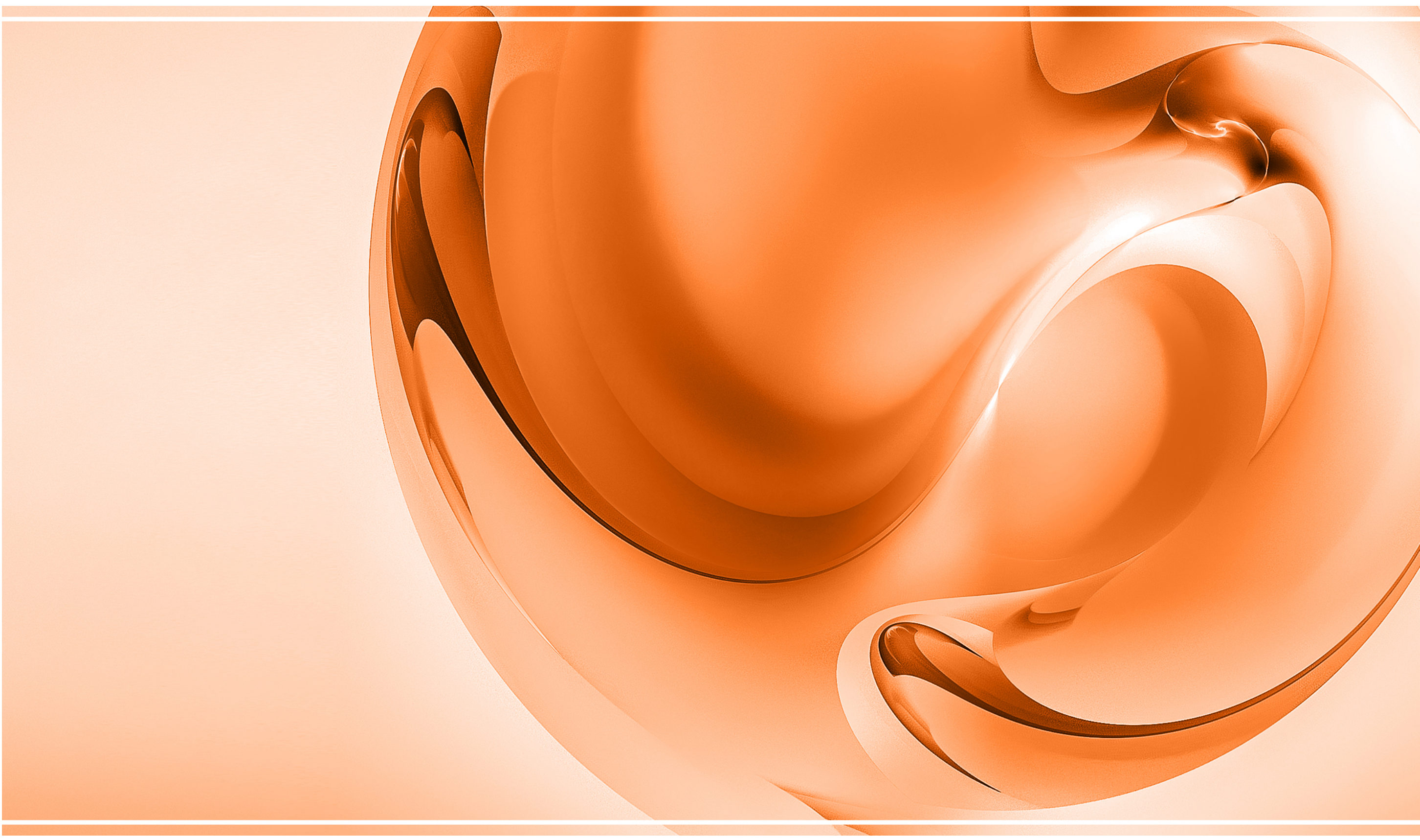
.NET Interface Guide

Dyalog version 19.0



DYALOG

The tool of thought for software solutions

Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2024 by Dyalog Limited
All rights reserved.

.NET Interface Guide

Dyalog version 19.0
Document Revision: 20240223_190

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

No part of this publication may be reproduced in any form by any means without the prior written permission of Dyalog Limited.

Dyalog Limited makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Dyalog Limited reserves the right to revise this publication without notification.

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

TRADEMARKS:

Array Editor is copyright of davidliebtag.com
Raspberry Pi is a trademark of the Raspberry Pi Foundation.
Oracle®, JavaScript™ and Java™ are registered trademarks of Oracle and/or its affiliates.
UNIX® is a registered trademark in the U.S. and other countries, licensed exclusively through X/Open Company Limited.
Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.
Windows® is a registered trademark of Microsoft Corporation in the U.S. and other countries.
macOS® and OS X® (operating system software) are registered trademarks of Apple Inc. in the U.S. and other countries.

All other trademarks and copyrights are acknowledged.

Contents

1 About This Document

This document describes the Dyalog interface to .NET, the cross-platform (Microsoft Windows, Linux and macOS) successor to Microsoft's .NET Framework. It describes how Dyalog communicates with .NET, but does not attempt to explain the features of .NET; for information concerning .NET, see Microsoft's documentation, articles and helpfiles (available from https://docs.microsoft.com/en-us/dotnet/).

.NET is not available for IBM AIX.

1.1 Audience

It is assumed that the reader has a working knowledge of Dyalog and a basic understanding of OO methodologies, and is familiar with .NET and/or .NET Framework.

For information on the resources available to help develop your Dyalog knowledge, see https://www.dyalog.com/introduction.htm.

1.2 Conventions

Unless explicitly stated otherwise, all examples in Dyalog documentation assume that `⎕IO` and `⎕ML` are both 1.

Various icons are used in this document to emphasise specific material.

General note icons, and the type of material that they are used to emphasise, include:

Hints, tips, best practice and recommendations from Dyalog Ltd.

Material of particular significance or relevance.

Legacy information pertaining to behaviour in earlier releases of Dyalog or to functionality that still exists but has been superseded and is no longer recommended.

Warnings about actions that can impact the behaviour of Dyalog or have unforeseen consequences.

A full list of the platforms on which Dyalog version 19.0 is supported is available at https://www.dyalog.com/dyalog/current-platforms.htm. Although the Dyalog programming language is identical on all platforms, differences do exist in the way some functionality is implemented and in the tools and interfaces that are available. Differences in behaviour between operating systems are identified with the following icons (representing macOS, Linux, Microsoft Windows and UNIX respectively):

2 Installation

2.1 Pre-requisites

See Microsoft's .NET webpages (https://dotnet.microsoft.com/) for information on whether the version of macOS/Linux/Microsoft Windows that you are running supports .NET.

.NET is not available for IBM AIX and is not supported on the Raspberry Pi models Zero, 1 or 2.

The Dyalog version 19.0 .NET interface requires .NET version 8.0 or later – it does not work with earlier versions of .NET.

The .NET interface only works with the Unicode edition of Dyalog; the Classic edition is not supported.

Once .NET has been successfully installed (see *Section 2.1.1*) no further installation is required to use the Dyalog .NET interface.

Exporting APL code to .NET assemblies is only supported on 64-bit versions of Dyalog.

2.1.1 Installing .NET

.NET can be downloaded from https://dotnet.microsoft.com/download – download the appropriate .NET SDK and install it according to Microsoft's instructions.

The default installation directory depends on the platform and installation method. Dyalog Ltd recommends that .NET is installed in the following platform-dependent directories:

- /usr/local/share/dotnet on macOS
- /usr/share/dotnet on Linux and Raspberry Pi

- C:\Program Files\dotnet on 64-bit Microsoft Windows
- C:\Program Files (x86)\dotnet on 32-bit Microsoft Windows

If you decide not to install .NET in the default directory, then you need to set the DOTNET_ROOT environment variable to point to your installation location before you start Dyalog. This is a Microsoft variable, not a Dyalog-specific one, so cannot be set in Dyalog's configuration files. See Microsoft's documentation for instructions on how to do this (https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-environment-variables).

On Raspberry Pi Bookworm, do not use the Microsoft-supplied `dotnet-install.sh` script as the resulting .NET installation cannot be used.

EXAMPLE

This example shows the steps taken on Linux to download the runtime to **/tmp/dotnet-runtime-8.0.0-linux-x64.tar.gz** – following these instructions it should not be necessary to define DOTNET_ROOT.

```
sudo mkdir -p /usr/share/dotnet
cd /usr/share/dotnet
sudo tar -zxvf /tmp/dotnet-runtime-8.0.0-linux-x64.tar.gz
sudo  /usr/share/dotnet/dotnet /usr/bin/dotnet
```

This is only an example of code that worked on a specific configuration in our tests; the latest instructions in Microsoft's .NET documentation should always be followed.

2.2 Files Installed with Dyalog

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

- **Dyalog.Net.Bridge.dll** – the interface library through which all calls between Dyalog and .NET are processed.
- **Dyalog.Net.Bridge.Host.<operating system>.dll** – auxiliary file
- **nethost.dll** – auxiliary file
- **Dyalog.Net.Bridge.deps.json** – auxiliary file
- **Dyalog.Net.Bridge.runtimeconfig.json** – auxiliary file

2.3 Enabling the .NET Interface

The .NET interface is enabled when the DYALOG_NETCORE configuration parameter is set to 1; this is the default setting on Linux (including the Raspberry Pi) and macOS. On Microsoft Windows the default setting is 0 for backwards compatibility (a setting of 0 enables the .NET Framework interface).

