SimpleCtl Control

```
      )CLEAR
clear ws
```

Starting with a `clear ws`, the first step is to make the `DyalogSamples` container namespace, and then change into it.

```
      )NS DyalogSamples
#.DyalogSamples

      )CS DyalogSamples
#.DyalogSamples
```

Next we must define `⎕USING` to include all of the .NET Namespaces that will be needed:

```
      ⎕USING←,⊂'System'
      ⎕USING,←⊂'System.Collections.Specialized,system.dll'
      ⎕USING,←⊂'System.Web,System.Web.dll'
      ⎕USING,←⊂'System.Web.UI'
      ⎕USING,←⊂'System.Web.WebControls'
      ⎕USING,←⊂'System.Web.HtmlControls'
```

Then we can build the first of the three example classes `SimpleCtl`, specifying its base class to be `Control` (actually, `System.Web.UI.Control`).

```
      'SimpleCtl' ⎕WC 'NetType' 'Control'

           )CS SimpleCtl
#.DyalogSamples.SimpleCtl
```

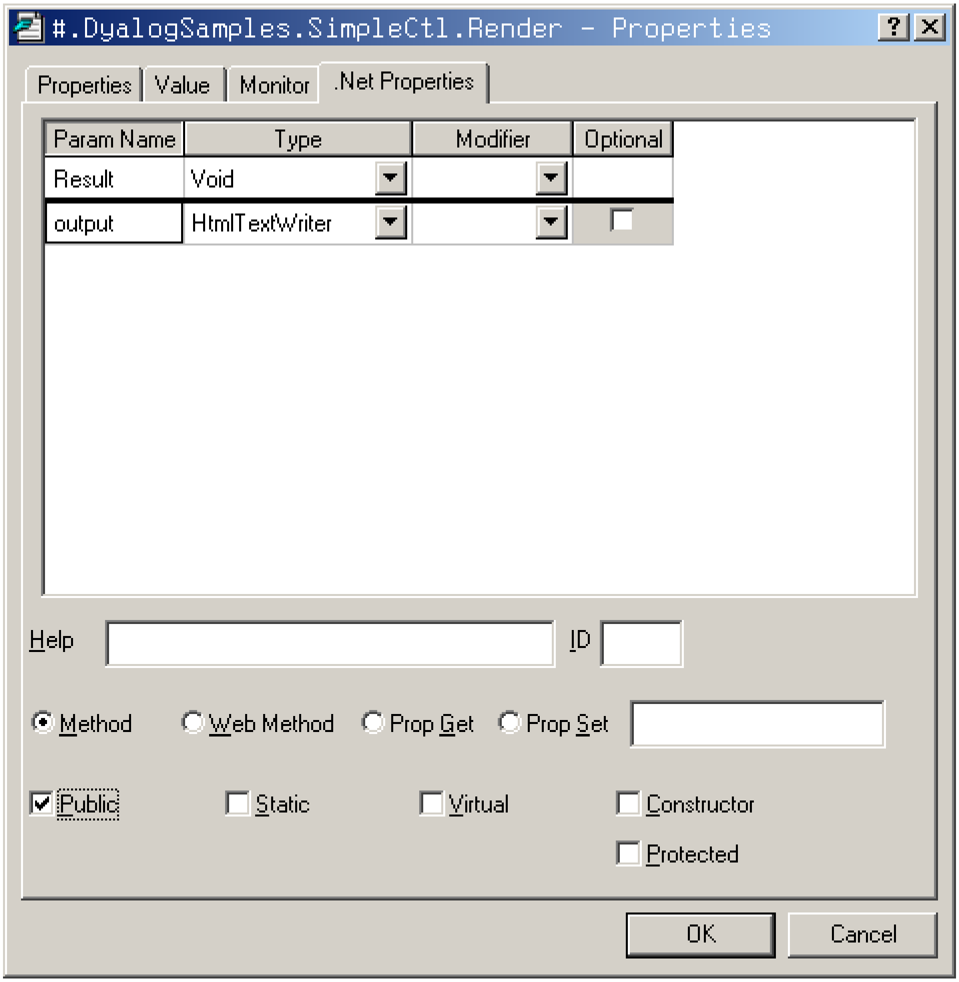
Having changed into the `SimpleCtl` namespace, we can define a function called `Render` that overrides the `Render` method that `SimpleCtl` has inherited from its base class, `System.Web.UI.Control`.

```
     ∇ Render output;HTML
[1]    HTML←'<h3>Hello World</h3>'
[2]    output.WriteLine HTML
     ∇
```

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.

The next step is to define the public interface for the `Render` function using its *.Net Properties* dialog box as shown below. The function is defined to be `void` (i.e. it does not return a result) and to take a single parameter of type `HtmlTextWriter`. Note that to successfully override the inherited method, the `Render` function must have exactly this *signature*.



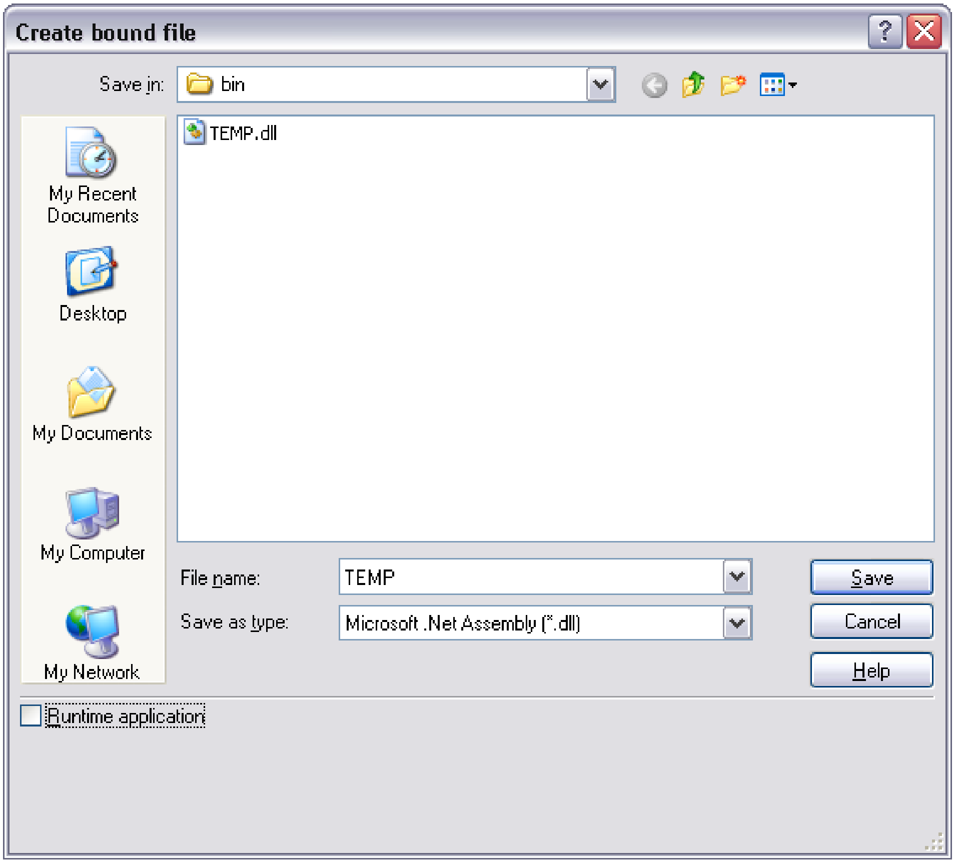
Param Name	Type	Modifier	Optional
Result	Void		
output	HtmlTextWriter		☐

● Method ○ Web Method ○ Prop Get ○ Prop Set

☑ Public ☐ Static ☐ Virtual ☐ Constructor ☐ Protected

Finally, we can save the workspace and generate the .NET Assembly. This must be located in the `Bin` subdirectory of `samples\asp.net\Temp` which itself is mapped to the IIS Virtual Directory `localhost/apl.net/Temp`.

```
      )SAVE
C:\Dyalog10\samples\ASP.NET\TEMP\BIN\TEMP saved...
```



Save in: bin

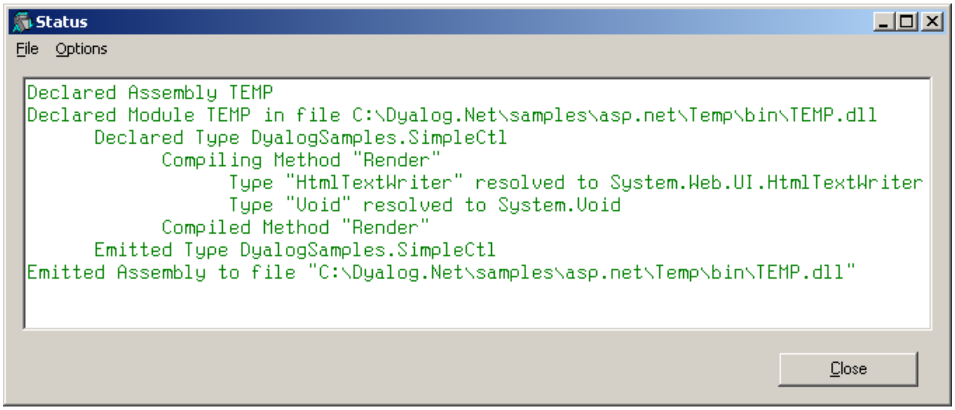
TEMP.dll

File name: TEMP

Save as type: Microsoft .Net Assembly (*.dll)

☐ Runtime application

When we select *Export…* from the *File* menu, the information displayed in the Status window confirms that the `SimpleCtl` class has been successfully emitted and saved.



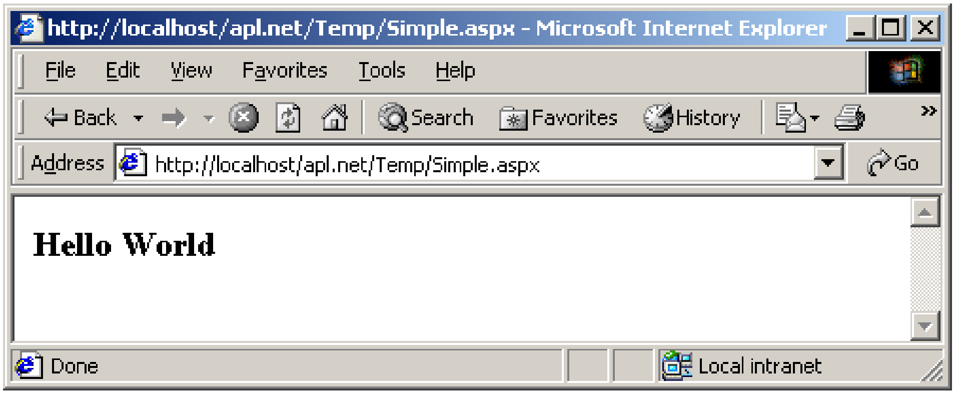
```
Declared Assembly TEMP
Declared Module TEMP in file C:\Dyalog.Net\samples\asp.net\Temp\bin\TEMP.dll
      Declared Type DyalogSamples.SimpleCtl
            Compiling Method "Render"
                  Type "HtmlTextWriter" resolved to System.Web.UI.HtmlTextWriter
                  Type "Void" resolved to System.Void
            Compiled Method "Render"
      Emitted Type DyalogSamples.SimpleCtl
Emitted Assembly to file "C:\Dyalog.Net\samples\asp.net\Temp\bin\TEMP.dll"
```

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 APL 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`. In this case, `DyalogSamples` is located by searching the Assembly TEMP.dll in the `Bin` subdirectory.



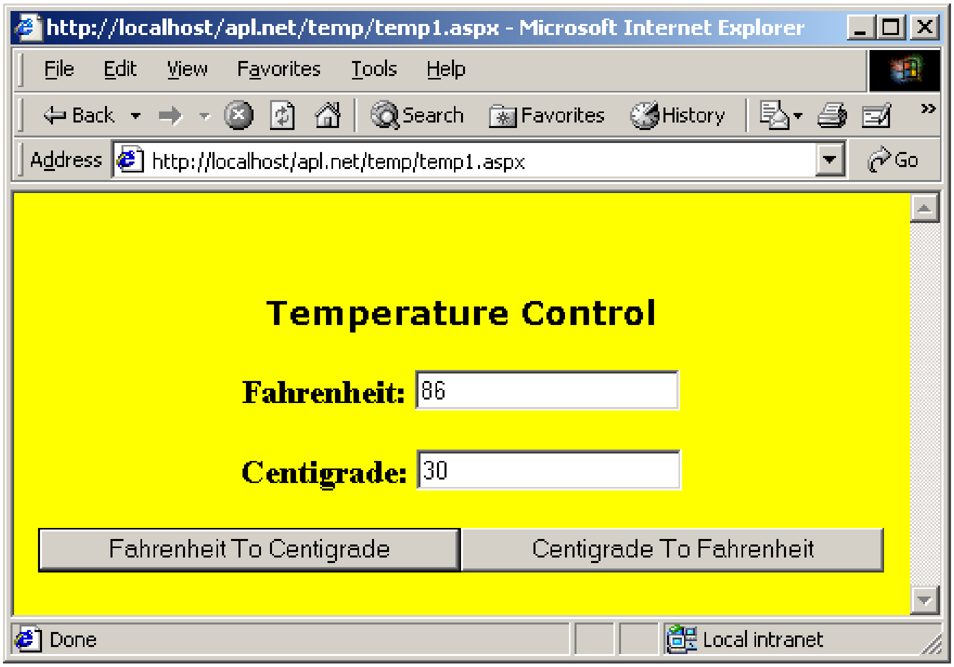
http://localhost/apl.net/Temp/Simple.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
History
Address
http://localhost/apl.net/Temp/Simple.aspx
Go
Done
Local intranet

Hello World

The TemperatureConverterCtl1 Control

The `TemperatureConverterCtl1` control is an example of a *compositional* control, i.e. 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.


http://localhost/apl.net/temp/temp1.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/apl.net/temp/temp1.aspx
Go
Fahrenheit To Centigrade
Centigrade To Fahrenheit
Done
Local intranet

Temperature Control

Fahrenheit: 86

Centigrade: 30

Starting with the TEMP workspace, the first step is to change into the *DyalogSamples* container namespace.

```
      )LOAD SAMPLES\ASP.NET\TEMP\BIN\TEMP
C:\Dyalog10\samples\ASP.NET\TEMP\BIN\TEMP saved...

      )CS DyalogSamples
#.DyalogSamples
```

The `TemperatureConverterCtl1` control is going to *contain* other standard controls as *child controls*. A control that acts as a container should implement an interface called `INamingContainer`. This interface does not in fact require any methods; it merely acts as a marker.

When we create a *`NetType`* namespace to represent the control, we need to specify that it provides this interface.

```
      'TemperatureConverterCtl1'⎕WC'NetType' 'Control'
       ('Interfaces' 'System.Web.UI.INamingContainer')

      )CS TemperatureConverterCtl1
#.DyalogSamples.TemperatureConverterCtl1
```

Child Controls

Whenever ASP.NET initialises a `Control`, it calls its `CreateChildControls` method (the default `CreateChildControls` method does nothing). So to make the appropriate child controls, we simply define a function called *`CreateChildControls`* with the appropriate public interface (no arguments and no result) as shown below.

```
     ∇ CreateChildControls

       Controls.Add LiteralControl.New'<h3>Fahrenheit: '
       m_FahrenheitTextBox←TextBox.New ⍬
       m_FahrenheitTextBox.Text←,'0'
       Controls.Add m_FahrenheitTextBox
       Controls.Add LiteralControl.New'</h3>'

       Controls.Add LiteralControl.New'<h3>Centigrade: '
       m_CentigradeTextBox←TextBox.New ⍬
       m_CentigradeTextBox.Text←,'0'
       Controls.Add m_CentigradeTextBox
       Controls.Add LiteralControl.New'</h3>'

       F2CButton←Button.New ⍬
       F2CButton.Text←'Fahrenheit To Centigrade'
       F2CButton.onClick←⎕OR'F2CConvertBtn_Click'
       Controls.Add F2CButton

       C2FButton←Button.New ⍬
       C2FButton.Text←'Centigrade To Fahrenheit'
       C2FButton.onClick←⎕OR'C2FConvertBtn_Click'
       Controls.Add C2FButton
     ∇
```

Line[2] 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[3-5] create a `TextBox` child control containing the text "0", and Line[5] adds it to the child control list.

