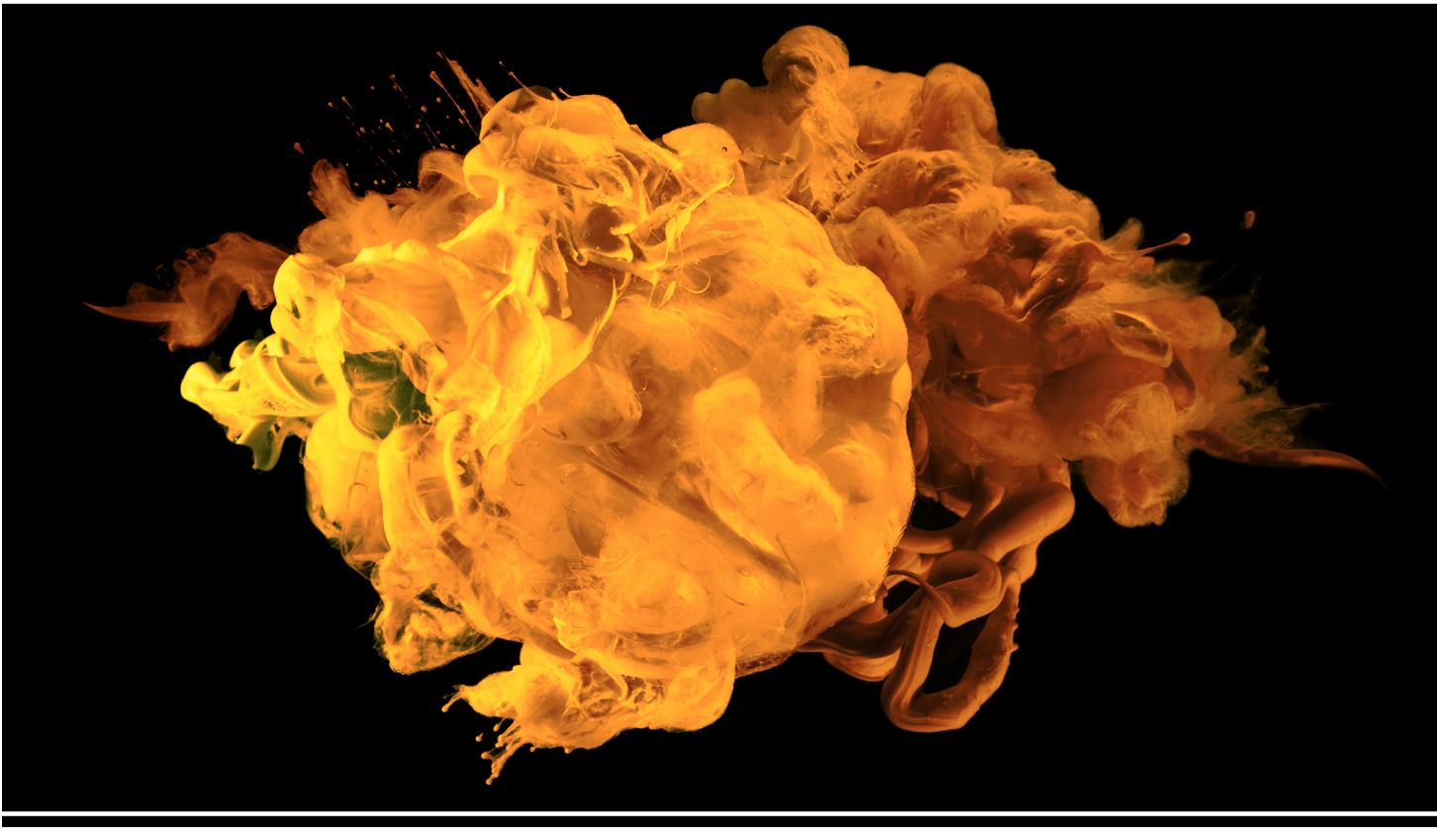
.NET Core Interface Guide

Dyalog version 18.0



DYALOG

The tool of thought for software solutions

.NET Core Interface Guide

Dyalog version 18.0
Document Revision: 20200526_180

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

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

Contents

1 About This Document

This document describes the Dyalog interface to Microsoft .NET Core, the cross-platform (Windows, Linux and macOS) successor to Microsoft's .NET Framework. It does not attempt to explain the features of .NET Core, except in terms of their APL interfaces. For information concerning .NET Core, see Microsoft's documentation, articles and helpfiles (available from https://docs.microsoft.com/en-us/dotnet/).