The .NET interface and .NET Framework interface cannot be enabled simultaneously.

For information on how to set configuration parameters, see the appropriate *Dyalog for <operating system> Installation and Configuration Guide*. To check the value of DYALOG_NETCORE, enter the following when in a Session:

```
+2 ⎕NQ'.' 'GetEnvironment' 'DYALOG_NETCORE'
```

If the result is `1` (or empty on Linux/macOS), then the .NET interface is enabled.

2.4 Verifying the Installation

Dyalog Ltd recommends that the following command is run at the start of any application that will use .NET:

```
      r←2250⌶⍬
```

This command identifies the state of the .NET interface while attempting to suppress all associated error messages (for more information on `2250⌶`, see the *Dyalog APL Language Reference Guide*):

- If `r≡1 1 ''` then the .NET interface should work
- If `r≡2 1 ''` then the .NET Framework interface should work (for more information see the *Dyalog for Microsoft Windows .NET Framework Interface Guide*)

For any other value of `r`, the interface will not work. An indication of why the interface is not working might be given in error messages in the status/Session window or `r[3]`.

If the interface is not working correctly, then:

- ensure that .NET has been installed according to Microsoft's .NET documentation (https://docs.microsoft.com/en-gb/dotnet/).
- check that DOTNET_ROOT is correctly set
- check that DYALOG_NETCORE is correctly set (that is, not set to 0)

If everything has been installed and enabled correctly, then the version of .NET in use will be returned by the following statement:

```
      ⎕USING←'System' ⋄ Environment.Version
```

3 .NET Classes

.NET conforms to Microsoft's Common Type System. This comprises a set of data types, permitted values and permitted operations that define the rules by which different languages can interact with one another – all co-operating languages that use these types can have their operations and values checked (by the Common Language Runtime) at runtime. .NET also provides its own built-in class library that provides all the primitive data types, together with higher-level classes that perform useful operations.

.NET classes are implemented as part of the Common Type System. *Types* include interfaces, value types and classes. .NET provides built-in primitive types as well as higher-level types that are useful in building applications. A *class* is a subset of Type (distinct from interfaces and value types) that encapsulates a particular set of methods, events and properties. The word *object* is usually used to refer to an *instance* of a class. An object is typically created by calling the system function `⎕NEW` with the class as the first element of the argument. An *assembly* is a file that contains all of the code and metadata for one or more classes. Assemblies can be dynamic (created in memory as needed) or static (files on disk). In this document, "assembly" refers to a file (usually with a **.dll** extension) on disk. Classes support inheritance, in that every class (but one) is based on a *base class*.

Through the use of instances of .NET classes, Dyalog gains access to a huge amount of component technology that is provided by .NET; the benefits of this approach include enhanced reliability, software management, code reusage and reduced maintenance.

3.1 Locating .NET Classes and Assemblies

.NET assemblies and the classes they contain are generally self-contained independent entities (although they can be based upon classes in other assemblies). This means that a class can be installed by copying the assembly file onto hard disk and uninstalled by erasing the file.

Microsoft supplies a tool for browsing .NET class libraries called **ildasm.exe** (Intermediate Language Disassembler). On Microsoft Windows, ILDASM has a GUI front end; it can be found in the .NET SDK and is distributed with Visual Studio. For other platforms, ILDASM is available as a command line tool that can be downloaded from https://www.nuget.org/packages/runtime.linux-x64.Microsoft.NETCore.ILDAsm/.

Although classes are arranged physically into assemblies, they are also arranged logically into namespaces. These are not related to Dyalog's namespaces and, to avoid confusion, are referred to in this document as .NET namespaces.

A single .NET namespace can map onto a single assembly. For example, the .NET namespace **System.IO** is contained in an assembly named **System.IO.FileSystem.dll**. However, a .NET namespace can be implemented by more than one assembly, removing the one-to-one-mapping between .NET namespaces and assemblies. For example, the main top-level .NET namespace, **System**, spans a number of different assembly files.

Within a single .NET namespace there can be numerous classes, each with its own unique name. The full name of a class is the name of the class prefixed by the name of the .NET namespace and a dot (the namespace name can also be delimited by dots). For example, the full name of the **DateTime** class in the .NET namespace **System** is **System.DateTime**. Any number of different versions of an assembly can be installed on a single computer, and there can be multiple .NET namespaces with the same name, implemented in different sets of assembly files.

To use a .NET class, it is necessary to tell the system to load the assembly in which it is defined. In many languages (including C#) this is done by supplying the *names* of the assemblies. To avoid having to refer to full class names, C# allows the .NET namespace prefix to be elided. In this case, the programmer must declare a list of .NET namespaces with `using` declaration statements. This list is then used to resolve unqualified class names referred to in the code. In either language, when the compiler encounters the unqualified name of a class, it searches the specified .NET namespaces for that class. In Dyalog, this mechanism is implemented by the `⎕USING` system variable. `⎕USING` performs the same two tasks that `using/imports` declarations provide in C#; that is, to give a list of .NET namespaces to be searched for unqualified class names and to specify the assemblies that 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 assembly either with a file name (the string ends with the extension **.dll**) or with an assembly name. If an assembly name is given, standard .NET rules are used to locate the assembly.

It is convenient to treat .NET namespaces and assemblies in pairs. For example, the `System.IO` namespace is located within the `System.IO.FileSystem` assembly.

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

If `⎕USING` is empty (`⎕USING←0⍴⊂''`), then Dyalog does not search for .NET classes to resolve names that would otherwise give a `VALUE ERROR`.

Assigning a simple character vector to `⎕USING` is equivalent to setting it to the enclose of that vector. The statement (`⎕USING←''`) does not empty `⎕USING`, but rather sets it to a single empty element, which gives access to the **System.Runtime** and **System.Private.CoreLib** assembly files without a namespace prefix.

3.2 Using .NET Classes

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

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

If the argument is a 2-element vector, an instance of the class is created using the constructor overload (see *Section 3.2.1*) whose argument matches the disclosed second element.

EXAMPLE

Creating an instance of the `DateTime` class requires an argument with two elements: (the class and the constructor argument; in this example the constructor argument is a 3-element vector representing the date). Many classes provide a default constructor that takes no arguments. From Dyalog , the default constructor is called by calling ⎕NEW with only a reference to the class in the argument.

To create a DateTime object whose value is 30 April 2008:

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

Alternatively, to use fully-qualified class names, one of the elements of ⎕USING must be an empty vector:

```
      ⎕USING←,⊂''
      mydt←⎕NEW System.DateTime (2008 4 30)
```

In both cases, the result of ⎕NEW is a reference to the newly created instance:

```
      ⎕NC ⊂'mydt'
9.2
```

When a reference to a .NET object is formatted, APL calls its ToString method to obtain a useful description or identification of the object (this topic is discussed in more detail in *Section 3.2.3*):

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

3.2.1 Constructors and Overloading

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

This concept, which is known as *overloading*, may seem somewhat alien to the APL programmer, who will be accustomed to defining functions that accept an arbitrary array. However, type checking, which is fundamental to .NET, 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 creating an instance of a class in C#, the `new` operator is used. At compile time, this is mapped to the appropriate constructor overload by matching the user-supplied parameters to the various forms of the constructor. A similar mechanism is implemented in Dyalog by the `⎕NEW` system function.

3.2.2 Resolving References to .NET Objects

When Dyalog executes an expression such as

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

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

The first time that Dyalog encounters a reference to a non-existent name (that is, a name that would otherwise generate a `VALUE ERROR`), it searches the .NET namespaces/assemblies specified by `⎕USING` for a .NET class of that name. If found, the name (in this case, `System.DateTime`) is recorded in the APL symbol table with a name class of 9.6 and is associated with the corresponding .NET Type. If not found, then `VALUE ERROR` is reported as usual. This search ONLY takes place if `⎕USING` has been assigned a non-empty value.

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

If `⎕NEW` is called with only a class as argument, then Dyalog attempts to call the overload of its constructor that is defined to take no arguments. If no such overload exists, then the call fails with a `LENGTH ERROR`.

If `⎕NEW` is called with a class as argument and a second element, then Dyalog calls the version of the constructor whose parameters match the second element supplied to `⎕NEW`. If no such overload exists, then the call will fail with either a `LENGTH ERROR` or a `DOMAIN ERROR`.

3.2.3 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`, which provides a default implementation. Many .NET classes provide their own `ToString` that overrides the one inherited from `System.Object` and returns a useful representation of 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. The monadic *format* function (`⍕`) and monadic `⎕FMT` have been extended to provide the same result and provide a shorthand method to call `ToString`. The default `ToString` supplied by `System.Object` returns the name of the object's Type. For a particular object in the namespace, this can be changed using the system function `⎕DF`.