Line[6] adds a second `LiteralControl` to terminate the "<H3>" tag.

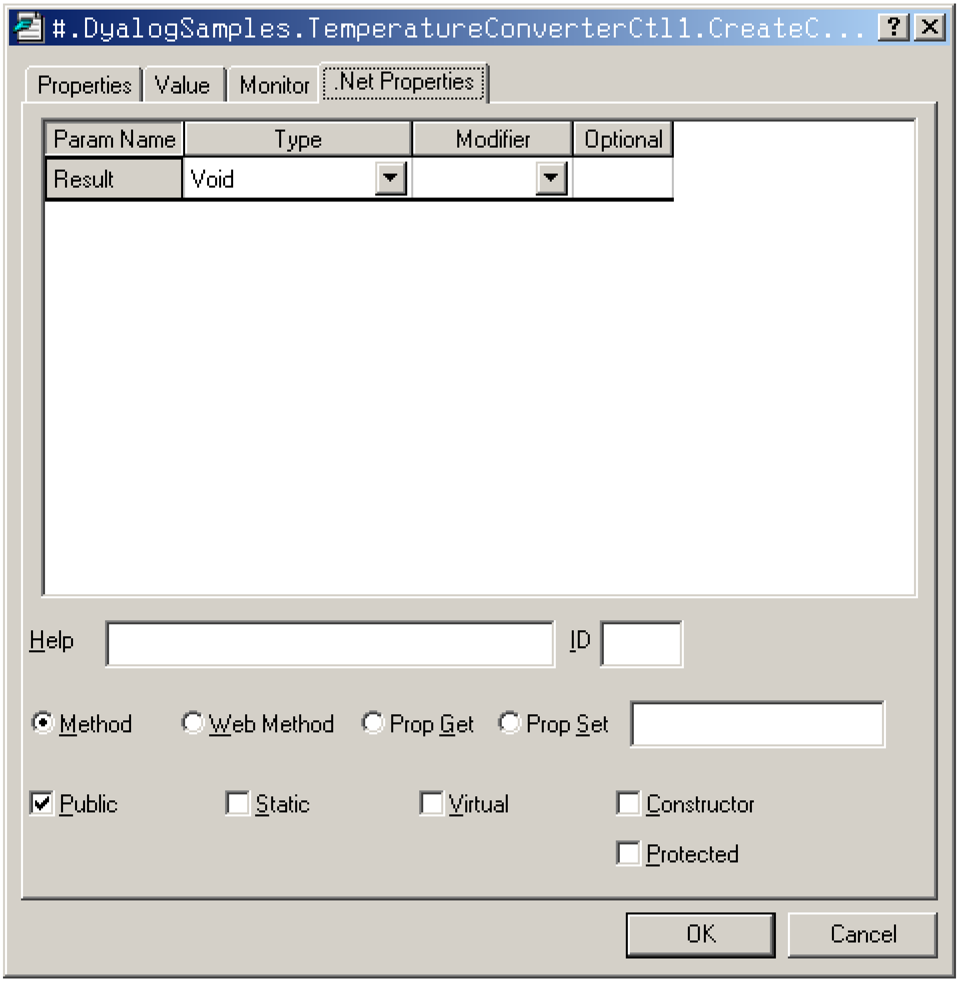
Lines [8-12] do the same for Centigrade.

Lines[14-15] create a `Button` control labelled "Fahrenheit To Centigrade". Line[16] 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[17] adds the button to the list of child controls.

Lines[19-22] 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.

The public interface for the *`CreateChildControls`* function is defined using the *.NET Properties* dialog box and is shown below.



#.DyalogSamples.TemperatureConverterCtl1.CreateC...

Param Name	Type	Modifier	Optional
Result	Void		

● Method ○ Web Method ○ Prop Get ○ Prop Set

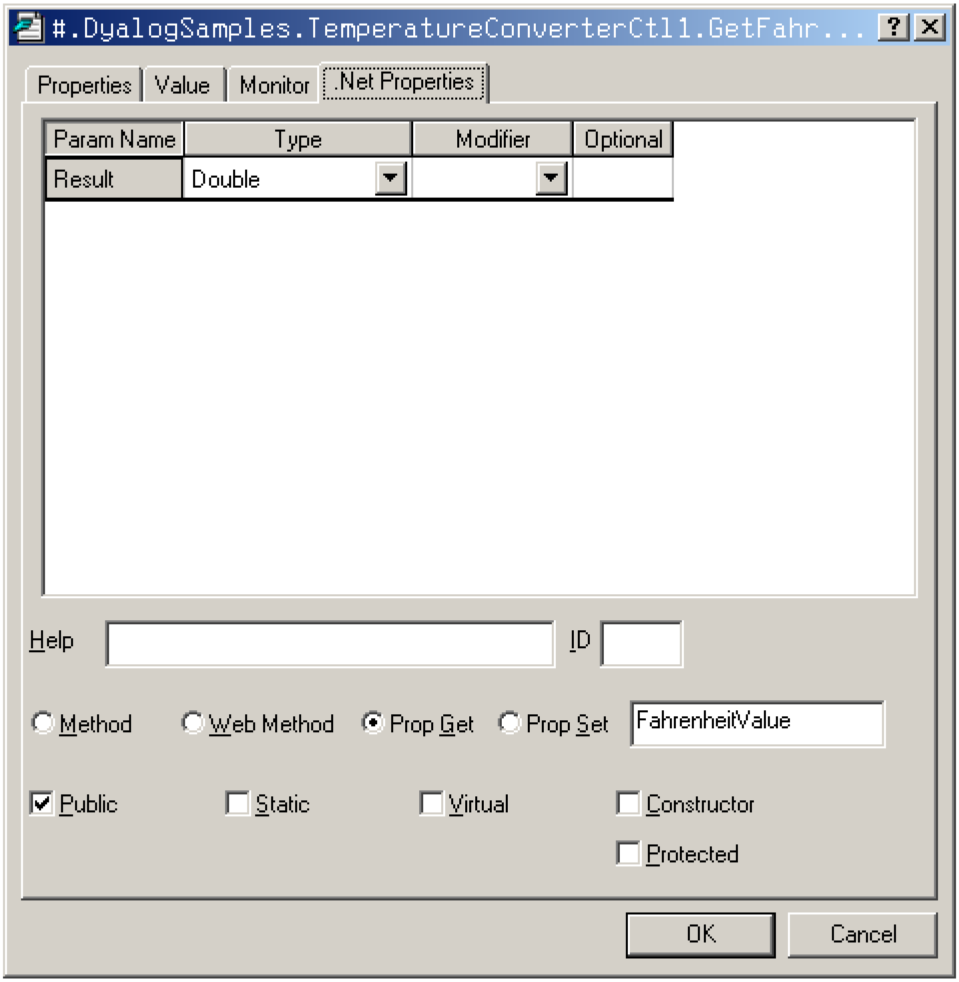
☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

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`.

```
      ∇ F←GetFahrenheitValue
[1]     F←⍎m_FahrenheitTextBox.Text
      ∇
```



Param Name	Type	Modifier	Optional
Result	Double		

○ Method ○ Web Method ● Prop Get ○ Prop Set FahrenheitValue

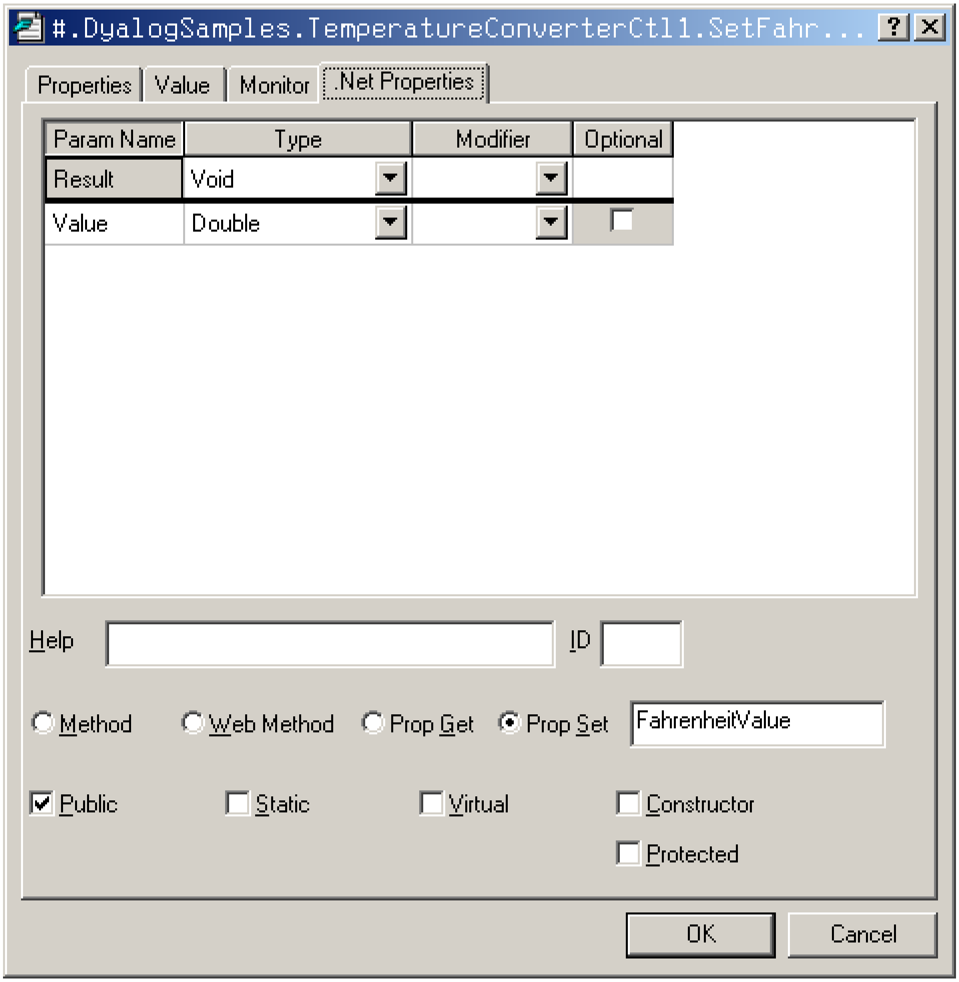
☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

Notice that the `GetFahrenheitValue` 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

The corresponding *Set* function is:

```
     ∇ SetFahrenheitValue F
[1]    m_FahrenheitTextBox.Text←⍕F
     ∇
```



#.DyalogSamples.TemperatureConverterCtl1.SetFahr...

Param Name	Type	Modifier	Optional
Result	Void		
Value	Double		☐

Help

ID

○ Method ○ Web Method ○ Prop Get ● Prop Set FahrenheitValue

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

Similar functions to handle the `Centigrade` 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 using `GetCentigradeValue`, converts it to Fahrenheit using `C2F`, and then sets the `FahrenheitValue` property using `SetFahrenheitValue`.

```
     ∇ C2FConvertBtn_Click args
[1]    SetFahrenheitValue C2F GetCentigradeValue
     ∇


     ∇ f←C2F c
[1]    f←32+c×1.8
     ∇


     ∇ F2CConvertBtn_Click args
[1]    SetCentigradeValue F2C GetFahrenheitValue
     ∇


     ∇ c←F2C f
[1]    c←(f-32)÷1.8
     ∇
```

These functions are all 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>
<h3><font face="Verdana" color="black">Temperature
Control</font></h3>

<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>
<h3><font face="Verdana" color="black">Temperature
Control</font></h3>

<form name="ctrl1" method="post" action="temp1.aspx"
id="ctrl1">
<input type="hidden" name="__VIEWSTATE"
value="YTB6MTc3MzAxNzYxNF9fX3g=03f01d88" />

<h3>Fahrenheit: <input name="TempCvtCtl1:ctrl1" type="text"
value="32" /></h3><h3>Centigrade: <input
name="TempCvtCtl1:ctrl4" type="text" value="0" /></h3><input
type="submit" name="TempCvtCtl1:ctrl6" value="Fahrenheit To
Centigrade" /><input type="submit" name="TempCvtCtl1:ctrl7"
value="Centigrade To Fahrenheit" />
</form>

</center>
</body>
</html>
```

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.

Starting with the TEMP workspace, the first step is to change into the `DyalogSamples` container namespace.

```
      )LOAD SAMPLES\ASP.NET\TEMP\BIN\TEMP
C:\Dyalog10\samples\ASP.NET\TEMP\BIN\TEMP saved...

      )CS DyalogSamples
#.DyalogSamples
```

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.

When we create a `NetType` namespace to represent the control, we need to specify that it provides these interfaces.