Microsoft does not support .NET on AIX.

1.1 Audience

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

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.

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information pertaining to behaviour in earlier releases of Dyalog or to functionality that still exists but has been superseded and is no longer recommended.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

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. A full list of the platforms on which Dyalog version 18.0 is supported is available at https://www.dyalog.com/dyalog/current-platforms.htm. Within this document, differences in behaviour between operating systems are identified with the following icons (representing macOS, Linux, UNIX and Microsoft Windows respectively):

2 Installation

The Dyalog .NET Core interface is still in the process of being developed. Currently it enables you to:

- create and use objects that are instances of .NET classes

Dyalog Ltd intends to enhance this interface to include additional functionality.

2.1 Pre-requisites

Microsoft does not support .NET Core on AIX or on the Raspberry Pi models Zero, 1 or 2. See Microsoft's .NET Core webpages (https://dotnet.microsoft.com/) for information on whether the version of Linux/Microsoft Windows that you are running supports .NET Core.

The Dyalog version 18.0 .NET Core interface requires Microsoft .NET Core version 3.1 – it does not work with other versions of .NET Core. For information on installing .NET Core, see *Section 2.1.1*.

Once Microsoft .NET Core has been successfully installed, no further installation is required to use the Dyalog .NET Core interface.

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

2.1.1 Installing .NET Core

Microsoft .NET Core can be downloaded from https://dotnet.microsoft.com/download (select the **Build Apps** option – **Download .NET Core SDK** and install it to the default location).

.NET Core should be installed according to Microsoft's instructions. Note that Microsoft only supply **.debs/.rpms** files for some releases of some Linux distributions; on others, such as Raspbian Buster, .NET Core needs to be installed manually.

If you decide not to install .NET Core in the default location, then you need to set the DOTNET_ROOT environment variable. See Microsoft's .NET Core documentation (https://docs.microsoft.com/en-gb/dotnet/) for instructions on how to do this. Note that this should not be set in Dyalog's configuration files; it is a Microsoft variable, not a Dyalog-specific one.

2.2 Files Installed with Dyalog

The components used to support the .NET Core 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 Core are processed.
- **Dyalog.Net.Bridge.Host.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 Core Interface

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

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

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

2.4 Verifying the Installation

If the interpreter cannot locate the .NET code, then an error message is generated when attempting the following:

```
      ⎕USING←'System'
      DateTime.Now
VALUE ERROR: Undefined name: DateTime
      DateTime.Now
```

In this situation, ensure that the .NET Core has been installed according to Microsoft's .NET Core documentation (https://docs.microsoft.com/en-gb/dotnet/) and the .NET Core interface has been enabled by setting DOTNET_ROOT (see *Section 2.3*).

EXAMPLE

This example shows the steps taken on a Linux/Pi system to downloaded the runtime to **/tmp/dotnet-runtime-3.1.3-linux-arm.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-3.1.3-linux-arm.tar.gz
sudo ln -s /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 Core documentation should always be followed.

3 .NET Classes

.NET Core 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 Core 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 Core 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 Core; 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** – this can be found in the .NET SDK and is distributed with Visual Studio.

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, the C# and Visual Basic languages allow the .NET namespace prefix to be elided. In this case, the programmer must declare a list of .NET namespaces with `Using` (C#) and `Imports` (Visual Basic) declaration statements. This list is then used to resolve unqualified class names referred to in the code. In either language, when the compiler encounters the unqualified name of a class, it searches the specified .NET namespaces for that class. In Dyalog, this mechanism is implemented by the `⎕USING` system variable. `⎕USING` performs the same two tasks that `Using/Imports` declarations and compiler directives provide in C# and Visual Basic; 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, which Dyalog assumes is located in the standard .NET Core directory that was specified when .NET Core was installed (for example, **C:/Program Files/dotnet/shared/Microsoft.NETCore.App/3.1.2/**).