EXAMPLE

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

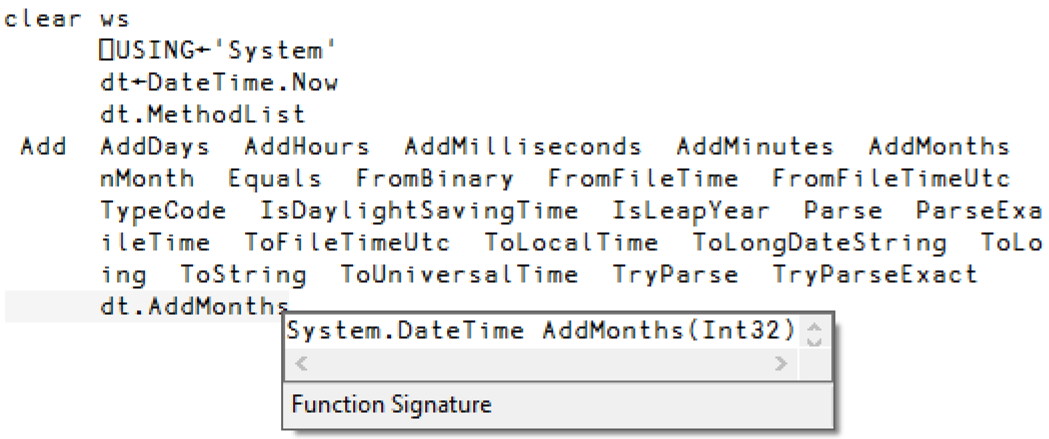
The type of an object can be obtained using the `GetType` method, which is supported by all .NET objects:

```
      z.GetType
System.DateTime
```

3.2.3.1 Value Tips for External Functions

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

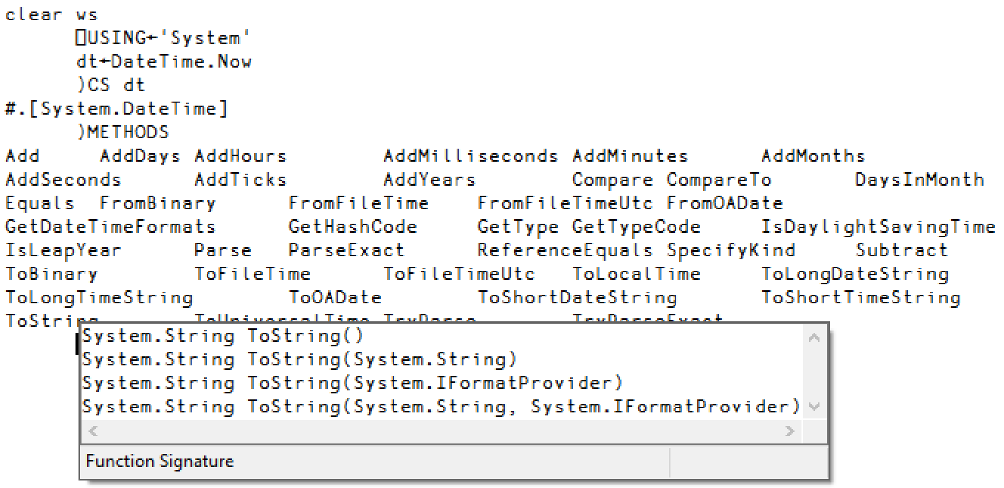
For example, *Figure 3-1* shows the mouse hovered over the external function `dt.AddMonths`, which reveals that it requires a single integer as its argument.



```
clear ws
      ⎕USING←'System'
      dt←DateTime.Now
      dt.MethodList
 Add  AddDays  AddHours  AddMilliseconds  AddMinutes  AddMonths
      nMonth  Equals  FromBinary  FromFileTime  FromFileTimeUtc
      TypeCode  IsDaylightSavingTime  IsLeapYear  Parse  ParseExa
      ileTime  ToFileTimeUtc  ToLocalTime  ToLongDateString  ToLo
      ing  ToString  ToUniversalTime  TryParse  TryParseExact
      dt.AddMonths
              System.DateTime AddMonths(Int32)
              Function Signature
```

Figure 3-1: *Function signature – single integer argument*

If an external function provides more than one signature, then they are all shown in the Value Tip (see *Figure 3-2*; the function `ToString` has four different overloads.



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

```
System.String ToString()
System.String ToString(System.String)
System.String ToString(System.IFormatProvider)
System.String ToString(System.String, System.IFormatProvider)
```

***Figure 3-2:** Function signature – multiple arguments*

3.2.4 Disposing of .NET Objects

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

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

Deallocated memory might not be reused immediately and might never be reused, depending on the algorithms used by the CLR garbage disposal.

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

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

This programming convenience is provide in Dyalog by the `:Disposable ... :EndDisposable` control structure. For more information on this control structure, see the *Dyalog Programming Reference Guide*.

3.3 Advanced Techniques

3.3.1 Shared Members

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

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

Example

```
      ⎕USING←,⊂'System'

      DateTime.Now
18/03/2020 11:14:05

      DateTime.IsLeapYear 2000
1
```

3.3.2 APL Language Extensions for .NET Projects

.NET provides a set of standard operators (methods) that are supported by certain classes, for example, methods to add and subtract .NET objects and methods to compare two .NET objects.

Example 1: DateTime – Adding and subtracting

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

```
       DT3←System.DateTime.Now
       DT3
15/02/2024 10:35:35
```

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

      DateTime.op_Addition DT3 TS
15/02/2024 11:36:36

      DateTime.op_Subtraction DT3 TS
15/02/2024 09:34:34
```

EXAMPLE 2: DATETIME – COMPARING

The `op_Equality` and `op_Inequality` operators compare two `DateTime` objects:

```
      DT1←⎕NEW DateTime (2024 4 30)
      DT2←⎕NEW DateTime (2024 1 1)

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

Some corresponding APL primitive functions have been extended to accept .NET objects as arguments and call these standard .NET methods internally. The methods and the corresponding APL primitives that are currently available are shown in *Table 3-1*.

Table 3-1: *.NET methods and their APL primitive function equivalents*

.NET Method	APL Primitive Function
`op_Equality`	= and ≡
`op_Inequality`	≠ and ≢

This means that Example 2 becomes:

```
      DT1←⎕NEW DateTime (2024 4 30)
      DT2←⎕NEW DateTime (2024 1 1)

      ⍝ Is DT1 equal to DT2?
      DT1 = DT2
0
```

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

3.3.3 Exceptions

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

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

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

```
      ⎕USING←'System'
      DT←⎕NEW DateTime (100000 0 0)
EXCEPTION: Year, Month, and Day parameters describe an un-
representable DateTime.
      DT←⎕NEW DateTime (100000 0 0)
        ∧
      ⎕EN
90
      ⎕EXCEPTION.Message
Year, Month, and Day parameters describe an un-representable
DateTime.

      ⎕EXCEPTION.Source
System.Private.CoreLib

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

The result of `⎕EXCEPTION.StackTrace` can depend on the exact version of .NET – your result might look different, but if it includes `System.DateTime..ctor(Int32 year, Int32 month, Int32 day)` then it is showing the correct exception for this example.

3.3.4 Specifying Overloads

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

Occasionally it might be desirable to override this mechanism and explicitly specify which overload to use. This can be done by calling the function and specifying the Variant operator `⍠` with the `OverloadTypes` option. This takes an array of references to .NET types, of the same length as the number of parameters to the function.

EXAMPLE

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

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

or (more simply):

```
      (foo ⍠Double)val
```

where `Double` is a reference to the .NET type `System.Double`.

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

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

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

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

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

3.3.4.1 Overloaded Constructors

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

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

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

3.4 Example Usage

3.4.1 Directory and File Manipulation

The .NET namespace System.IO (in the System.IO.FileSystem assembly) 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, System.IO.FileSystem'
      dir←'C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode'
      d←⎕NEW DirectoryInfo (⊂dir)
```

where d is an instance of the Directory class, corresponding to the directory **[DYALOG]**.

[DYALOG] refers to the directory in which Dyalog is installed; this example assumes **[DYALOG]** to be **C:/Program Files/Dyalog/Dyalog APL-64 19.0 Unicode**.

The GetFiles method returns a list of files (more precisely, FileInfo objects) that represent each of the files in the directory. Its optional argument specifies a filter. For example:

```
      d.GetFiles ⊂'*.exe'
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\dyaedit.exe
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\dyalog.exe
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\dyalogc.exe
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\
dyalogc64_unicode.exe  C:\Program Files\Dyalog\Dyalog APL-64
19.0 Unicode\dyalogrt.exe  C:\Program Files\Dyalog\Dyalog APL-64
19.0 Unicode\dyascript.exe
```

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

```
      (d.GetFiles ⊂'*.exe').Name
dyaedit.exe  dyalog.exe  dyalogc.exe  dyalogc64_unicode.exe
dyalogrt.exe  dyascript.exe
```

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

```
     (d.GetFiles ⊂'*.exe').CreationTime
 08/02/2024 20:51:24  08/02/2024 20:50:06  08/02/2024 ...
```

Calling the GetFiles overload that does not take any arguments (from Dyalog by supplying an argument of ⍬) returns a complete list of files:

```
      files←d.GetFiles ⍬
      files
C:\Program Files\Dyalog\Dyalog APL-64 19.0
Unicode\aplunicd.ini...
```

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

```
      files,[1.5]files.CreationTime
C:\...\...Unicode\aplunicd.ini              08/02/2024 20:12:02
C:\...\...Unicode\bridge190-64_unicode.dll  08/02/2024 20:47:36
...
```

3.4.2 Sending an Email

The .NET namespace System.Net.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 SmtpClient class.