```
      'TemperatureConverterCtl2'⎕WC'NetType' 'Control'
       ('Interfaces' 'System.Web.UI.IPostBackDataHandler,
                      System.Web.UI.IPostBackEventHandler')

      )CS TemperatureConverterCtl2
#.DyalogSamples.TemperatureConverterCtl2
```

Fahrenheit and Centigrade Values

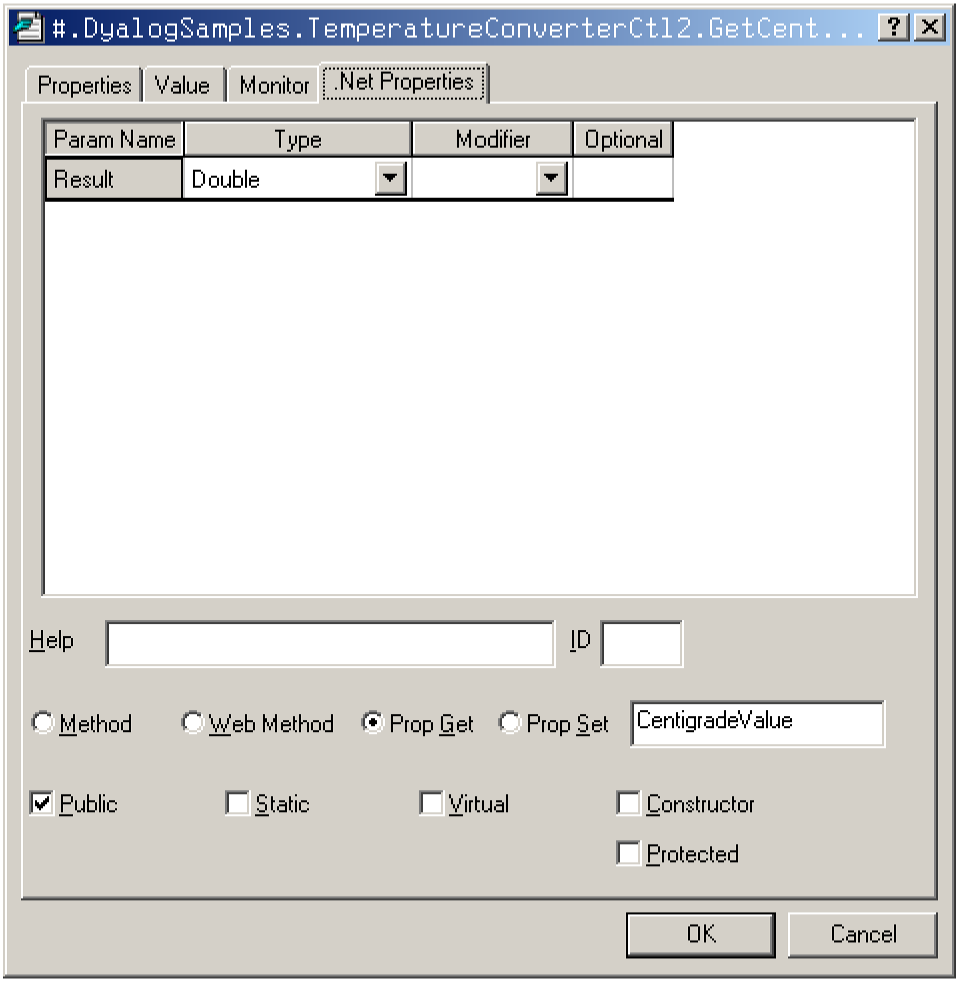
Like the previous `TemperatureConverterCtl2` control, the `TemperatureConverterCtl2` maintains two public properties named `CentigradeValue` and `FahrenheitValue` using *property get* and *property set* functions.

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 `GetCentigradeValue` 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←GetCentigradeValue
[1]     C←_CentigradeValue
      ∇
```



Param Name	Type	Modifier	Optional
Result	Double		

Help

ID

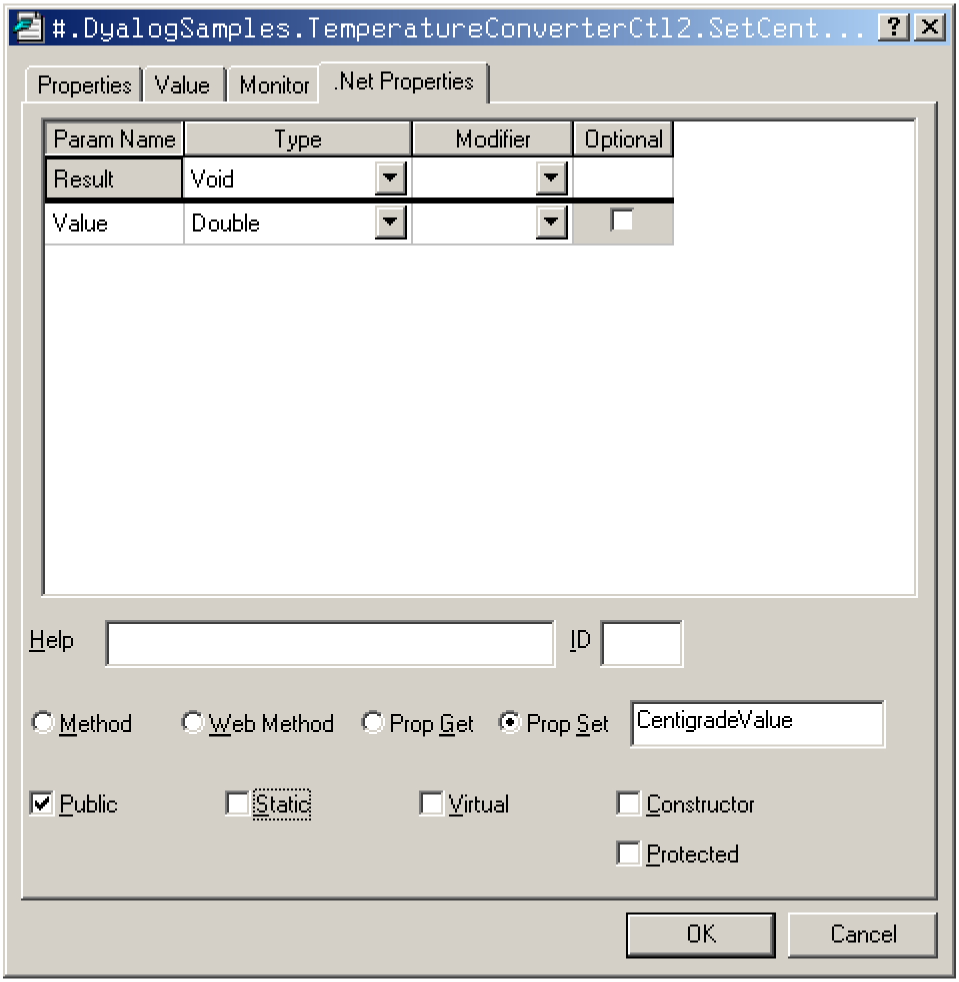
○ Method ○ Web Method ● Prop Get ○ Prop Set CentigradeValue

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

The `SetCentigradeValue` 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`.

```
      ∇ SetCentigradeValue C
[1]     _CentigradeValue←C
      ∇
```



#.DyalogSamples.TemperatureConverterCtl2.SetCent...

Param Name	Type	Modifier	Optional
Result	Void		
Value	Double		☐

Help

ID

○ Method ○ Web Method ○ Prop Get ● Prop Set CentigradeValue

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

The *property get* and *property set* functions for the `FahrenheitValue` property are similarly defined. The .NET Properties for these functions are similar to those for the `CentigradeValue` functions and are not shown.

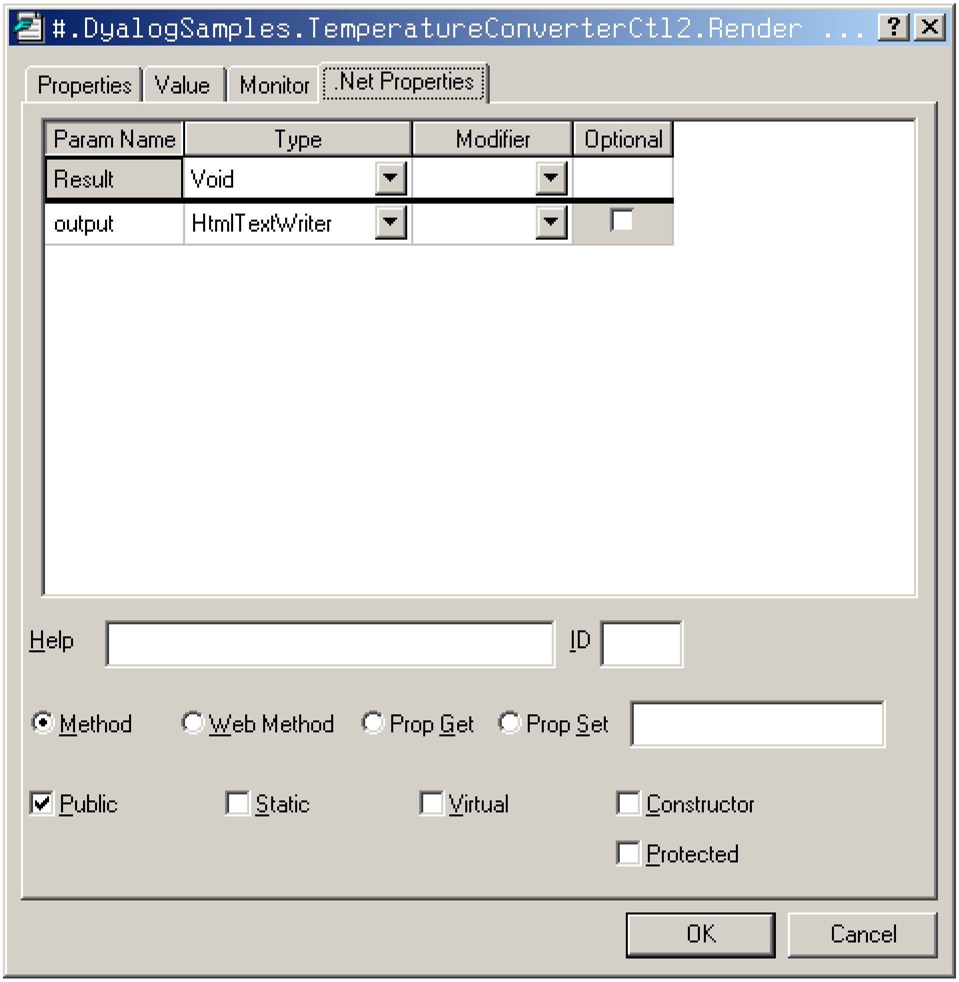
```
      ∇ F←GetFahrenheitValue
        F←_FahrenheitValue
      ∇

      ∇ SetFahrenheitValue F
        _FahrenheitValue←F
      ∇
```

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

        F←'<h3>Fahrenheit <input name='
        F,←UniqueID
        F,←' id=FahrenheitValue type=text value='
        F,←⍕_FahrenheitValue
        F,←'></h3>'
        output.Write F

        C←'<h3>Centigrade <input name='
        C,←UniqueID
        C,←' id=CentigradeValueKey type=text value='
        C,←⍕_CentigradeValue
        C,←'></h3>'
        output.Write C

        BF←'<input type=button value=FahrenheitToCentigrade
            '
        BF,←' onClick="jscript:'
        BF,←Page.GetPostBackEventReference
            this'FahrenheitToCentigrade'
        BF,←'">'
        output.Write BF

        CF←'<input type=button value=CentigradeToFahrenheit
            '
        CF,←' onClick="jscript:'
        CF,←Page.GetPostBackEventReference
            this'CentigradeToFahrenheit'
        CF,←'">'
        output.Write CF
```

```
[27]
[28]   output.WriteLine¨'' '<br>' '<br>'
     ∇
```



#.DyalogSamples.TemperatureConverterCtl2.Render ...

Param Name	Type	Modifier	Optional
Result	Void		
output	HtmlTextWriter		☐

● Method ○ Web Method ○ Prop Get ○ Prop Set

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

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[2-6] and lines [9-13] generate HTML that defines two text boxes in which the user may enter the Fahrenheit and centigrade values respectively. Lines[7 14] use the `Write` method of the `HtmlTextWriter` object to output the HTML.

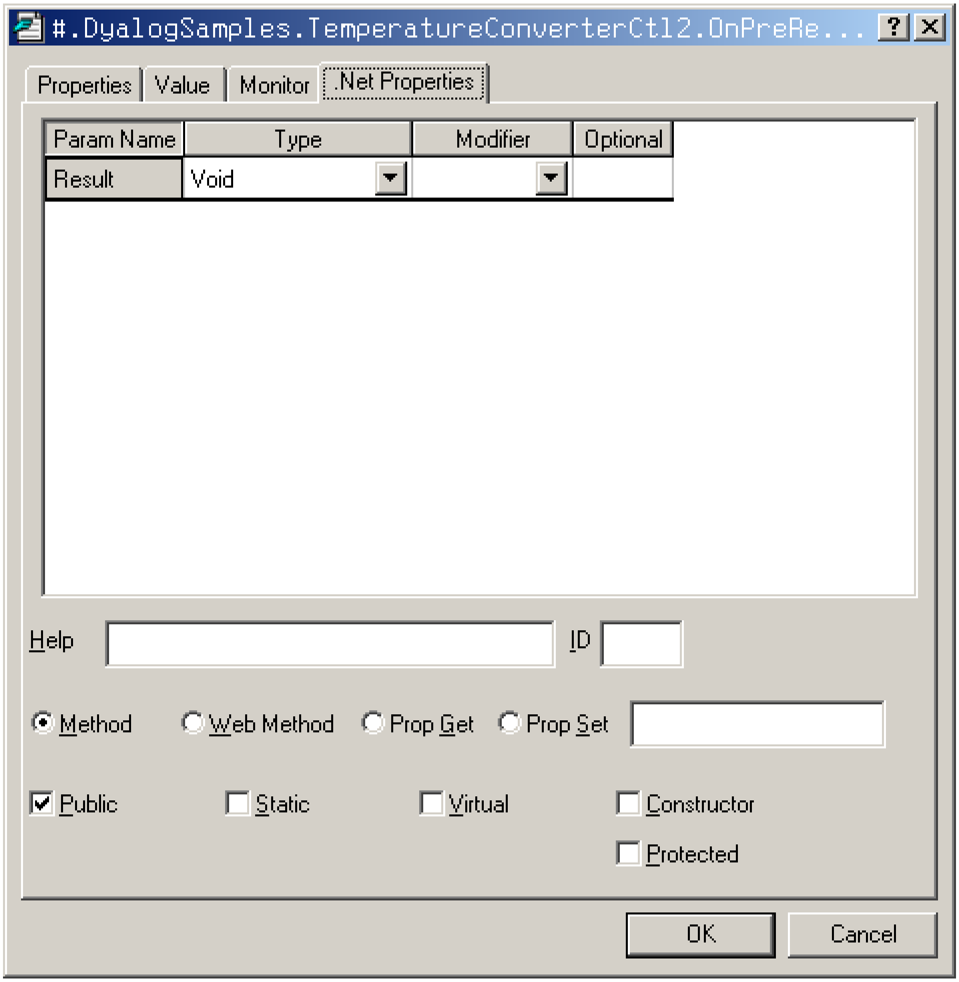
Lines[3 10] obtain the fully qualified identifier for this particular instance of the `TemperatureConverterCtl2` control from its `UniqueID` property. This is a property, which it inherits from `Control` and is therefore also a property of the current (APL) namespace

Lines[16-20] and Lines[22-26] generate and output the HTML to represent the two buttons that convert from Fahrenheit to Centigrade and from Centigrade to Fahrenheit respectively.

Lines[18 24] 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, i.e. send the page contents back to the server. `GetPostBackEventReference` is a (static) 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 (i.e. which button was pressed). The first parameter is a namespace reference to the current space which is generated by the function *`this`*.

The client-side script is itself generated, and inserted into the HTML stream, by calling the `RegisterPostBackScript` method. This is done by the *`OnPreRender`* function, which overrides the `OnPreRender` method of the `Control` class upon which `TemperatureConverterCtl2` is based. This method is called by ASP.NET before it calls `Render`.