Example

This example will only work if your computer is configured to allow you to send email.

```
∇ recip Send(subject msg);⎕USING;from;mail;to;builder;client;
                                      FROM_ADDRESS; EMAIL_SERVER
  ⎕USING←'System.Net.Mail,System.Net.Mail'

  FROM_ADDRESS←'someone@somewhere.com'
  EMAIL_SERVER←'mail.somwhere.com'
```

```
   from←⎕NEW MailAddress(⊂FROM_ADDRESS)
   to←⎕NEW MailAddress(recip '')
   mail←⎕NEW MailMessage (from to)
   mail.Body←msg
   mail.Subject←subject
   client←⎕NEW SmtpClient (⊂EMAIL_SERVER)
   client.Send mail
∇
```

This could then be called as follows:

```
'prime.minister@gov.uk' Send ('subject' ('line1' 'line2'))
```

3.4.3 Web Scraping

.NET provides a range of classes for accessing the internet from a program. This section works through an example that shows how to read the contents of a web page. It is complicated, but realistic (for example, 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.

Start by defining `⎕USING` so that it specifies all of the necessary .NET namespaces and assemblies:

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

The `WebRequest` class in the `System.Net` .NET namespace implements .NET's request/response model for accessing data from the internet. For this example, a `WebRequest` object needs to be associated with the URI http://www.dyalog.com (`WebRequest` is an example of a static class – its methods can be used without creating instances of it):

```
      wrq←WebRequest.Create ⊂'http://www.dyalog.com'
```

Potentially confusingly, if the URI specifies a protocol of "http://" or "https://", an object of type `HttpWebRequest` is returned rather than a simple `WebRequest`. The effect of this is that, at this stage, `wrq` is an `HttpWebRequest` object.

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

The HttpRequest class has a GetResponse method that returns a response from an internet resource. Although it is not yet HTML, 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.Http.HttpConnection+ChunkedEncodingReadStream
```

However, the Stream class is designed for byte input and output; what is needed in this example 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, create a new instance of a StreamReader by passing it the Stream as a parameter:

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

Finally, use the ReadToEnd method of the StreamReader to get the contents of the page:

```
      s←rdr.ReadToEnd
      ⍴s
20295
```

To avoid running out of connections, it is necessary to close the stream:

```
      str.Close
```

3.5 Enumerations

An enumeration is a set of named constants that can apply to a particular operation. For example, when opening 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 specifies this. If this is implemented using an enumerated constant, then the parameter can be one of a specific set of (typically) integer values, for example, 1 = read, 2 = write, 3 = read and write. However, to avoid using ambiguous numbers in code, it is conventional to use names to represent particular values. These are known as *enumerated constants* or, more simply, as *enums*.

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

Typically, an enumerated constant would be used as a parameter to a method or to specify the value of a property. For example, the `DayOfWeek` property of the `DateTime` object returns a value of Type `System.DayOfWeek` (it is incidental that both the Type and property are called `DayOfWeek`):

```
      ⎕USING←'' 'System'
      cal←⎕NEW DateTime(1981 09 23)
      cal.DayOfWeek
Wednesday
      cal.DayOfWeek.GetType
System.DayOfWeek
      System.DayOfWeek.⎕NL ¯2
Friday  Monday  Saturday  Sunday  Thursday  Tuesday  Wednesday
```

The function `System.Convert.ToBase64String` has some constructor overloads that take an argument of Type `System.Base64FormattingOptions`, which is an enum:

```
    System.Convert.ToBase64String
System.String ToBase64String(Byte[])
...
      System.Base64FormattingOptions.⎕NL ¯2
InsertLineBreaks  None
```

Hence:

```
      (⎕UCS 13 )∊ System.Convert.ToBase64String(⊂⍳100) System.
                          Base64FormattingOptions.InsertLineBreaks
1
      (⎕UCS 13 )∊ System.Convert.ToBase64String(⊂⍳100) System.
                                      Base64FormattingOptions.None
0
```

An enum has a value that can be used in place of the enum itself when such usage is unambiguous. For example, the `System.Base64FormattingOptions.InsertLineBreaks` enum has an underlying value of `1`:

```
      Convert.ToInt32 Base64FormattingOptions.InsertLineBreaks
1
```

This means that the scalar value `1` can be used as the second parameter to `ToBase64String`:

```
      (⎕UCS 13 )∊ System.Convert.ToBase64String(⍳100) 1
1
```

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

3.6 Handling Pointers with Dyalog.ByRef

Certain .NET methods take parameters that are pointers, for example, the `DivRem` method that is provided by the `System.Math` class. This method performs an integer division, returning the quotient as its result, and the remainder in an address specified as a pointer by the calling program.

APL does not have a mechanism for dealing with pointers, so Dyalog provides a .NET class for this purpose. This is the `Dyalog.ByRef` class, which is provided in **Dyalog.Net.Core.Bridge.dll** (which is automatically loaded by Dyalog).

To gain access to the `Dyalog` .NET namespace, it must be specified by `⎕USING`. The assembly (DLL) from which it is obtained (the **Dyalog.Net.Bridge.dll** file) does not need to be specified as it is automatically loaded when Dyalog starts:

```
      ⎕USING←'System.IO,System.IO.FileSystem' 'Dyalog'
```

The `Dyalog.ByRef` class represents a pointer to an object of type `System.Object`. It has a number of constructors, some of which are used internally by Dyalog. Only two of these are of particular interest – the one that takes no parameters, and the one that takes a single parameter of type `System.Object`. The former is used to create an empty pointer; the latter to create a pointer to an object or some data.

For example, to create an empty pointer:

```
      ptr1←⎕NEW ByRef
```

or, to create pointers to specific values:

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

As a single parameter is required, it must be enclosed if it is an array with several elements. Alternatively, the parameter can 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
```

If the `Value` property is referenced without first setting it, a `VALUE ERROR` is returned:

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

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

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

```
      remptr←⎕NEW ByRef
      remptr.Value
VALUE ERROR
      remptr.Value
     ^

      Math.DivRem 311 99 remptr
3
      remptr.Value
14
```

Sometimes a .NET method can 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, the `Dyalog.ByRef` class can be used to call this method. For example, the `System.IO.FileStream` class contains a `Read` method that populates its first argument with the bytes in the file:

```
      ⎕USING←'System.IO' 'Dyalog' 'System'
      fs←⎕NEW FileStream ('c:\tmp\jd.txt' FileMode.Open)
      fs.Length
25

      fs.Read(arg←⎕NEW ByRef,(⊂25⍴0))0 25
25

      arg.Value
104 101 108 108 111 32 102 114 111 109 32 106 111 104 110 32 100
97 105 110 116 114 101 101 10
```

3.7 DECF Conversion

Incoming .NET data types `System.Decimal` and `System.Int64` are converted to 126-bit decimal numbers (DECFs). This conversion is performed independently of the value of `⎕FR`.

To perform arithmetic on values imported in this way, set `⎕FR` to 1287, at least for the duration of the calculations.

4 APL Source Files

APL Source files contain definitions (the “source”) of one or more named APL objects, that is, functions, operators, namespaces, classes, interfaces and arrays. They cannot contain anything else. They are not workspace-oriented (although you can call workspaces from them) but are simply character files containing function bodies and expressions. This means that they would be valid right arguments to `2 ⎕FIX`.

APL Source files employ Unicode encoding, so you need a Unicode font with APL symbols, such as APL385 Unicode, to create or view them. They can be viewed and edited using any character-based editor that supports Unicode text files.

To enter Dyalog APL symbols into an APL Source file, you need the Dyalog Input Method Editor (IME) or other APL compatible keyboard. The Dyalog IME can be configured from the **Dyalog Configuration** dialog box. You can change the associated **.DIN** file or there are various other options. APL Source files can also be edited using Microsoft Word, although they must be saved as text files without any Word formatting. For more information, see the *Dyalog for Microsoft Windows Installation and Configuration Guide*.

APL Source files can be identified by the **.apl** file extension. This can either specify .NET classes or represent an APL application in a text source format (as opposed to a workspace format). Such applications do not necessarily require .NET. The **.apl** file extension can, optionally, be further categorised, for example:

- **.apla** files contain array definitions
- **.aplc** files contain class definitions
- **.aplf** files contain function definitions
- **.apli** files contain interface definitions
- **.apln** files contain namespace definitions
- **.aplo** files contain operator definitions

4.1 The Dyalog .NET Compiler

APL Source files are compiled into executable code by the Dyalog .NET Compiler, which is called **dyalogc.exe**.

By default, **dyalogc.exe** compiles to .NET. If the `-framework` option is set, it will instead compile to .NET Framework.

For backwards compatibility, the Dyalog .NET Compiler is also distributed on Microsoft Windows with the names identified in *Table 4-1*.

Table 4-1: *Version-specific Dyalog .NET Compilers*

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

The Dyalog .NET Compiler can be used to:

- compile APL Source files into a workspace (**.dws**) – this can subsequently be run using **dyalog.exe** or **dyalogrt.exe**.
- compile APL Source files into a .NET class (**.dll**) – this can subsequently be used by any other .NET-compatible host language, such as C#.

The script is designed to be run from a command prompt. Navigate to the appropriate directory and type `dyalogc /?` to query its usage; the following output is displayed (the output displayed here is for Microsoft Windows; the command line options are not all applicable on other platforms):

```
c:/Program Files/Dyalog/Dyalog APL-64 19.0 Unicode>dyalogc /?
Dyalog .NET Compiler 64 bit. Unicode Mode. Version 19.0.48745.0
Copyright Dyalog Ltd 2000-2024

dyalogc.exe command line options:

-?                      Usage
-r:<file>               Add reference to assembly
-o[ut]:<file>           Output file name
-res:<file>             Add resource to output file
-icon:<file>            File containing main program icon
-q                      Operate quietly
-v                      Verbose
-v2                     More verbose
-s                      Treat warnings as errors
-nonet                  Creates a binary that does not use Microsoft
                        .NET
```

```
-net                    Creates a binary that targets .NET Version >=5
-framework              Creates a binary that targets .NET Framework
-runtime                Build a non-debuggable binary
-t:library              Build .NET library (.dll)
-t:workspace            Build dyalog workspace (.dws)
-t:nativeexe            (Windows only) Build native executable (.exe).
                        Default
-t:standalonenativeexe  (Windows only) Build native executable (.exe).
                        Default
-lx:<text>              (Windows only) Specify entry point (Latent
                        Expression)
-cmdline:<text>         Specify a command line to pass to the
                        interpreter
-nomessages             (.NET Framework only) Process does not use
                        windows messages. Use when creating a process
                        to run under IIS
-console|c              Creates a console application
-multihost              Support multi-hosted interpreters
-unicode                Creates an application that runs in a Unicode
                        interpreter
-wx:[0|1|3]             Sets ⎕WX for default code
-a:file                 (.NET Framework only) Specifies a JSON file
                        containing attributes to be attached to the
                        binary
-i:Process              (.NET Framework only) Set the isolation mode of
                        a .NET Assembly
-i:Assembly             (.NET Framework only) Set the isolation mode of
                        a .NET Assembly
-i:AppDomain            (.NET Framework only) Set the isolation mode of
                        a .NET Assembly
-i:Local                (.NET Framework only) Set the isolation mode of
                        a .NET Assembly
```

The –a option specifies the name of a JSON file that contains assembly information. For example:

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

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

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

4.2 Creating an APL Source File

Conceptually, the simplest way to create an APL Source file is with a text editor, although you can use many other tools, for example, Microsoft Visual Studio. It is important to ensure that the file is saved with the appropriate file extension (see *Section 4*).

4.3 Copying Code from the Dyalog Session

You might find it easy to write APL code using the Dyalog Session's function/class editor, or you might already have code in a workspace that you want to copy into an APL Source file. In either case, you can transfer code from the Session into an appropriate text editor using the clipboard.

When pasting APL code from the Session into a text editor, line numbers can be included; although this is allowed, it is not recommended in APL Source files.

4.4 General Principles of APL Source Files

The layout of an APL Source file differs according to what it defines. However, within the APL Source file, the code layout rules are basically the same.

An APL Source 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 Dyalog .NET Compiler. These all start with a colon symbol (`:`) in the manner of control structures. For example, the `:Namespace` statement tells the Dyalog .NET 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 configure 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 APL Source file and executed by the Dyalog .NET Compiler in the order in which they appear.

The statements are executed at compile time, and not at run-time, and can, therefore, only be used for initialisation.

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

The Dyalog .NET Compiler will execute any valid APL expression that you include. However, the results might not be useful and could 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 (`∇`) characters. These are fixed by the Dyalog .NET Compiler using `⎕FX`. Line numbers and white space formatting are ignored.

4.5 Creating Programs (.exe) with APL Source Files

This section is specific to the Microsoft Windows operating system only.

The following examples, which illustrate how you can create an executable program (**.exe**) directly from an APL Source file, can be found in the **[DYALOG]/Samples/bound_exe** directory. The examples require write access to successfully build the samples, therefore Dyalog Ltd recommends copying the **[DYALOG]/Samples/bound_exe** directory to somewhere you have write access.

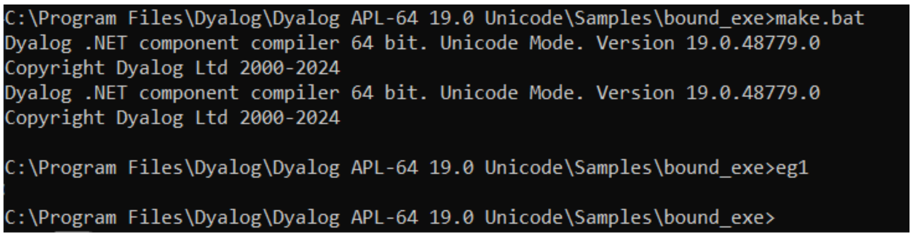
Example: Simple GUI

The **eg1.apln** APL Source file illustrates the simplest possible GUI application that displays a message box containing the string "Hello World":

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

The code must be contained within `:NameSpace` and `:EndNamespace` statements, and must define a ⎕LX either within the APL Source file itself or as a parameter to the `dyalogc` command. In this example, ⎕LX is defined within the APL Source file.

This is compiled to a Windows executable (**.exe**) using **make.bat** and run from the same command window (see *Figure 4-1* and *Figure 4-2*).



```
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\bound_exe>make.bat
Dyalog .NET component compiler 64 bit. Unicode Mode. Version 19.0.48779.0
Copyright Dyalog Ltd 2000-2024
Dyalog .NET component compiler 64 bit. Unicode Mode. Version 19.0.48779.0
Copyright Dyalog Ltd 2000-2024

C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\bound_exe>eg1

C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\bound_exe>
```

Figure 4-1:** Compiling and running APL Source file **eg1.apln

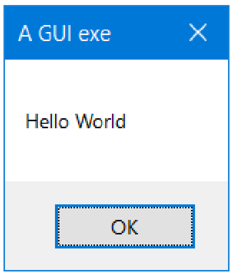


***Figure 4-2:** "Hello World" Message Box (**eg1.exe**)*

The resulting executable can be associated with a desktop icon, and will run without a command prompt window. Any default APL output that would normally be displayed in the session window will be ignored.

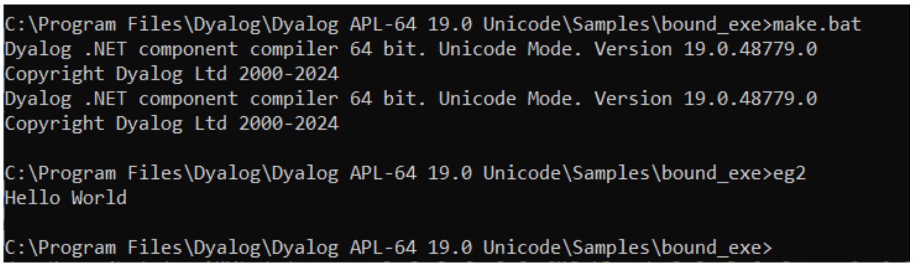
EXAMPLE: SIMPLE CONSOLE

The **eg2.apln** APL Source file illustrates the simplest possible application that displays the text "Hello World".:

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

The code must be contained within `:NameSpace` and `:EndNamespace` statements, and must define a `⎕LX` either in the APL Source file itself or as a parameter to the `dyalogc` command. In this example, `⎕LX` is defined within the APL Source file.

This is compiled to a Windows executable (**.exe**) using **make.bat** and run from the same command window (see *Figure 4-3*). The `/console` flag in **make.bat** instructs the Dyalog .NET 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 is instead displayed in the command window from which the program was run.



```
C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\bound_exe>make.bat
Dyalog .NET component compiler 64 bit. Unicode Mode. Version 19.0.48779.0
Copyright Dyalog Ltd 2000-2024
Dyalog .NET component compiler 64 bit. Unicode Mode. Version 19.0.48779.0
Copyright Dyalog Ltd 2000-2024

C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\bound_exe>eg2
Hello World

C:\Program Files\Dyalog\Dyalog APL-64 19.0 Unicode\Samples\bound_exe>
```

Figure 4-3:** Compiling and running APL Source file **eg2.apln

4.5.1 Defining Namespaces

At least one namespace must be specified in an APL Source file. Namespaces are specified in an APL Source file using the `:Namespace` and `:EndNamespace` statements. Although you can use `⎕NS` and `⎕CS` within functions inside an APL Source file, you should not use these system functions outside function bodies; such use is not prevented, but the results will be unpredictable.

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

`: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 within that namespace. Similarly, all assignments define variables inside that namespace.

5 Writing .NET Classes

Dyalog allows you to build new .NET classes, components and controls:

- A component is a class with emphasis on clean-up and containment, and implements specific interfaces.
- A control is a component with user interface capabilities.

.NET classes created by Dyalog can be hosted by any application or programming language that supports .NET.