```
     ∇ OnPreRender
[1]    Page.RegisterPostBackScript
     ∇
```



#.DyalogSamples.TemperatureConverterCtl2.OnPreRe...

Param Name	Type	Modifier	Optional
Result	Void		

Help

ID

● Method ○ Web Method ○ Prop Get ○ Prop Set

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

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

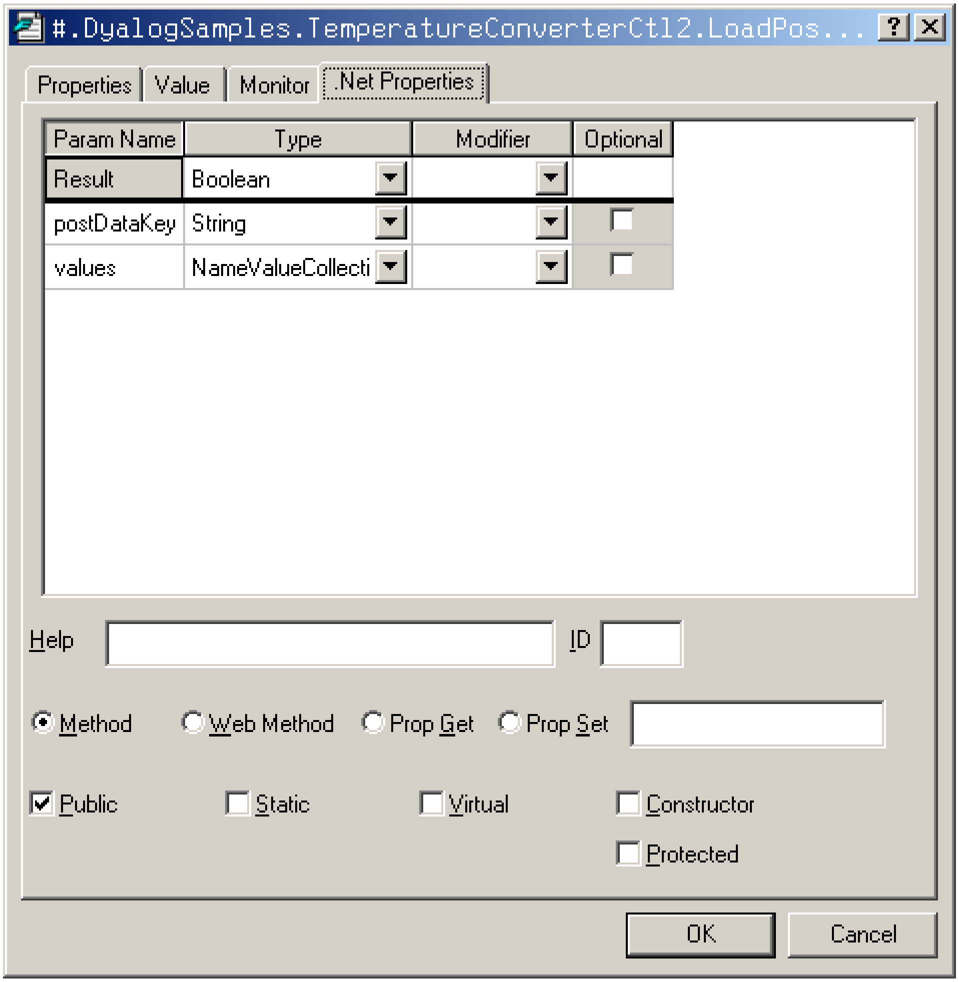
Once the server-side control has rendered the HTML for the browser, the user is free to type numbers into the text boxes and to press the buttons.

When the user presses a button, the browser runs the client-side JavaScript code (that was inserted by the *`OnPreRender`* function) that in turn generates a postback to the server.

When we created `TemperatureConverterCtl2` with *`⎕WC`*, we specified that it supported the `IPostBackDataHandler` interface. This interface must be implemented by controls that want to receive postback data (i.e., 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]     postDataKey values←args
[2]     controlValues←values.Item postDataKey
[3]     new←ParseControlValues controlValues
[4]     R←∨/new=_FahrenheitValue _CentigradeValue
[5]     _FahrenheitValue _CentigradeValue←new
      ∇
```



Param Name	Type	Modifier	Optional
Result	Boolean		
postDataKey	String		☐
values	NameValueCollecti		☐

Help

ID

● Method ○ Web Method ○ Prop Get ○ Prop Set

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

Line[7] obtains the two parameters from the argument and Line[8] 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, i.e. 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`*`[10]`.

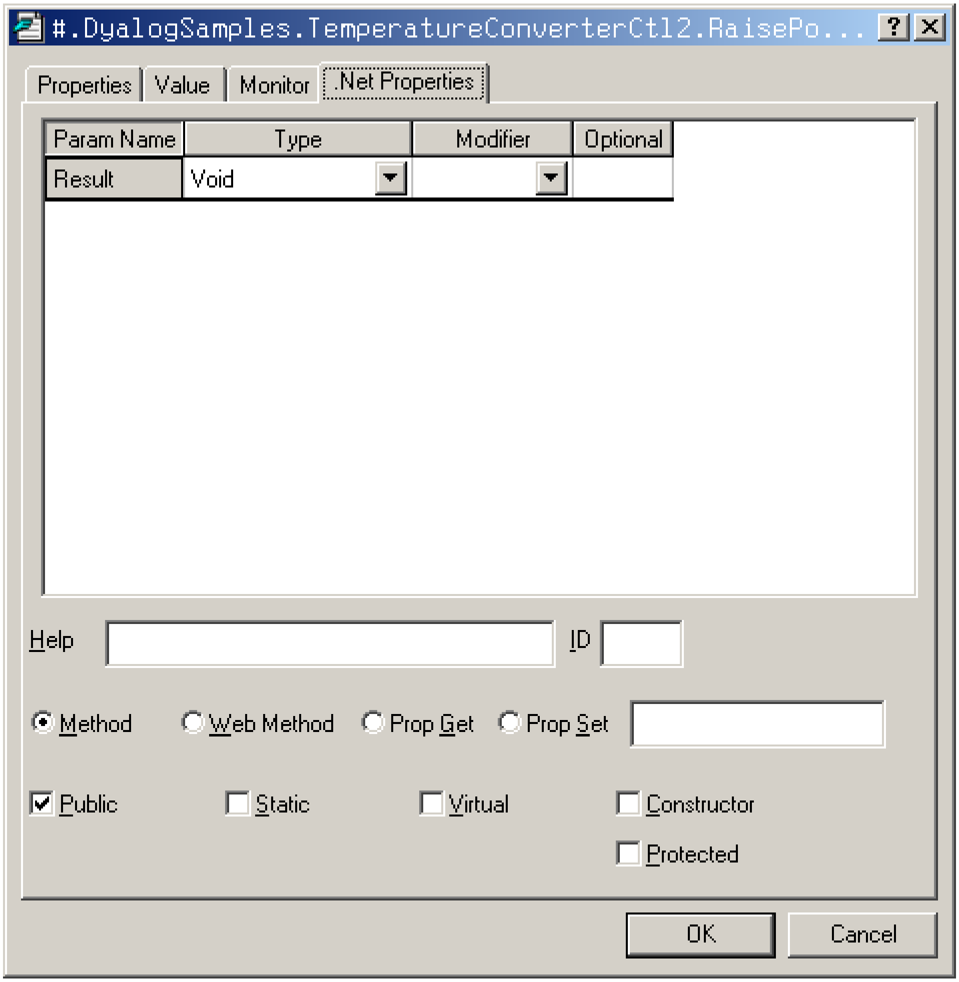
Finally, the page framework calls the `RaisePostBackEvent` method, passing it a string that identifies the page element that caused the post back.

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]      ⍝ Do nothing
       ∇
```



#.DyalogSamples.TemperatureConverterCtl2.RaisePo...

Param Name	Type	Modifier	Optional
Result	Void		

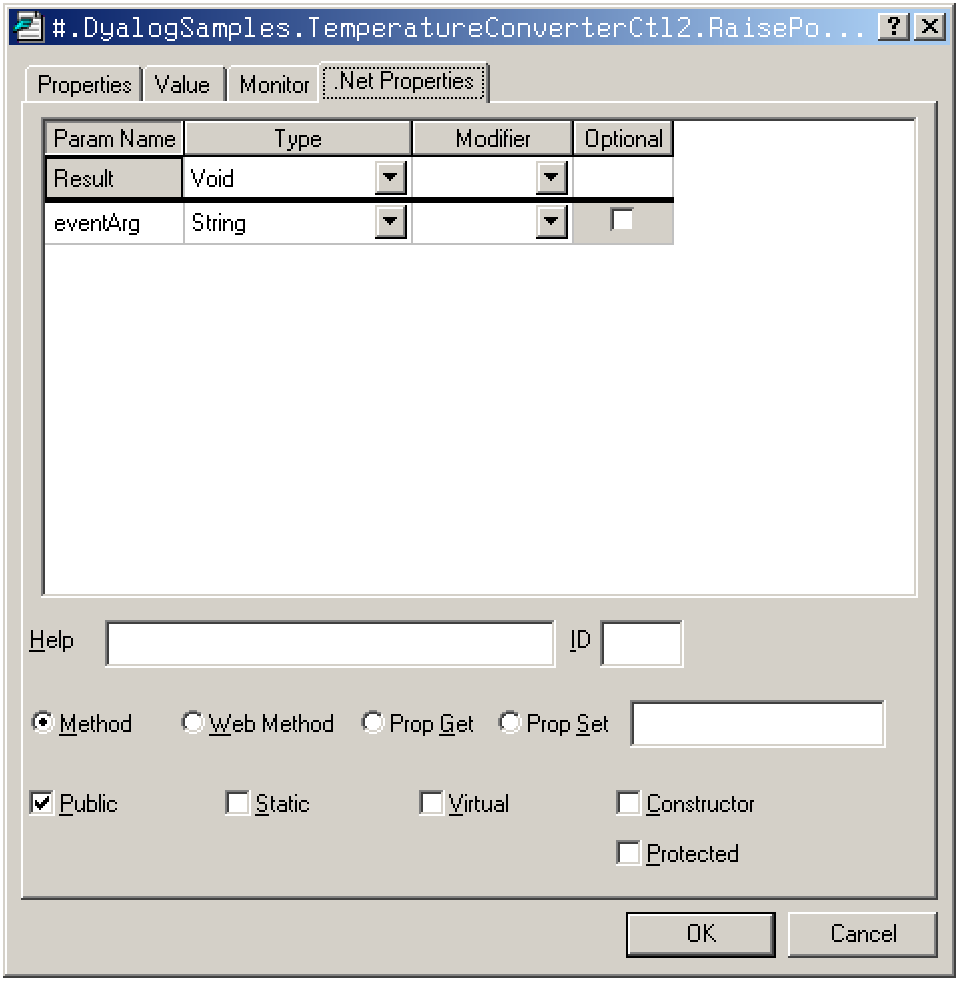
● Method ○ Web Method ○ Prop Get ○ Prop Set

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

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]     :Select eventArgument
[2]     :Case 'FahrenheitToCentigrade'
[3]         _CentigradeValue←F2C _FahrenheitValue
[4]     :Case 'CentigradeToFahrenheit'
[5]         _FahrenheitValue←C2F _CentigradeValue
[6]     :EndSelect
      ∇
```



Param Name	Type	Modifier	Optional
Result	Void		
eventArg	String		☐

Help
ID

● Method ○ Web Method ○ Prop Get ○ Prop Set

☑ Public ☐ Static ☐ Virtual ☐ Constructor

☐ Protected

Finally, the page framework calls the *`OnPreRender`* and *`Render`* functions again, which generate new HTML for the browser.

Using the Control on a Page

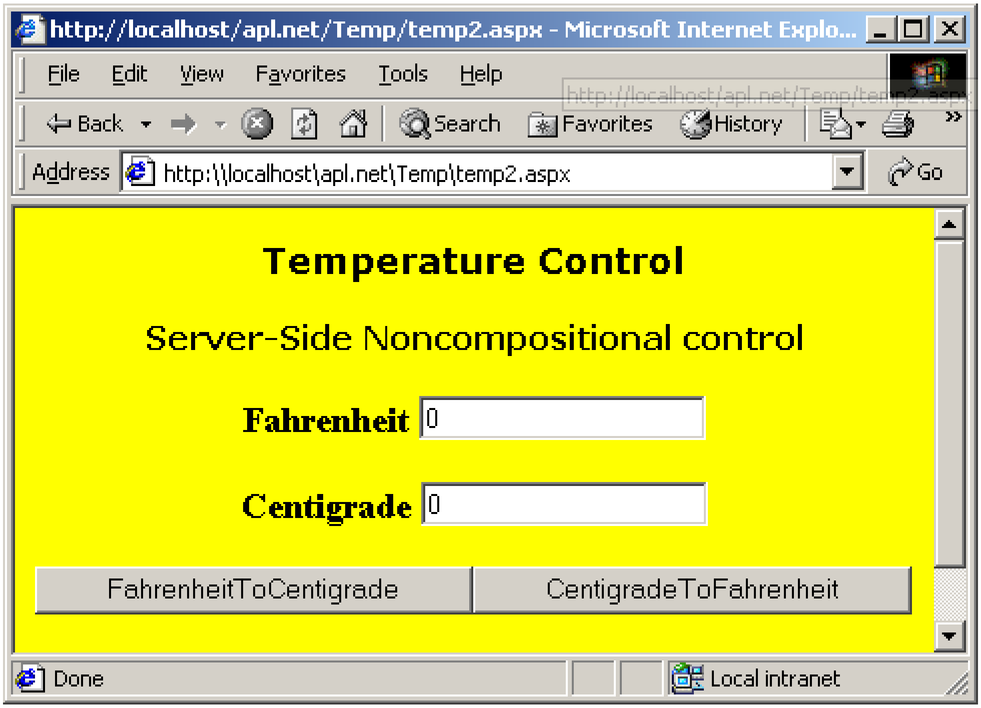
Once all the functions, and their public interfaces for the `TemperatureConverterCtl2` have been defined, the workspace is saved and `TEMP.DLL` is remade using *Export* from the Session *File* menu. For brevity, this process is not shown pictorially here.

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>
<h3><font face="Verdana" color="black">
Temperature Control</font></h3>
<h4><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>
<h3><font face="Verdana" color="black">Temperature
Control</font></h3>
<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=9cfcfa5c" />

<script language="javascript">
<!--
function __doPostBack(eventTarget, eventArgument) {
      var theform = document.ctrl1
      theform.__EVENTTARGET.value = eventTarget
      theform.__EVENTARGUMENT.value = eventArgument
      theform.submit()
}
// -->
</script>

<h3>Fahrenheit <input name=TempCvtCtl2 id=FahrenheitValue
type=text value=0></h3><h3>Centigrade <input name=TempCvtCtl2
id=CentigradeValueKey type=text value=0></h3><input
type=button value=FahrenheitToCentigrade
onClick="jscript:__doPostBack('TempCvtCtl2','FahrenheitToCent
igrade')"><input type=button value=CentigradeToFahrenheit
onClick="jscript:__doPostBack('TempCvtCtl2','CentigradeToFahr
enheit')">
<br>
<br>

</form>

</center>
</body>
</html>
```


http://localhost/apl.net/Temp/temp2.aspx - Microsoft Internet Explo...
File Edit View Favorites Tools Help
Back Search Favorites History
Address http:\\localhost\apl.net\Temp\temp2.aspx
Go
FahrenheitToCentigrade
CentigradeToFahrenheit
Done
Local intranet

Temperature Control

Server-Side Noncompositional control

Fahrenheit 0

Centigrade 0

CHAPTER 10

APLScript

Introduction

`APLScript` is a Dyalog APL 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 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 to view them. The Microsoft font *Arial Unicode MS* is included in Dyalog APL. The *SimPL* font designed by Phil Chastney may be downloaded from the British APL Association web site *http://www.vector.org.uk*.

In order to type Dyalog APL symbols into an `APLScript` file, you also need the Dyalog APL Input Method Editor (IME), which is included with Dyalog APL and automatically added to your working set of IMEs during installation.

The Dyalog APL IME may be configured using *Control panel/Keyboard*. In particular, you may change the associated `.DIN` file from the dialog box obtained by pressing *IME Settings* in the *Input Locales* tab. Under Windows XP, this is done using *Control panel/Regional and Language Options*.

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.

The APLScript Compiler, aplc.exe

APLScript files are *compiled* into executable code by the APLScript compiler `aplc.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, `aplc.exe` 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 APL 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 APL run-time `DYALOGRT.EXE`. Note too that the Dyalog APL dynamic link lbrary does not use MAXWS but instead allocates workspace dynamically as required. See User Guide for further details.
- Compile a Dyalog APL Workspace (.dws) into a native Windows executable program, with the same characteristics and advantages described above.

The `aplc.exe` program is designed to be run from a command prompt. If you type `aplc /?` (to query its usage) the following output is displayed:

```
aplc.exe command line options:

/?                  Usage
/r:file             Add reference to assembly
/o[ut]:file         Output file name
/res:file           Add resource to output file
/q                  Operate quietly
/v                  Verbose
/s                  Treat warnings as errors
/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. Used
                    when creating a process to run under IIS
/console            Creates a console application
```

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 *SimPL* or *Arial Unicode MS.*
3. Select the APL keyboard (IME) by clicking on your keyboard selector in the System Tray. When you do so, the keyboard icon will change to the APL IME icon, and a small floating toolbar containing an APL button will appear on your display. 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. Initially the APL button is depressed to indicate that keystrokes are being interpreted by the Dyalog APL IME. This utilises the input table (.DIN file) defined by the registry key *Software\Dyadic\IME\Translate Table*. Normally this will be the same as the one specified for your Dyalog APL Session.
5. Now type in your APL code. If you use a Unified keyboard, you will discover that Ctrl+ keystrokes generate APL symbols For example, Ctrl+n generates `⊤`. If you use an APL/ASCII keyboard, you will discover that Ctrl+n and Ctrl+o switch between APL and ASCII. In both cases, the keystrokes are intercepted immediately by the Dyalog APL IME and do not perform the standard Notepad operations (ctrl+n would normally open a new file). Note that you can toggle the APL button temporarily disable and enable the APL IME.
6. 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.

Transferring code from the Dyalog APL Session

You may find it easier to write APL code using the Dyalog APL function 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 (e.g. 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.

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, which is using the Arial Unicode MS font.

Unfortunately, when Dyalog APL requests Unicode data from the clipboard that was written as ASCII text by another application (including Dyalog APL Version 8 or 9), Windows converts the plain ASCII text to Unicode and gets it wrong. To make it possible to transfer APL expressions to and from previous versions of Dyalog APL, or to an editor using the (non-Unicode) Dyalog Std TT or Dyalog Alt TT font, you must set this parameter to 0.

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, 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.

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 `aplc.exe`. 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. However, note that it is not possible to export these functions as methods of .NET classes unless they have already been saved with .NET properties. Otherwise, all exported functions must be defined explicitly in the `APLScript`.

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.

Creating Programs (.exe) with APLScript

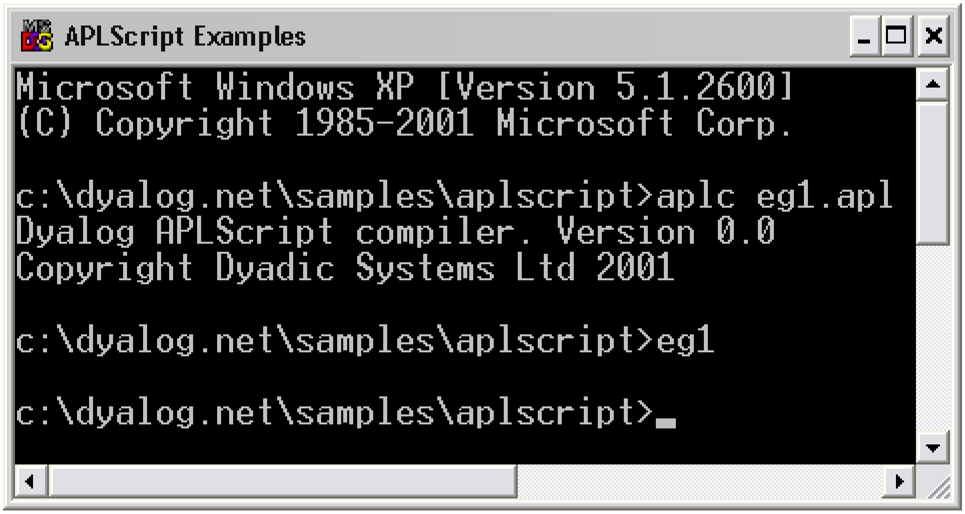
The following examples, which illustrate how you can create an executable program (`.exe`) direct from an APLScript file, may be found in the directory `samples\aplscript`.

A simple GUI example

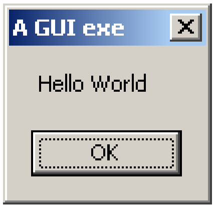
The following APLScript illustrates the simplest possible GUI application that displays a message box containing the string "Hello World".

```
⎕LX←'RUN'
∇RUN;M
'M'⎕WC'MsgBox' 'A GUI exe' 'Hello World'
⎕DQ'M'
∇
```

This example, which is saved in the file `eg1.apl`, is compiled to a Windows executable (`.exe`) using **aplc** and run from the same command window as shown below. Notice that it is essential to define a ⎕*LX* either in the APLScript itself, or as a parameter to the **aplc** command.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\dyalog.net\samples\aplscript>aplc eg1.apl
Dyalog APLScript compiler. Version 0.0
Copyright Dyadic Systems Ltd 2001

c:\dyalog.net\samples\aplscript>eg1

c:\dyalog.net\samples\aplscript>_
```



Hello World

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

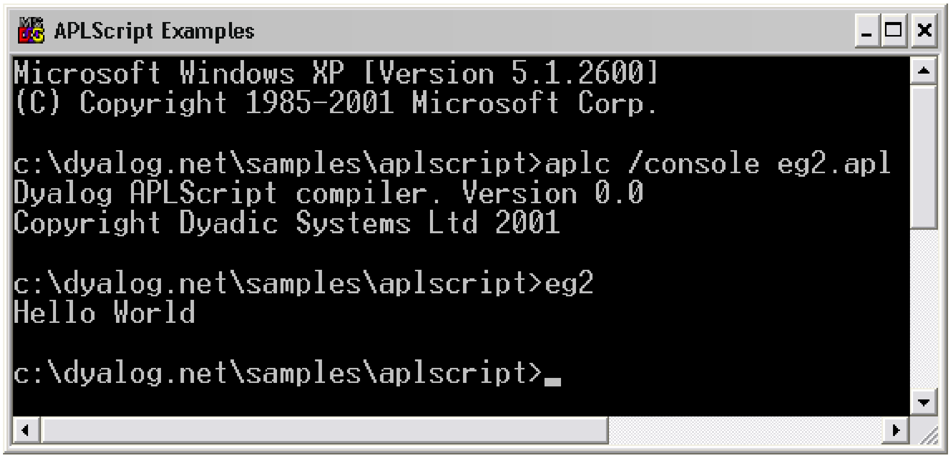
A simple console example

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

This example, which is saved in the file `eg2apl`, 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.

```
⎕LX←'RUN'
∇RUN
'Hello World'
∇
```

Once more, it is essential to define a `⎕LX` either in the APLScript itself, or as a parameter to the **aplc** command.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\dyalog.net\samples\aplscript>aplc /console eg2.apl
Dyalog APLScript compiler. Version 0.0
Copyright Dyadic Systems Ltd 2001

c:\dyalog.net\samples\aplscript>eg2
Hello World

c:\dyalog.net\samples\aplscript>_
```

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.

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

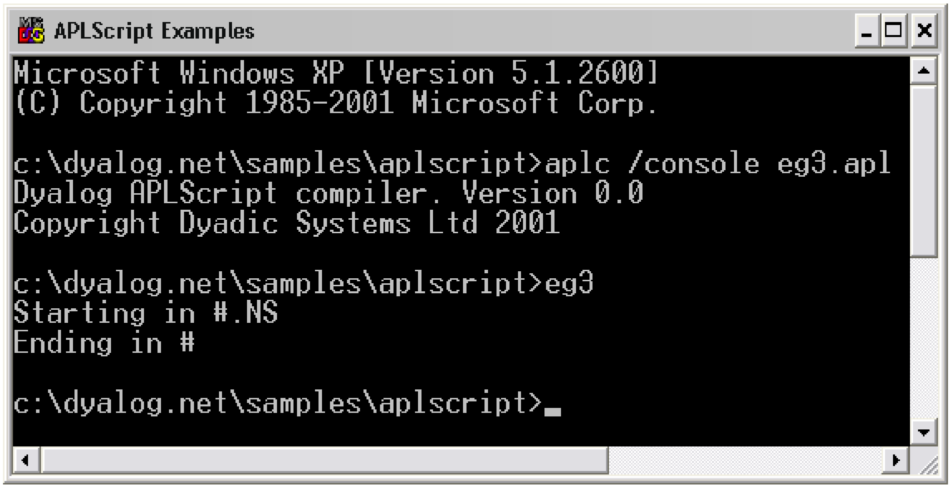
The following example illustrates how APL namespace usage is handled in APLScript. The program, contained in the file `eg3.apl`, is as follows:

```
⎕LX←'RUN'

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

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

This somewhat contrived example illustrates how a namespace is defined 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.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\dyalog.net\samples\aplscript>aplc /console eg3.apl
Dyalog APLScript compiler. Version 0.0
Copyright Dyadic Systems Ltd 2001

c:\dyalog.net\samples\aplscript>eg3
Starting in #.NS
Ending in #

c:\dyalog.net\samples\aplscript>_
```

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.

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 `Access: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 uses instances of a (local) class by invoking the `New` method in statements elsewhere in the `APLScript`.

Exporting Functions as 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 have to begin with 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 APL 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).

`:Access 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.

`:ParameterList type1 name1, type2 name2, ...`

Declares a parameter to the method to have a given data type and name. `Name` 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.

`:Returns Type`

Declares the result of the method to have a given data type.

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:

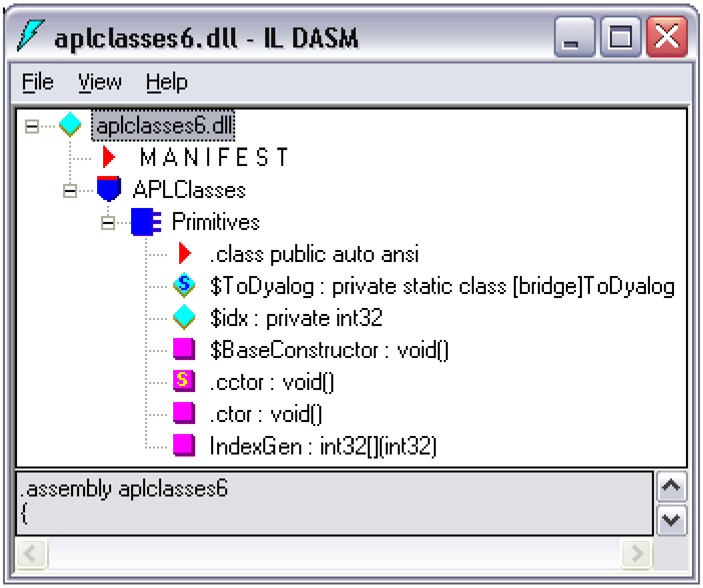
```
:Namespace APLClasses

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

∇ R←IndexGen N
:Access Public
:ParameterList Int32 number
:Returns Int32[]
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.

The following command shows how `aplclasses6.apl` is compiled to a .NET Assembly using the `/t:library` flag.

```
APLClasses>aplc /t:library aplclasses6.apl
Dyalog APLScript compiler Version 1.0
Copyright Dyadic Systems Limited 2002
APLClasses>
```

The next picture shows a view of the resulting `aplclasses6.dll` using ILDASM.



```
aplclasses6.dll
  MANIFEST
  APLClasses
    Primitives
      .class public auto ansi
      $ToDyalog : private static class [bridge]ToDyalog
      $idx : private int32
      $BaseConstructor : void()
      .cctor : void()
      .ctor : void()
      IndexGen : int32[](int32)
```

```
.assembly aplclasses6
{
```

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

```
      )CLEAR
clear ws

      ⎕USING←'APLClasses,Samples\APLClasses\aplclasses6.dll'
      APL←Primitives.New ⍬
      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 Type:Name`

Declares a new property called `Name` whose data type is `Type`. The latter may be and valid .NET type.

`: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.

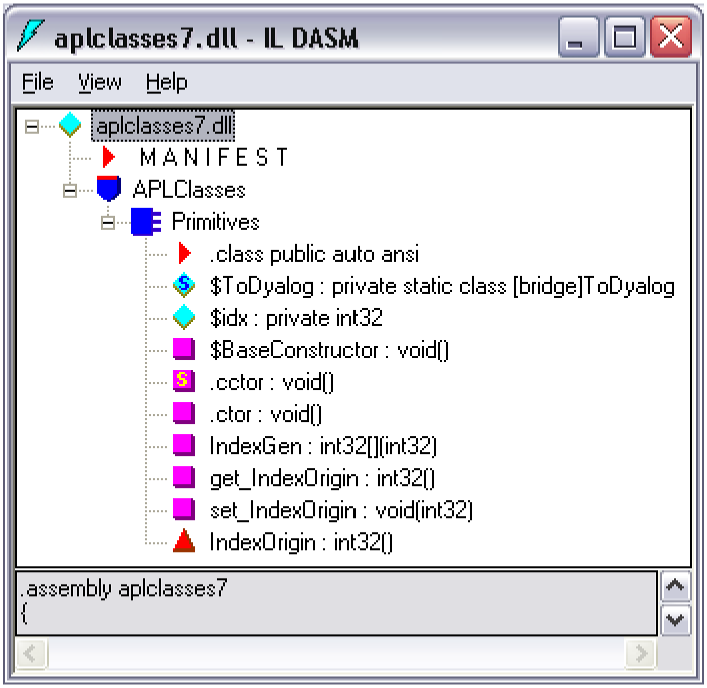
The accessor used to reference the value of the property is represented by a function named `get` that is defined within the `:Property` block. The accessor used to assign a value to the property is represented by a function named s`et` that is defined within the `:Property` block. No other function definitions or statements are allowed inside a `: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 must match the `Type` specified by the `:Property` statement. 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 the data type `Type` specified by the `:Property` 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'

∇ R←IndexGen N
:Access Public
:ParameterList Int32 number
:Returns Int32[]
R←⍳N
∇
:Property Int32:IndexOrigin
      ∇io←get
      io←⎕IO
      ∇
      ∇set io
      :If io∊0 1
          ⎕IO←io
      :EndIf
      ∇
:EndProperty

:EndClass

:EndNamespace
```

The `ILDASM` view of the new `aplclasses7.dll`, with the addition of an `IndexOrigin` property, is illustrated below.



```
aplclasses7.dll
    MANIFEST
    APLClasses
        Primitives
            .class public auto ansi
            $ToDyalog : private static class [bridge]ToDyalog
            $idx : private int32
            $BaseConstructor : void()
            .cctor : void()
            .ctor : void()
            IndexGen : int32[](int32)
            get_IndexOrigin : int32()
            set_IndexOrigin : void(int32)
            IndexOrigin : int32()
.assembly aplclasses7
{
```

For other examples of the use of property definitions, see *The Components File Solution* in Chapter 11.