With one exception, every .NET class inherits from exactly one base class. This means that it begins with all of the behaviour of the base class, in terms of the base class properties, methods and events. You can 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.

5.1 Assemblies, Namespaces, and Classes

To create a .NET class in Dyalog, create a standard APL class and export the workspace as a .NET assembly (***.dll**).

Exporting APL code to .NET assemblies is only supported on 64-bit versions of Dyalog.

.NET classes are organised in .NET namespaces. If you wrap your class (or classes) within an APL namespace, the name of that namespace will be used to identify the name of the corresponding .NET namespace in your assembly.

If a class is to be based upon a specific .NET class, then the name of that .NET class must be specified as the base class in the `:Class` statement, and the `:Using` statements must correctly locate the base class. Otherwise, the class is assumed to be based on `System.Object`. If you use any .NET types within your class, you must ensure that these too are located by `:Using`.

Once you have defined the functionality of your .NET classes, you can save them in an assembly. This is achieved in one of the following ways:

- Select **Export...** from the Session's **File** menu. You will be prompted to specify the directory and name of the assembly (DLL), and it will then be created and saved.
- Use the Bind method (see *Section 5.1.1*).
- Use the Dyalog .NET Compiler (see *Section 5.4*).

Your .NET class is now ready for use by any .NET development environment, including APL.

When a Dyalog .NET class is invoked by a host application, it automatically loads the Dyalog DLL, which is the developer/debug or run-time dynamic link library version of Dyalog. The Dyalog .NET class, and all the Dyalog DLLs on which it depends, reside in the same directory as the host program.

If you want to include a Dyalog .NET class in a Visual Studio application, Dyalog Ltd recommends that you add the bridge DLL as a reference in a Visual Studio .NET project.

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

5.1.1 The Bind Method

The `Bind` method is described in the *Dyalog for Microsoft Windows Object Reference Guide*. A subset of the `Bind` method can be used on any supported platform to export .NET assemblies. Specifically, the expression:

```
      2 ⎕NQ '.' 'Bind' <filename> 'Library'
```

creates a .NET assembly (in `<filename>`) that contains the APL code in the classes in the active workspace

This use of the `Bind` method is similar to selecting **File** > **Export...** in the Session.

5.2 Tutorial

All the examples in this tutorial are to be executed as simple console applications written in C#.

The code for all of the examples is provided in the **[DYALOG]/Samples/aplclasses/** directory:

- **aplclassesN/aplclassesN.dws** – workspaces containing the source code for the Dyalog classes
- **aplclassesN/net/project/Program.cs** – the corresponding C# source code for hosting the Dyalog classes.

Each workspace contains a .NET namespace called `APLClasses` which itself contains a single .NET class called `Primitives` that exports a single method called `IndexGen`. When executing each example, the workspace (**aplclassesN.dws** will be exported to the **/net/project/bin/Debug/net8.0** sub-directory as a .NET assembly called **aplclassesN.dll**.

The examples in the tutorial require write access to successfully build the samples. Dyalog Ltd recommends copying the **[DYALOG]/Samples/aplclasses** directory to somewhere you have write access; in this tutorial that location will be identified as **<your_dir>**.

To compile the C# source code

1. On the command line, navigate to **<your_dir>/aplclassesN/net**.
2. Run **build** (Linux and macOS)/**build.bat** (Microsoft Windows).
   This invokes the Dyalog script compiler to compile **aplclassesN.dws** to **aplclassesN.dll**, and then invokes the C# compiler to compile the C# source code (**Program.cs**) to produce an executable called **project.exe** in **<your_dir>/aplclassesN/net/project/bin/Debug/net8.0**.

5.2.1 Example 1

Load the **aplclasses1.dws** workspace from **<your_dir>/aplclasses1**, then view the `Primitives` class:

```
      )ED APLClasses.Primitives
```

```
:Class Primitives
:using System
∇R←IndexGen N
:access public
:signature Int32[]←IndexGen Int32
R←⍳N
∇
:EndClass ⍝ Primitives
```

`Primitives` contains one public method/function, called `IndexGen`.

The public characteristics for the exported method are included in the definition of the class and its functions, as specified in the `:Signature` statement. This has the following syntax:

```
:Signature [rslttype←] name [arg1type [arg1name]
                                    [,argNtype [argNname]]*]
```

where:

- `rslttype` is the type of the result returned by the function – in this example, the function returns an array of 32-bit integers
- `name` is the exported name (it can be different from the APL function name but it must be provided) – in the example, the name of the exported method is `IndexGen`
- `argNtype [argNname]` are any arguments are to be supplied, each type-name pair separated from the next by a comma. In this example, the function takes a single integer as its argument.

For more information on `:Signature`, see the *Dyalog Programming Reference Guide*.

When the class is fixed, APL will try to find the .NET data types that have been specified for the result and for the parameters. If one or more of the data types are not recognised as available .NET types, then a warning will be displayed in the status window and APL will not fix the class. If you see such a warning, you have either entered an incorrect data type name, or you have not set `:using` correctly, or some other syntax problem has been detected (for example, the function could be missing a terminating ∇). In this example, the only data type used is `System.Int32`; as `:using System` is included in the definition, `Int32` is correctly located.

In earlier versions of Dyalog, the statements `:Returns` and `:ParameterList` were used instead of `:Signature`. They are still accepted for backwards compatibility reasons, but are considered deprecated.

5.2.1.1 aplclasses1

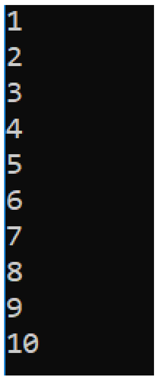
The C# source code (**<your_dir>/aplclasses1/net/project/Program.cs**) can be used to call the Dyalog.NET class. The `using` statements specify the names of .NET namespaces to be searched for unqualified class names. The program creates an object called `apl` of type `Primitives` by calling the `new` 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]);
        }
    }
```

To compile the C# source code

1. On the command line, navigate to **<your_dir>/aplclasses1/net**.
2. Run **build** (Linux and macOS)/**build.bat** (Microsoft Windows).
   This invokes the Dyalog script compiler to compile **aplclasses1.dws** to **aplclasses1.dll**, and then invokes the C# compiler to compile the C# source code (**Program.cs**) to produce an executable called **project.exe** in **<your_dir>/aplclasses1/net/project/bin/Debug/net8.0**.

The output when the program is run is displayed in a console window (see *Figure 5-1*).



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

Figure 5-1: *Program output in console window*

5.2.2 Example 2

In *Section 5.2.1*, APL supplied a default constructor, which was used to create an instance of the Primitives class. It was inherited from the base class (`System.Object`) and called without arguments. This example extends that by adding a constructor that specifies the value of `⎕IO`.

Load the **aplclasses2.dws** workspace from **<your_dir>/aplclasses2**, then view the `Primitives` class:

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

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

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

:EndClass ⍝ Primitives
```

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

5.2.2.1 aplclasses2

The C# source code (**<your_dir>/aplclasses2/net/project/Program.cs**) can be used to call the new version of the Dyalog .NET class:

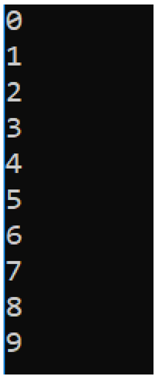
```
using System;
using APLClasses;
public class MainClass
      {
      public static void Main()
            {
            Primitives apl = new Primitives(0);
            int[] rslt = apl.IndexGen(10);

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

The program is the same as in the previous example (see *Section 5.2.1*), except that the code that creates an instance of the `Primitives` class now specifies an argument; in this example, 0.

To compile the C# source code

1. On the command line, navigate to **<your_dir>/aplclasses2/net**.
2. Run **build** (Linux and macOS)/**build.bat** (Microsoft Windows).
   This invokes the Dyalog script compiler to compile **aplclasses2.dws** to **aplclasses2.dll**, and then invokes the C# compiler to compile the C# source code (**Program.cs**) to produce an executable called **project.exe** in **<your_dir>/aplclasses2/net/project/bin/Debug/net8.0**.

The output when the program is run is displayed in a console window (see *Figure 5-2*) – the amended line numbers show the effect of changing the index origin from 1 (the default) to 0.