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

```
      )CLEAR
clear ws

      ⎕USING←'APLClasses,Samples\APLClasses\APLClasses7.DLL'
      APL←Primitives.New ⍬
      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 member 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, but Dyalog APL does not itself allow indexing to be used on objects. This means that although you can define an APL class that exports an indexer, you can use the indexer from C# or Visual Basic, but not from APL.

Indexers are defined using `:Indexer` and `:EndIndexer` statements. An indexer block pertains to the class within which it is defined, and there may be only one indexer defined for a class.

`:Indexer Type:Name`

Declares an indexer called `Name` whose data type is Type. The latter may be and valid .NET type and specifies the data type of the indexed element of the class.

`:EndIndexer`

Terminates an indexer definition block

An indexer must have at least one parameter that is defined by a `:ParameterList` statement within the `:Indexer` block. These parameters identify the element of the object that is to be accessed.

Within a `:Indexer` block, you must define the accessors of the indexer. The accessors specify the code that is associated with referencing and assigning an element of the object.

The accessor used to reference the value of an element is represented by a function named `get` that is defined within the `:Indexer` block. The accessor used to assign a value to the element is represented by a function named `set` that is defined within the `:Indexer` block. No other function definitions or statements are allowed inside a `:Indexer` block.

The `get` function is used to retrieve an element of the object and must be a monadic result returning function. The argument for the `get` function will contain the parameters defined by the `:ParameterList` statement for the indexer. The data type of its result must match the `Type` specified by the `:Indexer` statement. The `set` function is used to change an element of the object and must be a monadic function with no result. The argument to the `set` function will contain the parameters defined for the indexer, and an additional parameter that specifies the new value of the element. This will have the data type `Type` specified by the `:Indexer` statement.

Note. The `:Indexer` statement in Dyalog APL 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. However, Dyadic has chosen to use the C# model which is also supported by Visual Studio.

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 APL. 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 <script> and </script> tags and normally appears in the <head> </head> 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="apl" runat=server>
```

and finishes with:

```
</script>
```

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

Web Service Layout

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

```
<%@ WebService Language="apl" 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="apl"`, 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 in the `Machine.config` file which is updated to associate `Language="apl"` with the appropriate Dyalog APL process when Dyalog APL is installed.

The `APLScript` compiler is itself written in Dyalog APL.

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 APL 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.

CHAPTER 11

Visual Studio Integration

Introduction

Dyalog APL supports loose integration with Microsoft Visual Studio.NET. Loose integration allows you to create Visual Studio projects using APLScript, and build .EXEs and .DLLs using Visual Studio as the front-end tool.

Dyalog APL is not yet tightly integrated with Visual Studio, and does not, for example, permit you to use the Visual Studio User Interface design tools.

The Dyalog APL installation program adds some sample APL applications in the appropriate Visual Studio directory, which are described in this Chapter.

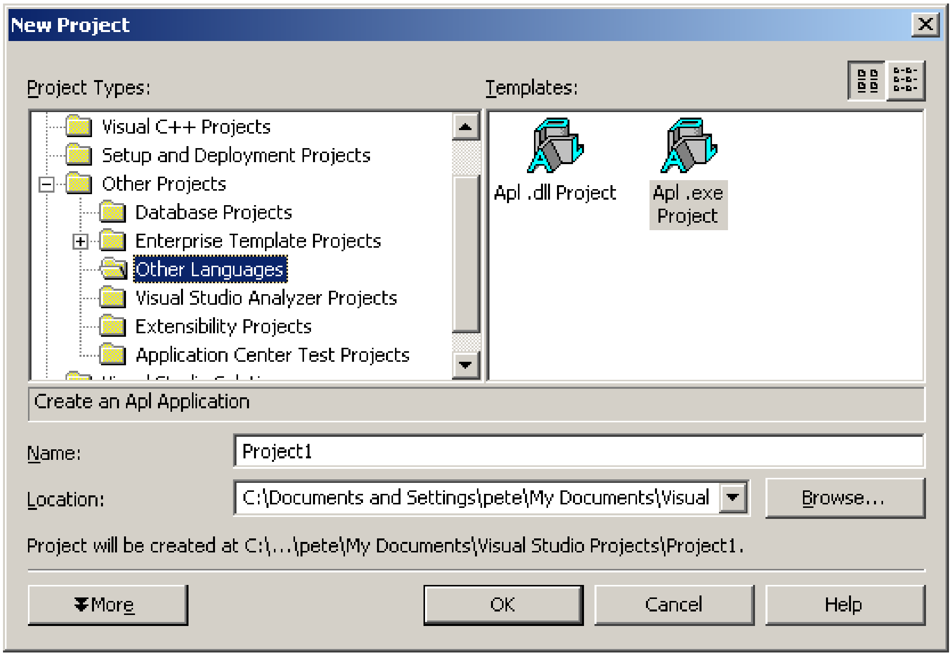
To begin with, the Hello World example shows you how to go about creating a .EXE program file using Visual Studio and APLScript.

Hello World Example

This example illustrates what is involved how you go about creating an application program (.exe) using APLScript with Visual Studio.

Creating an APL.EXE Project

Start Visual Studio and click *New Project*, or select *File/New/Project* from the menu bar. Navigate to the Other Projects/Other Languages folder. This gives you a choice of two APL templates as shown below.



Project Types: Other Languages

Templates: Apl .exe Project

Create an Apl Application

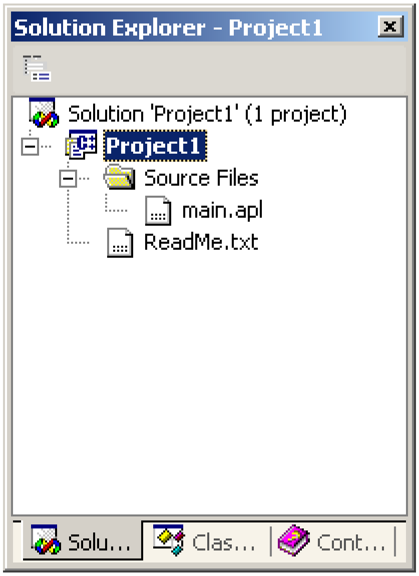
Name: Project1

Location: C:\Documents and Settings\pete\My Documents\Visual

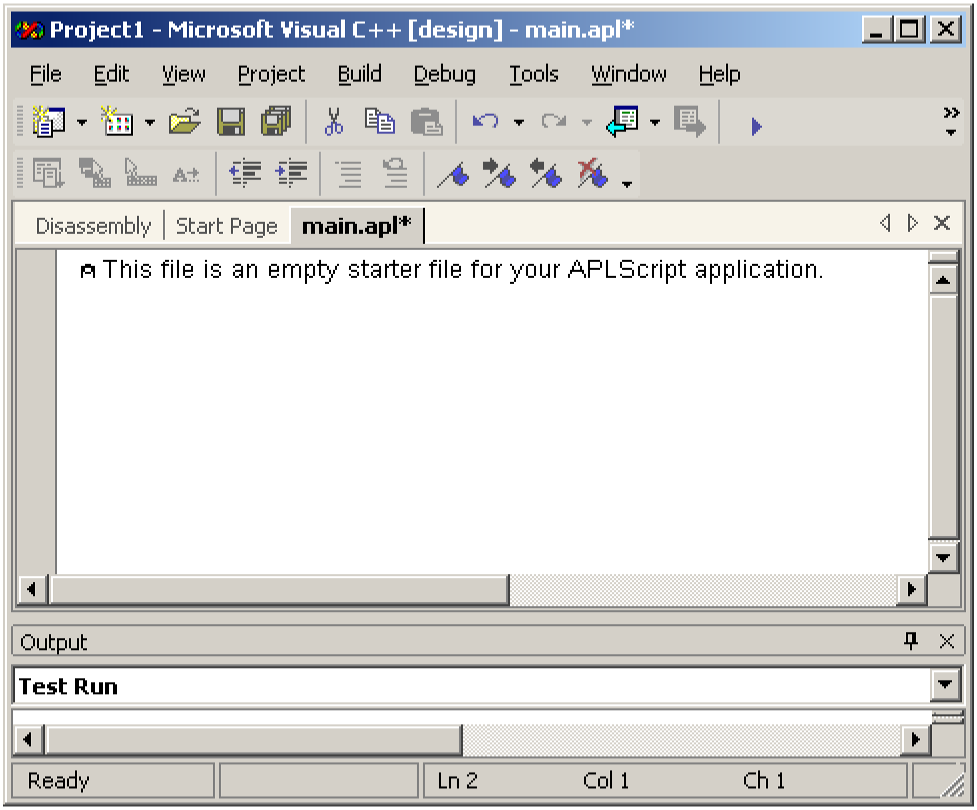
Project will be created at C:\...\pete\My Documents\Visual Studio Projects\Project1.

Select *APL.exe Project*, and click *OK.*

Visual Studio will then create a new Project, in this case named `Project1`, containing a single source code file named `main.apl` and a `ReadMe.txt` as shown below. The latter contains instructions about using Visual Studio with Dyalog APL.



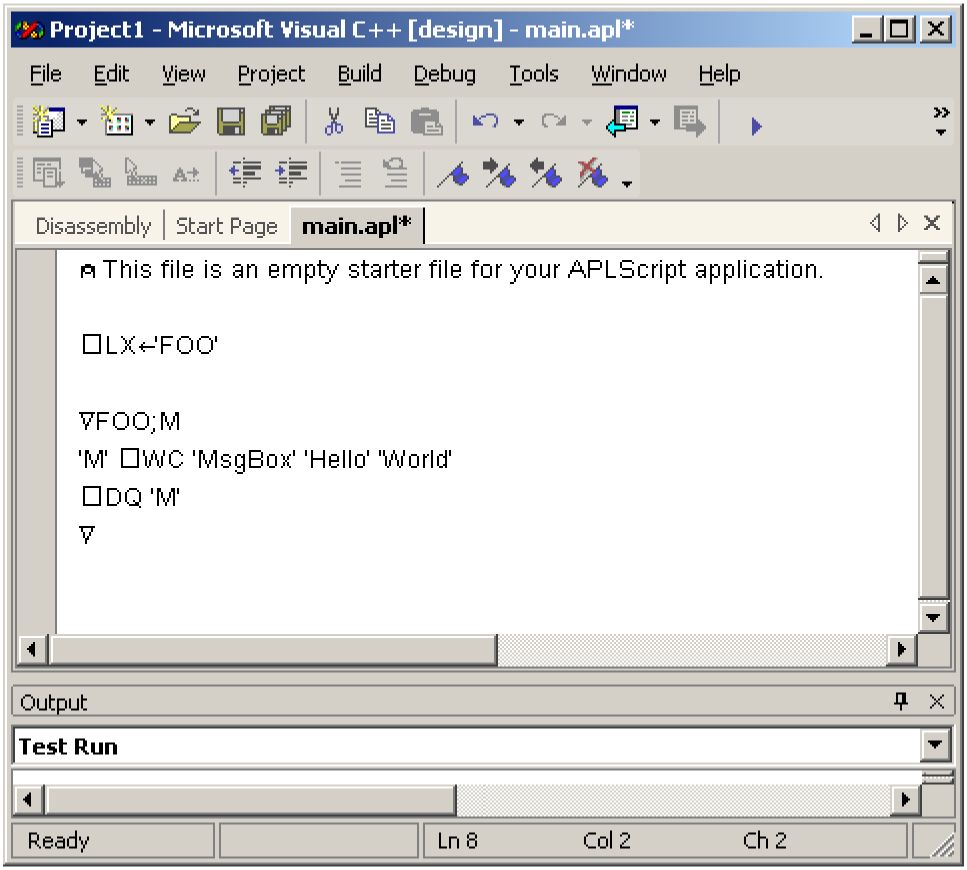
`main.apl` is an APLScript file containing a single comment as illustrated below. (Note that, at the time of writing, there is a bug in the .NET Framework that causes the APL comment sign to be displayed incorrectly, and you will need to change it.)


Project1 - Microsoft Visual C++ [design] - main.apl*
File Edit View Project Build Debug Tools Window Help
Disassembly
Start Page
main.apl*
Output
Ready
Ln 2
Col 1
Ch 1

```
⍝ This file is an empty starter file for your APLScript application.
```

Test Run

Select the APL keyboard (see *Input Method Editor*), and type your APLScript program. The following example illustrates code to display a *Hello World* message box.

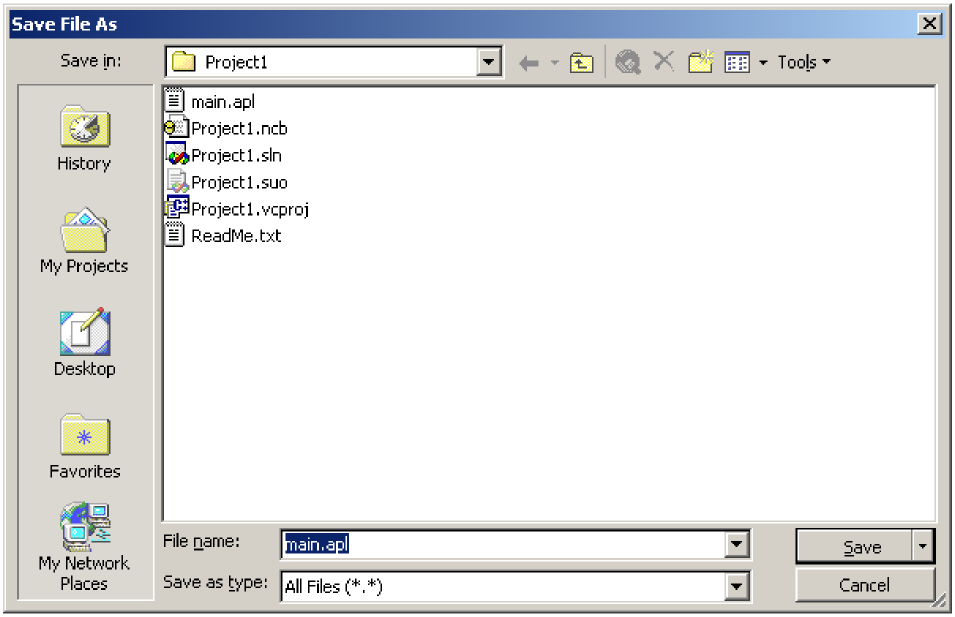


```
⍝ This file is an empty starter file for your APLScript application.

⎕LX←'FOO'

∇FOO;M
'M' ⎕WC 'MsgBox' 'Hello' 'World'
⎕DQ 'M'
∇
```

Notice that you must set `⎕LX` to start your application, but you do not have to explicitly call `⎕OFF` to end it.

The next step, re-saving `main.apl`, is only necessary because of a current limitation in Visual Studo.NET (it forgets the Unicode encoding).

Choose *Save main.apl As…* from the *File* menu. This brings up the dialog box shown below.



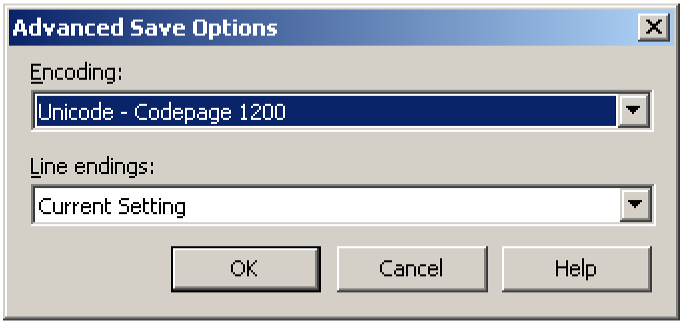
Save in: Project1

main.apl
Project1.ncb
Project1.sln
Project1.suo
Project1.vcproj
ReadMe.txt

File name: main.apl

Save as type: All Files (*.*)

Now click the dropdown part of the *Save* button and choose *Save with Encoding…* from the pop-up menu. This brings up the Advanced Save Options dialog illustrated below.

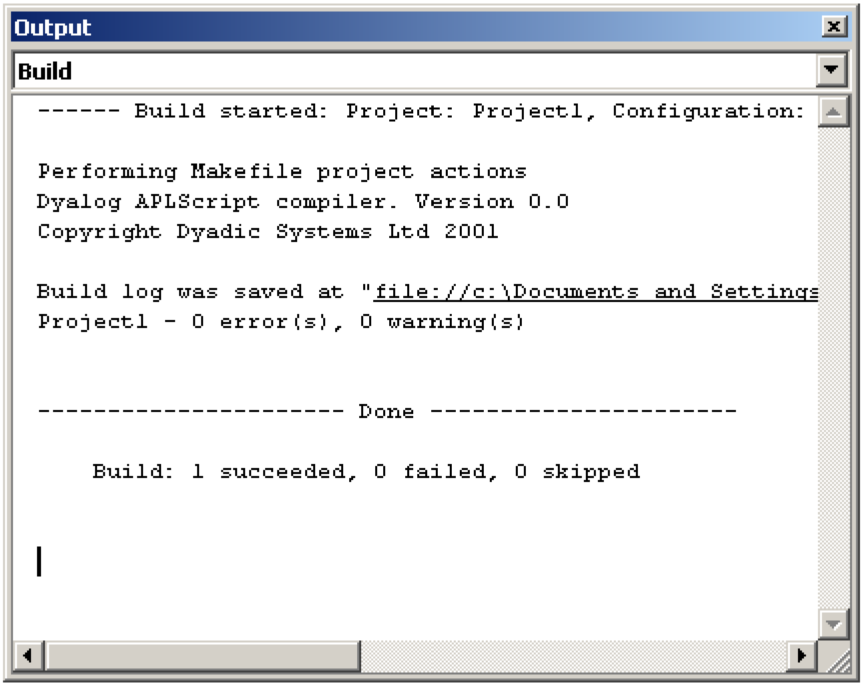


Encoding: Unicode - Codepage 1200

Line endings: Current Setting

Choose *Unicode - Codepage 1200* from the *Encoding* menu, then click *OK*.

The next step is to build the project. To do this, select *Build* from the *Build* menu. Assuming that your code is correct, the following messages will appear in the *Output* window.

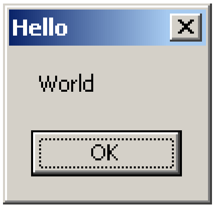


```
------ Build started: Project: Project1, Configuration:

Performing Makefile project actions
Dyalog APLScript compiler. Version 0.0
Copyright Dyadic Systems Ltd 2001

Build log was saved at "file://c:\Documents and Settings
Project1 - 0 error(s), 0 warning(s)


---------------------- Done ----------------------

    Build: 1 succeeded, 0 failed, 0 skipped
```

The result of this process is an executable program named `Project1.exe`.

To run the program, select Debug/Start Without Debugging from the menu (or press Ctrl+F5)

The program displays the dialog box shown below, waits for you to click *OK*, and then exits.



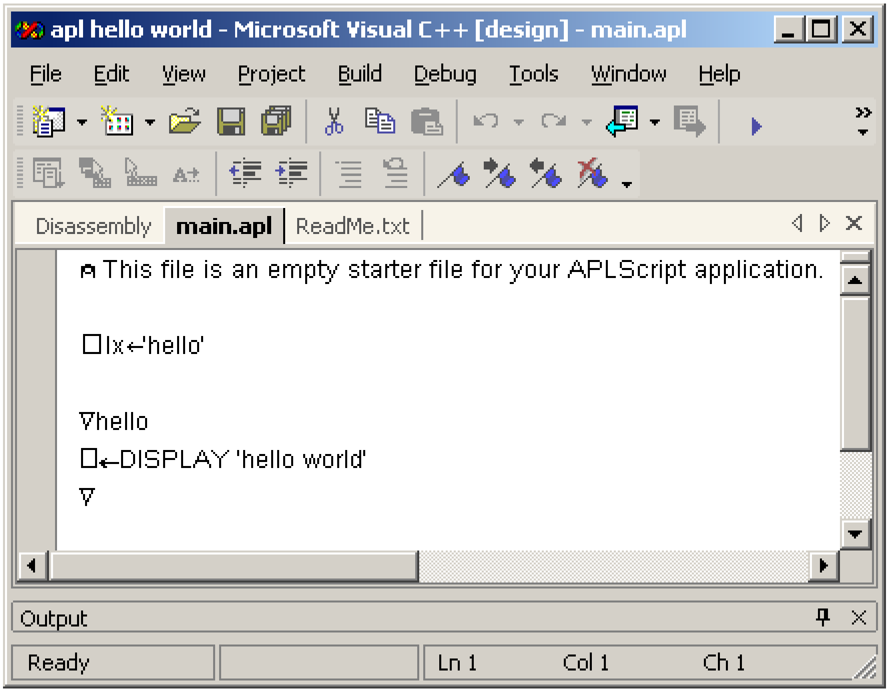
Using an Existing Workspace

This next example takes the approach a stage further and illustrates how an application built using Visual Studio can access an existing workspace.

Go to the *Start* page and click *New Project*.

Follow the steps described previously to create a new APL.EXE project (named "hello world", and type the APLScript code shown below into `main.apl`.

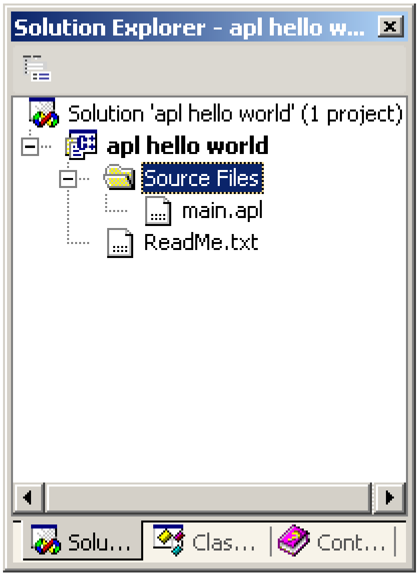
Notice that it refers to a function *`DISPLAY`* that is not itself defined in the script.



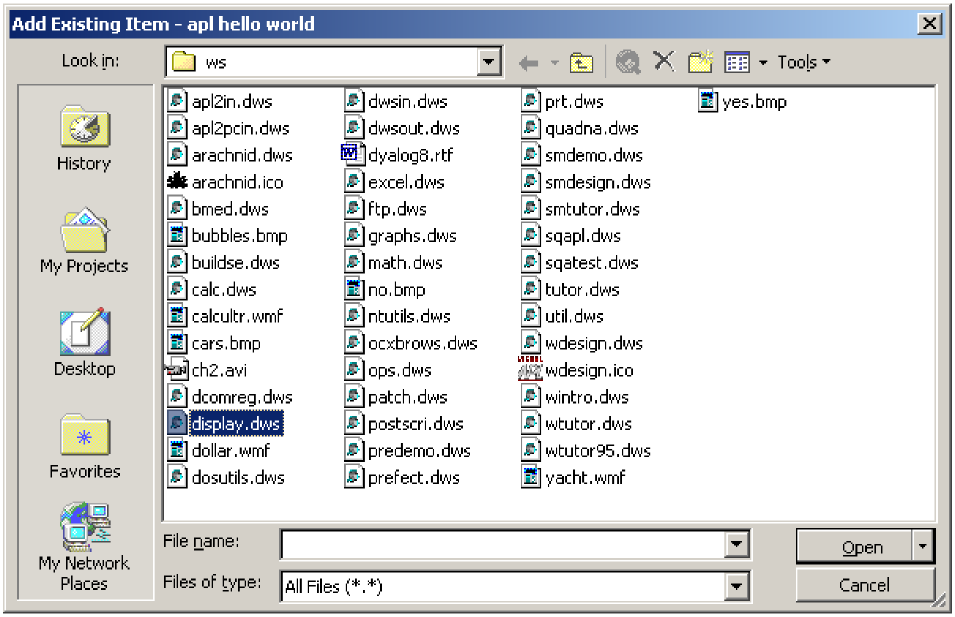
```
⍝ This file is an empty starter file for your APLScript application.

⎕lx←'hello'

∇hello
⎕←DISPLAY 'hello world'
∇
```

The DISPLAY function will be provided by the DISPLAY workspace, which you can add to a project as follows.

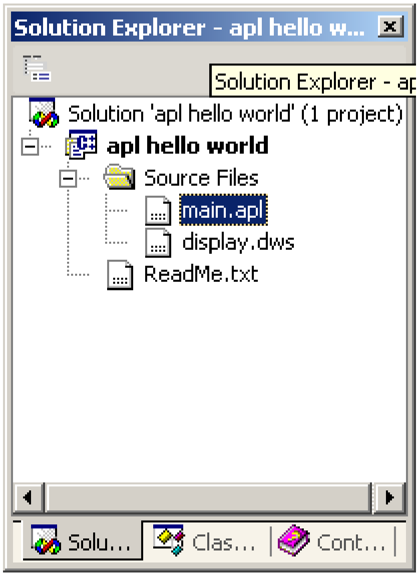
In the *Solution Explorer* window, select *Source Files* and click the right button to bring up the context menu.


Solution Explorer - apl hello w...
Solution 'apl hello world' (1 project)
apl hello world
Source Files
main.apl
ReadMe.txt
Solu...
Clas...
Cont...

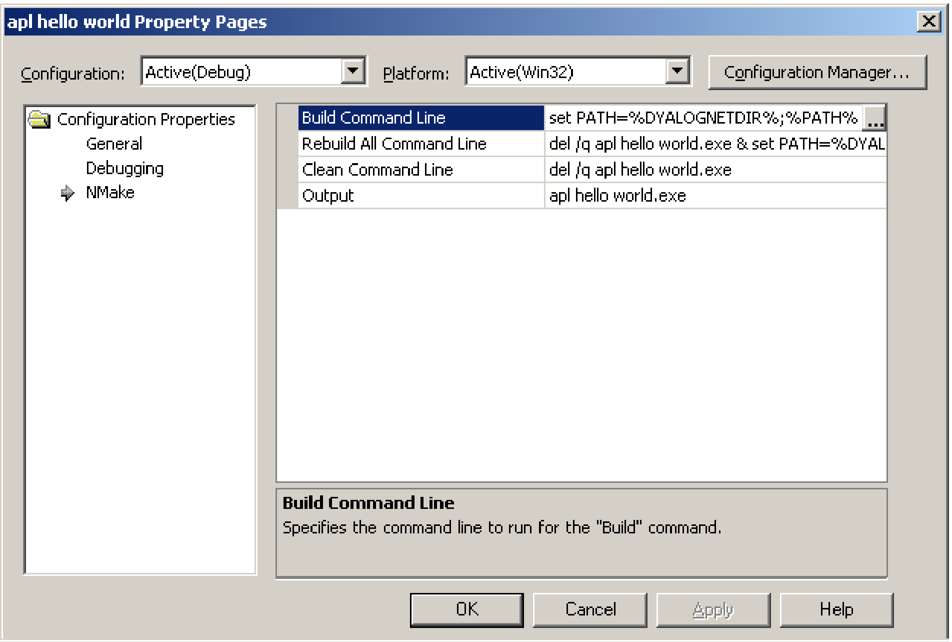
Select *Add*, and then *Add Existing Item*…. This brings up a file selection dialog. Navigate to c:\dyalog90\ws and choose display.dws.



This file is then added to the project as shown below.



The next stage is to add information to the *Build Command Line* for the project (this will happen automatically in future versions). To do this, select the *apl hello world* project, click the right button and select *Properties*. This brings up the *apl hello world Property Pages* dialog. Select the *NMake* page (from the left pane) as shown below.



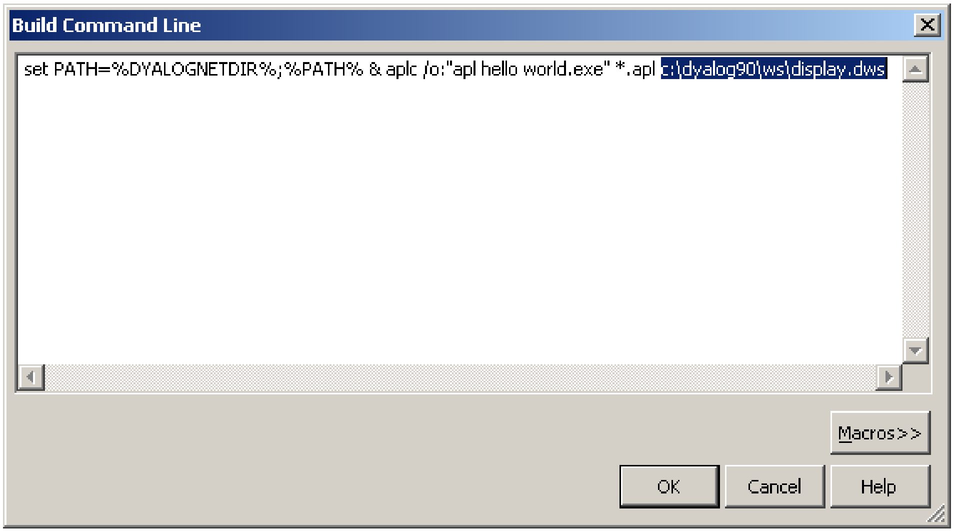
Configuration: Active(Debug)
Platform: Active(Win32)

Build Command Line	set PATH=%DYALOGNETDIR%;%PATH%
Rebuild All Command Line	del /q apl hello world.exe & set PATH=%DYAL
Clean Command Line	del /q apl hello world.exe
Output	apl hello world.exe

Build Command Line
Specifies the command line to run for the "Build" command.

Click on *Build Command Line*, then click the "…" button to bring up the *Build Command Line* editor.

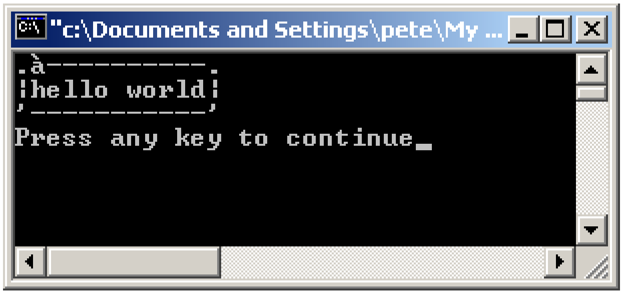
Change the workspace reference from **.dws* to *c:\dyalog90\display.dws* (or wherever it is installed on your computer) as shown below. Then click *OK*.



```
set PATH=%DYALOGNETDIR%;%PATH% & aplc /o:"apl hello world.exe" *.apl c:\dyalog90\ws\display.dws
```

Then select *Build* from the menu ,or press Ctrl+Shift+b, to build the project.

Run the program by selecting *Debug/Start without Debugging* or press Ctrl+F5. This particular program sends its output to a console window as shown below.



```
.à----------.
¦hello world¦
'-----------'
Press any key to continue_
```

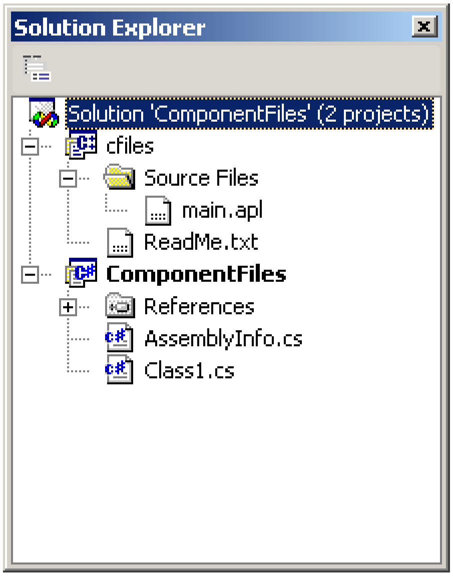
The Component Files Solution

This example illustrates a Visual Studio Solution that combines an APL project with a project written in another language, in this case C#. It illustrates how a C# program can read and write APL component files.

As part of the installation of Dyalog APL, the *Component Files* solution sample is installed in `My Documents\Visual Studio Projects\ComponentFiles`

Select *Open Solution* from the *File* menu, navigate to the `ComponentFiles` folder, open it and then select `ComponentFiles.sln`.

The Solution Explorer window shows that there are 2 projects. The first, called `cfiles`, is written in APL; the second, called `ComponentFiles`, is written in C#.



The cfiles project (APL)

The APLScript in `main.apl` is shown below. It defines a class named `ComponentFile`. This has a single constructor function called *`make`* which takes the name of an APL component file as its parameter. This function ties the file, and creates an (internal) object called *`components`* that is an instance of the internal class, `FileComponents`. This object is exposed via a property named `Components`. This is a read-only property because only its *`get`* function has been defined. The `ComponentFile` class provides an override for the `ToString` method which displays the name of the file prefixed by the string *Componentfile:*. Finally, `ComponentFile` exposes a method called `Close` that closes the file.