```
0
1
2
3
4
5
6
7
8
9
```

Figure 5-2: *Program output in console window*

5.2.3 Example 3

The correct .NET behaviour when an APL function fails with an error is to generate an exception; this example shows how this is achieved.

In .NET, 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` can be used to generate an exception. To do this, its right argument should be 90 and its left argument should be an object of type `System.Exception` or an object that inherits from `System.Exception`.

When you create the instance of the `Exception` class, you can specify a string (which will be its `Message` property) containing information about the error.

Load the **aplclasses3.dws** workspace from **<your_dir>/aplclasses3**, then view its improved (compared with that in *Section 5.2.2*) `CTOR` constructor function:

```
     ∇ CTOR IO;EX
[1]    :Implements constructor
[2]    :Access public
[3]    :Signature CTOR Int32 IO
[4]    :If IO∊0 1
[5]      ⎕IO←IO
[6]    :Else
[7]        EX←⎕NEW ArgumentException,⊂⊂'IndexOrigin must be 0 or
                                                                1'
```

```
[8]       EX ⎕SIGNAL 90
[9]     :EndIf
      ∇
```

5.2.3.1 aplclasses3

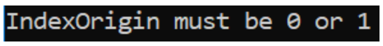
The C# source code (**<your_dir>/aplclasses3/net/project/Program.cs**) contains code to catch the exception and display the exception message:

```
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);
    }
        }
    }
```

To compile the C# source code

1. On the command line, navigate to **<your_dir>/aplclasses3/net**.
2. Run **build** (Linux and macOS)/**build.bat** (Microsoft Windows).
   This invokes the Dyalog script compiler to compile **aplclasses3.dws** to **aplclasses3.dll**, and then invokes the C# compiler to compile the C# source code (**Program.cs**) to produce an executable called **project.exe** in **<your_dir>/aplclasses3/net/project/bin/Debug/net8.0**.

The output when the program is run is displayed in a console window (see *Figure 5-3*).



```
IndexOrigin must be 0 or 1
```

Figure 5-3: *Program output in console window*

5.2.4 Example 4

This example builds on the example in *Section 5.2.3*, and illustrates how you can implement *constructor overloading* by establishing several different constructor functions.

For this example, when a client application creates an instance of the `Primitives` class, is should be able to specify either 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`.

Load the **aplclasses4.dws** workspace from **<your_dir>/aplclasses4**; the new version of the `Primitives` class includes the following additions:

```
     ∇ CTOR1 IO
[1]    :Implements constructor
[2]    :Access public
[3]    :Signature CTOR1 Int32 IO
[4]    CTOR IO 0
     ∇

     ∇ CTOR2 IOML
[1]    :Implements constructor
[2]    :Access public
[3]    :Signature CTOR2 Int32 IO,Int32 ML
[4]    CTOR IOML
     ∇

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

The `:Signature` statements for these three functions show that `CTOR1` is defined as a constructor that takes a single `Int32` parameter and `CTOR2` is defined as a constructor that takes two `Int32` parameters; `CTOR` has no .NET properties defined. In .NET terminology, `CTOR` is not a *private constructor* but rather an internal function that is invisible to the outside world.

Next, a function called `GetIOML` is defined and exported as a public method. This function returns the current values of `⎕IO` and `⎕ML`:

```
     ∇ r←GetIOML
[1]    :access public
[2]    :signature Int32[]←GetIOML
[3]    r←⎕IO ⎕ML
     ∇
```

5.2.4.1 aplclasses4

The C# source code (**<your_dir>/aplclasses4/net/project/Program.cs**) contains code to invoke the two different constructor functions `CTOR1` and `CTOR2`:

```
using System;
using APLClasses;
public class MainClass
       {
       public static void Main()
               {
               Primitives apl10 = new Primitives(1);
               int[] rslt10 = apl10.GetIOML();
               for (int i=0;i<rslt10.Length;i++)
                       Console.WriteLine(rslt10[i]);

               Primitives apl03 = new Primitives(0,3);
               int[] rslt03 = apl03.GetIOML();
               for (int i=0;i<rslt03.Length;i++)
                       Console.WriteLine(rslt03[i]);
               }
       }
```

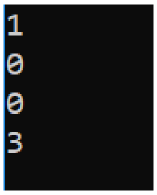
This code creates two instances of the `Primitives` class called `apl10` and `apl03`; the first is created with a constructor parameter of `(1)`, and the second with two constructor parameters `(0,3)`.

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.

To compile the C# source code

1. On the command line, navigate to **<your_dir>/aplclasses4/net**.
2. Run **build** (Linux and macOS)/**build.bat** (Microsoft Windows).
   This invokes the Dyalog script compiler to compile **aplclasses4.dws** to **aplclasses4.dll**, and then invokes the C# compiler to compile the C# source code (**Program.cs**) to produce an executable called **project.exe** in **<your_ dir>/aplclasses4/net/project/bin/Debug/net8.0**.

The output when the program is run is displayed in a console window (see *Figure 5-4*).



```
1
0
0
3
```

Figure 5-4: *Program output in console window*

5.2.5 Example 5

This example builds on the example in *Section 5.2.4*, 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 called `IndexGen1`, `IndexGen2` and `IndexGen3` respectively. The *index generator* function (monadic `⍳`) performs all of these operations, therefore the three APL functions are identical. However, their public interfaces, as defined in their `:Signature` statement, are all different. The overloading is achieved by entering the same name for the exported method (`IndexGen`) for each of the three APL functions.

Load the **aplclasses5.dws** workspace from **<your_dir>/aplclasses5**; the new version of the `Primitives` class includes three different versions of `IndexGen`. The first is the version we have seen before, which is defined to take a single argument of type `Int32` and to return a 1-dimensional array (vector) of type `Int32`:

```
     ∇ R←IndexGen1 N
[1]    :Access public
```

```
[2]    :Signature Int32[]←IndexGen Int32 N
[3]     R←⍳N
     ∇
```

The second version is defined to take two arguments of type `Int32` and to return a 2-dimensional array, each of whose elements is a 1-dimensional array (vector) of type `Int32`:

```
     ∇ R←IndexGen2 N
[1]    :Access public
[2]    :Signature Int32[][,]←IndexGen Int32 N1, Int32 N2
[3]     R←⍳N
     ∇
```

Although we could define seven more different versions of the method, taking 3, 4, 5 (and so on) 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:

```
     ∇ R←IndexGen3 N
[1]    :Access public
[2]    :Signature Array←IndexGen Int32[] N
[3]    R←⍳N
     ∇
```

All these functions use the same descriptive name, `IndexGen`.

A function can have several `:Signature` statements. As the three functions perform exactly the same operation, we could replace them with a single function:

```
     ∇ R←IndexGen1 N
[1]    :Access public
[2]    :Signature Int32[]←IndexGen Int32 N
[3]    :Signature Int32[][,]←IndexGen Int32 N1, Int32 N2
[4]    :Signature Array←IndexGen Int32[] N
[5]     R←⍳N
     ∇
```

5.2.5.1 aplclasses5

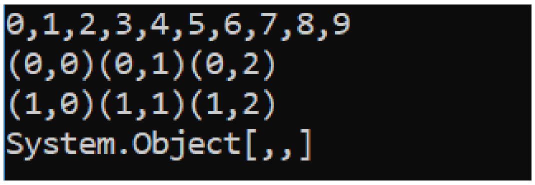
The C# source code (**<your_dir>/aplclasses5/net/project/Program.cs**) contains code to invoke the three different variants of `IndexGen` in the new **aplclasses.dll**. It uses a local sub-routine `PrintArray()`:

```
using System;
using APLClasses;
public class MainClass
      {
      static void PrintArray(int[] arr)
      {
            for (int i=0;i<arr.Length;i++)
                {
                Console.Write(arr[i]);
                if (i!=arr.Length-1)
                   Console.Write(",");
                }
      }

      public static void Main()
            {
            Primitives apl = new Primitives(0);
            int[] rslt = apl.IndexGen(10);
            PrintArray(rslt);
            Console.WriteLine("");
            int[,][] rslt2 = apl.IndexGen(2,3);


            for (int i=0;i<2;i++)
                     {
                     for (int j=0;j<3;j++)
                              {
                              int[] row = rslt2[i,j];
                              Console.Write("(");
                              PrintArray(row);
                              Console.Write(")");
                              }
            Console.WriteLine("");
                     }

            int[] args = new int[3];
            args[0]=2;
            args[1]=3;
            args[2]=4;
            Array rslt3 = apl.IndexGen(args);
            Console.WriteLine(rslt3);

      }
```

To compile the C# source code

1. On the command line, navigate to **<your_dir>/aplclasses5/net**.
2. Run **build** (Linux and macOS)/**build.bat** (Microsoft Windows).
   This invokes the Dyalog script compiler to compile **aplclasses5.dws** to **aplclasses5.dll**, and then invokes the C# compiler to compile the C# source code (**Program.cs**) to produce an executable called **project.exe** in **<your_ dir>/aplclasses5/net/project/bin/Debug/net8.0**.

The output when the program is run is displayed in a console window (see *Figure 5-5*).



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

Figure 5-5: *Program output in console window*

5.3 Interfaces

Interfaces define additional sets of functionality that classes can implement; however, interfaces contain no implementation except for static methods and static fields. An interface specifies a contract that a class implementing the interface must follow. Interfaces can contain shared (known as "static" in many compiled languages) or instance methods, shared fields, properties, and events. All interface members must be public. Interfaces cannot define constructors. The .NET runtime allows an interface to require that any class that implements it must also implement one or more other interfaces.

When you define a class, you list the interfaces which it supports following a colon after the class name. The value of ⎕USING (possibly set by :Using) is used to locate interface names.

If you specify that your class implements a certain interface, you must provide all of the members (methods, properties, and so on) defined for that interface. However, some interfaces are only marker interfaces and do not specify any members.

EXAMPLE

```
:Class Names: Object, IEnumerable,IEnumerator
```

This class is illustrated in the **aplclasses8.apln** APL Source file in **[DYALOG]/Samples/aplclasses/aplclasses8**.