```
⎕io←1
⎕ml←0

:Class ComponentFile

      :Class FileComponents

               ∇ make arg
            [1]  :Access constructor
            [2]  :ParameterList Int32
            [3]
            [4]  tie←arg
               ∇

            :Property Int32:Count
               ∇ r←get
            [1]  r←¯1+2⊃⎕fsize tie
               ∇
            :EndProperty

               ∇ r←Add array
            [1] :Access public
            [2] :ParameterList Array
            [3] :Returns Int32
            [4]
            [5] r←array ⎕fappend tie
               ∇

            :Indexer Array:Item
            :ParameterList Int32
               ∇ set args
            [1] (2⊃args) ⎕freplace tie (1⊃args)
               ∇
               ∇r←get index
            [1] r←⎕fread tie index
               ∇
            :EndIndexer

      :EndClass
```

```
     ∇ make args
      :Access public constructor
      :ParameterList String

      tie←args ⎕ftie 0
      components←FileComponents.New tie
∇

     r←ToString  ⍝ overrides base class member
      :Access public
      :Returns String

      r←'ComponentFile:' ,⎕fnames[⎕fnums⍳tie;]
∇

:Property FileComponents:Components
     ∇r←get
      r←components
     ∇
:EndProperty

     ∇Close
      :Access Public

      ⎕funtie tie
     ∇
:EndClass
```

The internal class `FileComponents` is defined within the `ComponentFile` class and is local to it. Its constructor function *`make`* simply remembers the tie number of the file in a variable called tie. This variable is local to this particular instance of the `FileComponents` object.

The `FileComponents` class exports a property named `Count` whose value is the number of components in the file. This property is read-only because there is no *`set`* function defined for it.

The `FileComponents` class exports a method named `Add`, which takes an object of type `Array`, and appends it to the file.

The `FileComponents` class has a `:`*`Indexer`* section which specifies that an instance of the class may be referenced using indexing, as if it were an array, if the host language supports this feature. The C# code to do this is discussed later. The `:`*`Indexer`* statement specifies the type and name of an element of the *object-as-an-array*; in this case the element is called `Item` and is of type `Array`. The `:`*`ParameterList`* statement of the indexer specifies that the index itself is of type `Int32`. The *`set`* function takes both parameters; the index and the (new) element value. The *`get`* function takes only the index.

The ComponentFiles project (C#)

The C# source code for `Class1.cs`, that employs the APL `ComponentFile` class, is shown below.

The program first creates an instance of the `ComponentFile` class named `file`. Next, it displays the contents of each component in the file using its DumpArray subroutine. Notice that `File.Components` refers to a `FileComponents` object, and `File.Components[I]` refers to the ith element of this object, namely the ith component in the file. The program goes on to add a component using the `Add` method, and to replace it using indexing.

```
using System;
namespace ComponentFiles
{
class Class1
{
	public static void Main()
	{
	ComponentFile file = new
		ComponentFile(".\\cfiles.dcf");

	for (int i=1;i<=file.Components.Count;i++)
		DumpArray(file.Components[i]);
	Console.WriteLine(file.ToString());
	Console.Write("file.Count:");
	Console.WriteLine(file.Components.Count);
	Console.Write("file.Components[2]:");
	DumpArray(file.Components[2]);
	int[] New = new int[3];
	New[0]=1;
	New[1]=3;
	New[2]=5;
	Console.Write("Added component at ");
	int at = file.Components.Add(New);
	Console.WriteLine(at);
	Console.Write("New component contains:");
	DumpArray(file.Components[at]);
	New[0]=11;
	New[1]=33;
	New[2]=55;
	Console.Write("Overwritten component.Now contains:");
	file.Components[at]=New;
	DumpArray(file.Components[at]);
	file.Close();
	}
```

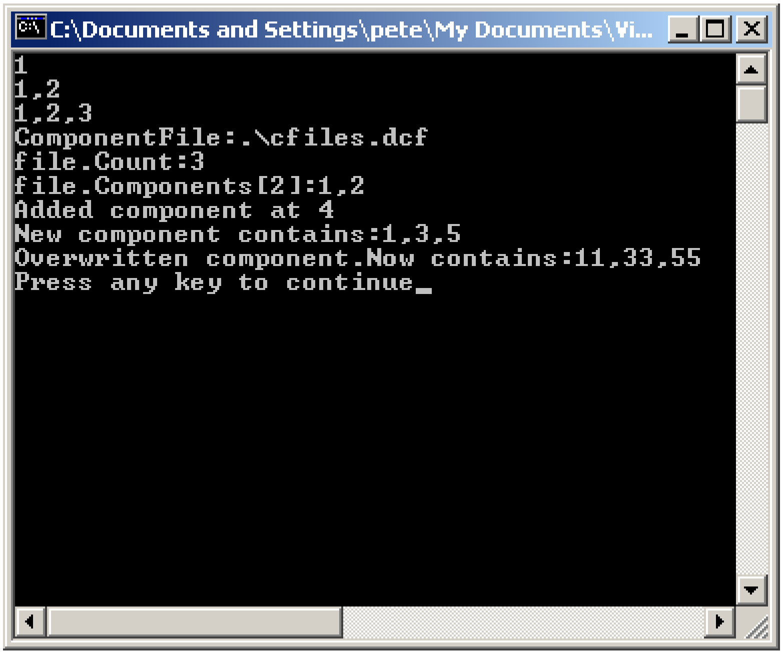
```
      static void DumpArray(Array a)
      {
            switch (a.Rank)
            {
                  case 1:
                        for (int i=0;i<a.Length;i++)
                        {
                              if (i!=0)
                                    Console.Write(",");
                              Console.Write(a.GetValue(i));
                        }
                        break;
            }
            Console.WriteLine();
      }
}
}
```

Running the Solution

Select *Build* from the menu, or press Ctrl+Shift+b, to build the project.

Run the program by selecting *Debug/Start without Debugging* or press Ctrl+F5. This particular program sends its output to a console window as shown below.

Starting with a component file containing 3 components, the output from the program is shown below.



```
1
1,2
1,2,3
ComponentFile:.\cfiles.dcf
file.Count:3
file.Components[2]:1,2
Added component at 4
New component contains:1,3,5
Overwritten component.Now contains:11,33,55
Press any key to continue_
```

CHAPTER 12

Implementation Details

Introduction

`dyalog10.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. `dyalog10.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.

`dyalog10.dll` is the full developer version of the DLL that 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

`dyalog10rt.dll` is the re-distributable run-time version of `dyalog10.dll` and contains no debugging facilities.

If there are several applications running on your computer that use APL .NET Classes, each one will have a separate copy of `dyalog10.dll` loaded in its process space. However, each application could be providing services to a number of users, and could be hosting a number of different APL .NET classes.

The dyalog10.dll active workspace

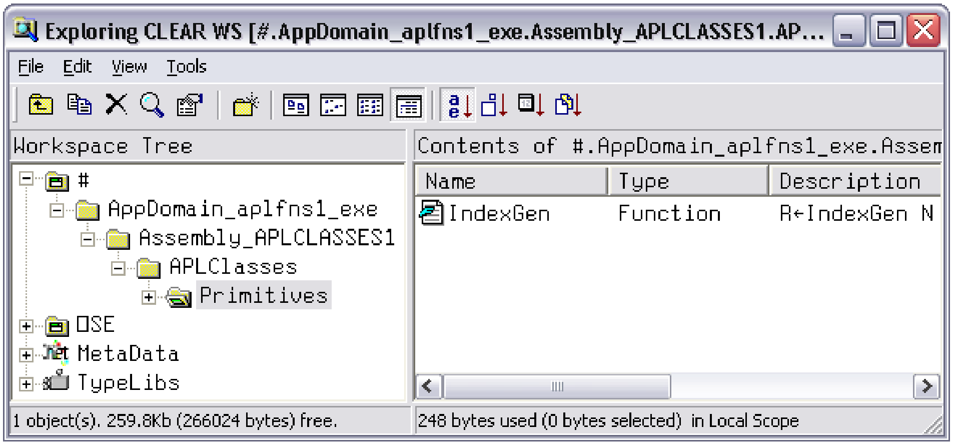
`dyalog10.dll` has a workspace associated with it that contains all the APL objects it is currently hosting.

The workspace will contain one or more namespaces associated with .NET *AppDomains*. An AppDomain is a .NET Class that represents an application domain, which is an isolated environment where applications execute. 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 `dyalog10.dll` workspace when the `aplfns1.exe` program is executed. This example is discussed as Example1 in Chapter 5. To cause the suspension, a stop has been set on `IndexGen[1]`.



Name	Type	Description
IndexGen	Function	R←IndexGen N

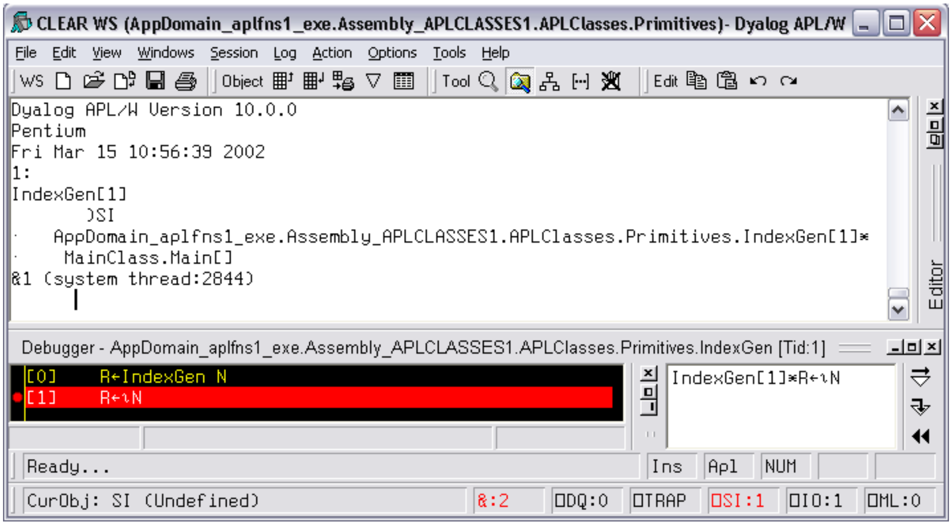
1 object(s). 259.8Kb (266024 bytes) free.

248 bytes used (0 bytes selected) in Local Scope

In this case, there is a single AppDomain involved whose name, `aplfns1.exe,` is specified by .NET. APL has made a corresponding namespace called `AppDomain_aplfns1`. Next, there is a namespace associated with the Assembly `aplclasses1`, named `Assembly_APLCLASSES1`. Beneath this is a namespace called `APLClasses` associated with the .NET Namespace of the same name. Finally, there is a namespace called `Primitives` that represents the APL class of that name. This namespace contains all the code associated with the class; in this case, just a single function called `IndexGen`.

The next picture shows the APL Session window that is displayed with execution suspended on `IndexGen[1]`. Notice that the State Indicator in Dyalog APL has been extended to display 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 `aplfns1.cs`. Note that .NET calls are slightly indented.

This extension to `)SI` applies also to DYALOG.EXE. For example, if you attach an APL callback function to a Winforms Button object, the callback is executed as a result of a call from the Button object back into the APL environment. The State Indicator will show the entire call stack, including methods in the .NET components.


CLEAR WS (AppDomain_aplfns1_exe.Assembly_APLCLASSES1.APLClasses.Primitives)- Dyalog APL/W
File
Edit
View
Windows
Session
Log
Action
Options
Tools
Help
WS
Object
Tool
Edit
Editor
Debugger - AppDomain_aplfns1_exe.Assembly_APLCLASSES1.APLClasses.Primitives.IndexGen [Tid:1]

```
Dyalog APL/W Version 10.0.0
Pentium
Fri Mar 15 10:56:39 2002
1:
IndexGen[1]
      )SI
·   AppDomain_aplfns1_exe.Assembly_APLCLASSES1.APLClasses.Primitives.IndexGen[1]*
·    MainClass.Main[]
&1 (system thread:2844)
```

```
[0]     R←IndexGen N
[1]     R←⍳N
```

```
IndexGen[1]*R←⍳N
```

```
Ready...                                  Ins   Apl   NUM
CurObj: SI (Undefined)       &:2   ⎕DQ:0   ⎕TRAP   ⎕SI:1   ⎕IO:1   ⎕ML:0
```

Notice too that `IndexGen` has been started on APL thread 1 which, in this case, is associated with system thread 2844. 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. This is implemented by having a separate **unnamed** instance namespace associated with every client instance of the APL object. Each instance namespace starts off as a *clone* of the class namespace (in this case `Primitives`), but as the object is used, values of variables will typically differ from one instance to another. Note that the instance namespaces are created using a *shallow copy*. Functions and variables in the instance are effectively just pointers to functions and variables in the class and do not use undue amounts of workspace. Only when an instance variable changes value will it occupy additional space in the workspace.

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 your write an APL Web Page that calls an APL Web Service, both of which are being hosted by a single `dyalog10.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 `dyalog10.dll`, the Dyalog APL 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 APL 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 (`dyalog10.dll`). The latter is actually the simpler of the two and will be considered first.

DYALOG10.DLL Threading

In this case, all calls into `dyalog10.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 `dyalog10.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 APL. 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 APL will potentially *thread switch*, i.e. switch execution from one APL thread to another, at the start of any line of APL code. In addition, Dyalog APL 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.

Debugging an APL .NET Class

All APL.NET objects are executed by the Dyalog APL dynamic link library `dyalog10.dll` or `dyalog10rt.dll`. The former contains all of the development and debug facilities of the APL Session, including the Editors and Tracer. The latter contains no debugging facilities at all. You choose to which of the two DLLs your APL .NET class is bound, when you create the class.

If an APL .NET object that is bound to `dyalog10.dll` 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 `dyalog10rt.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 DLLs your APL.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 `dyalog10.dll`. If *Runtime application* is checked, the .NET Class will be bound to `dyalog10rt.dll`.

If you build a .NET class using the APLScript compiler, it will by default be bound to `dyalog10.dll`. If you specify the `/runtime` flag, it will be bound to `dyalog10rt.dll`.

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 `dyalog10.dll`. If it specifies `"Debug=false"`, the Web Page or Web Service will be bound to `dyalog10rt.dll`.

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="apl" runat="server">

∇Rotate args
:Access Public
:ParameterList 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 `dyalog10.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, e.g. by calling a method, for this to occur. Note that this technique may not work if `dyalog10.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 APL DLL does not display an icon in the System Tray.

Using the Session, Editor and Tracer

When an APL.NET object suspends execution, all other active APL .NET objects bound to `dyalog10.dll` that are currently being executed by the same client application will also suspend. Furthermore, all the classes currently being hosted by `dyalog10.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* DLL, these objects will be hosted in a separate workspace attached to `dyalog10rt.dll` and will not be visible to the developer.

Debugging a running APL.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 `dyalog10.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 APL 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.

Index