Following the colon, the first name is the base class; in this case it is the most basic .NET class, `Object`. After the (optional) base class name is the list of interfaces that are implemented (omitted if there are no such interfaces). The `Names` class implements two interfaces, `IEnumerable` and `IEnumerator`.

`IEnumerable` and `IEnumerator` are required interfaces for an object that allows itself to be enumerated, that is, its contents can be iterated though one at a time. They define certain methods that get called at the appropriate time by the calling code when enumeration is required (for example, the `foreach` C# keyword or `:For/:In` in Dyalog APL. For more information, see https://learn.microsoft.com/en-us/dotnet/api/system.collections.ienumerable?view=net-8.0.

5.4 Creating .NET Classes with APL Source Files

New .NET classes can be defined and used within an APL Source file. This section provides a brief introduction to writing classes, aimed specifically at APL Source files – see the *Dyalog APL Programming Reference Guide* for more information on writing classes in Dyalog.

A class is defined by `:Class` and `:EndClass` statements:

- `:Class Name: Type` declares a new class called `Name`, which is based on the *base class* `Type`, which can be any valid .NET class.
- `:EndClass` terminates a class definition block.

The methods provided by the class are defined as function bodies enclosed within these statements. You can also define sub-classes or nested classes using nested `:Class` and `:EndClass` statements.

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

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

5.4.1 Example: Creating A .NET Class Using an APL Source File

The following code illustrates how you can create a .NET Class using an APL Source file. The example class is the same as in *Section 5.2.1*. The APL Source file **[DYALOG]/Samples/aplclasses/aplclasses6/aplclasses6.apln** is:

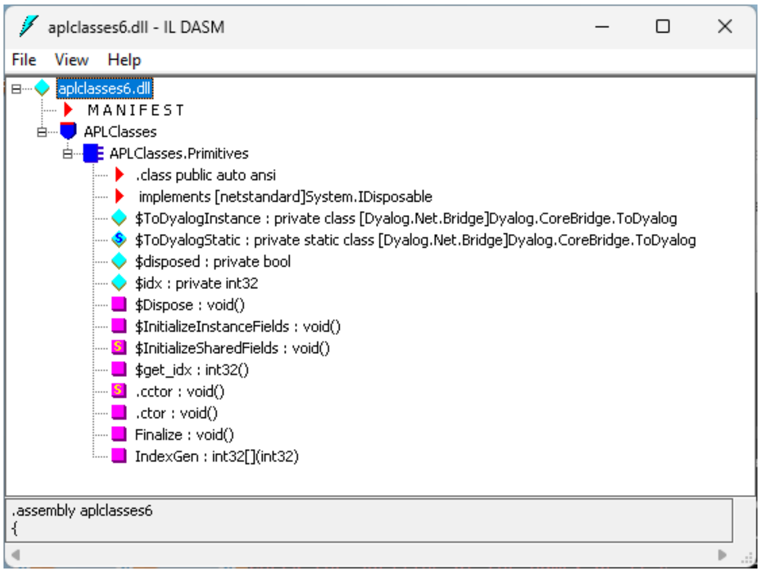
```
:Namespace APLClasses

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

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

:EndNamespace
```

This APL Source file code defines a namespace called `APLClasses`. This namespace acts as a container and is there to establish a .NET namespace of the same name within the resulting .NET assembly. Within `APLClasses` is 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.apln** is compiled to a .NET assembly using the `/t:library` flag.

```
aplclasses6> dyalogc.exe /t:library aplclasses6.apln
Dyalog APLScript compiler 64 bit. Unicode Mode. Version 19.0.48666.0
Copyright Dyalog Ltd 2000-2024
aplclasses6>
```

Figure 5-6 shows a view of the resulting **aplclasses6.dll** using ILDASM.


aplclasses6.dll - IL DASM

File View Help

```
aplclasses6.dll
    MANIFEST
    APLClasses
        APLClasses.Primitives
            .class public auto ansi
            implements [netstandard]System.IDisposable
            $ToDyalogInstance : private class [Dyalog.Net.Bridge]Dyalog.CoreBridge.ToDyalog
            $ToDyalogStatic : private static class [Dyalog.Net.Bridge]Dyalog.CoreBridge.ToDyalog
            $disposed : private bool
            $idx : private int32
            $Dispose : void()
            $InitializeInstanceFields : void()
            $InitializeSharedFields : void()
            $get_idx : int32()
            .cctor : void()
            .ctor : void()
            Finalize : void()
            IndexGen : int32[](int32)
```

```
.assembly aplclasses6
{
```

Figure 5-6: *ILDASM view of aplclasses6.dll structure*

As with other .NET classes, this .NET class can be called from APL. For example:

```
      )CLEAR
clear ws
      ⎕USING←'APLClasses,[DYALOG]/Samples/aplclasses/
                                 aplclasses6/net/aplclasses6.dll'
      APL←⎕NEW Primitives
      APL.IndexGen 10
1 2 3 4 5 6 7 8 9 10
```

5.4.2 Defining Properties

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

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

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

The `get` function is used to retrieve the value of the property and must be a niladic result returning function. The data type of its result determines the `Type` of the property. The `set` function is used to change the value of the property and must be a monadic function with no result. The argument to the function will have a data type `Type` specified by the `:Signature` statement. A property that contains a `get` function but no `set` function is effectively a read-only property.

EXAMPLE

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

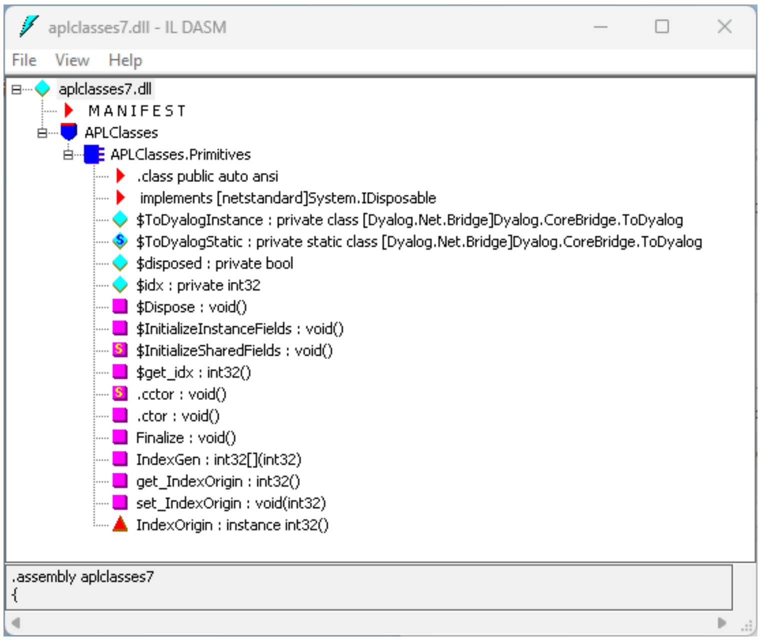
This declares a new property called `Name` whose data type is `System.Double`. When defining a property, the data type can be any valid .NET type that can be located through `⎕USING`.

The APL Source file **[DYALOG]/Samples/aplclasses/aplclasses7/aplclasses7.apln** shows how a property called `IndexOrigin` can be added to this example. Within the `:Property` block there are two functions called `get` and `set`; these functions use the previously-described fixed names and syntax, and are used to reference and assign a new value respectively:

```
:Namespace APLClasses

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

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

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

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

:EndProperty
:EndClass
:EndNamespace
```

The ILDASM view of the new **aplclasses7.dll**, showing the new IndexOrigin property, is shown in *Figure 5-7*.



```
aplclasses7.dll - IL DASM
File  View  Help
aplclasses7.dll
   MANIFEST
   APLClasses
      APLClasses.Primitives
         .class public auto ansi
         implements [netstandard]System.IDisposable
         $ToDyalogInstance : private class [Dyalog.Net.Bridge]Dyalog.CoreBridge.ToDyalog
         $ToDyalogStatic : private static class [Dyalog.Net.Bridge]Dyalog.CoreBridge.ToDyalog
         $disposed : private bool
         $idx : private int32
         $Dispose : void()
         $InitializeInstanceFields : void()
         $InitializeSharedFields : void()
         $get_idx : int32()
         .cctor : void()
         .ctor : void()
         Finalize : void()
         IndexGen : int32[](int32)
         get_IndexOrigin : int32()
         set_IndexOrigin : void(int32)
         IndexOrigin : instance int32()
.assembly aplclasses7
{
```

Figure 5-7: *ILDASM view of aplclasses7.dll structure*

As with other .NET classes, this .NET class can be called from APL. For example:

```
      )CLEAR
clear ws
      ⎕USING←'APLClasses,[DYALOG]/Samples/aplclasses/
                                  aplclasses7/net/aplclasses7.dll'
      APL←⎕NEW Primitives
      APL.IndexGen 10
1 2 3 4 5 6 7 8 9 10

      APL.IndexOrigin
1

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

5.4.3 Indexers

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

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

The `:Property Default` statement in Dyalog is closely modelled on the indexer feature in C# and employs similar syntax.

If you use `ILDASM` to browse a .NET class containing an indexer, you will see the indexer as the *default property* of that class, which is how it is implemented.

Index