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.assemblies** 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 an 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 Core, 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.

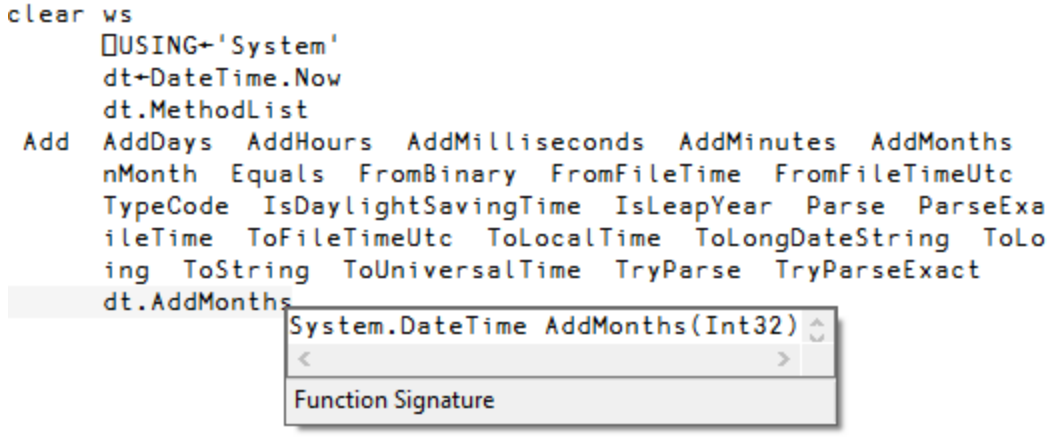


Figure 3-1: *Function signature – single integer argument*

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.

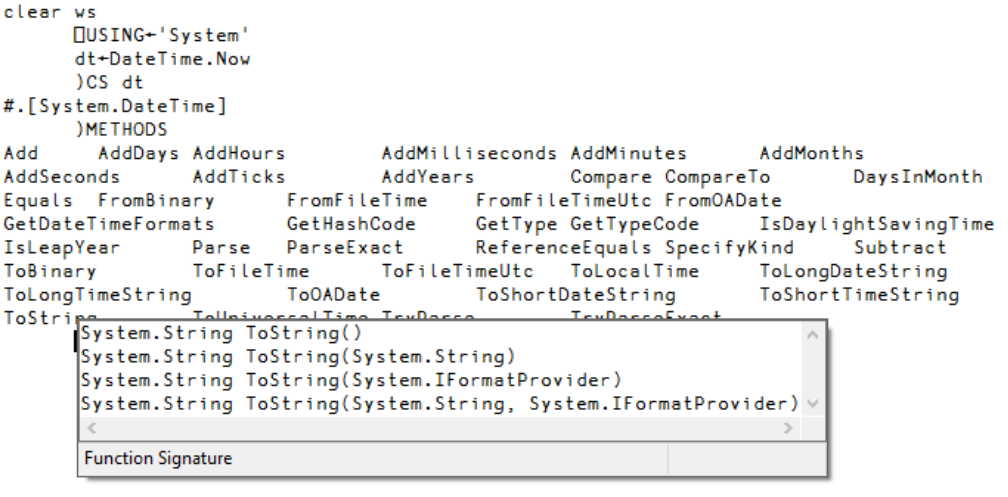


Figure 3-2: *Function signature – multiple arguments*

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 irespective 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

The .NET Framework provides a set of standard operators (methods) that are supported by certain classes. These operators include methods to compare two .NET objects and methods to add and subtract objects. In the case of the `DateTime` class, there are operators to compare two `DateTime` objects. For example:

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

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

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

```
      DT3←DateTime.Now
      DT3
18/03/2020 11:15:10

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

The 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 are shown in *Table 3-1*.

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

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

.NET Method	APL Primitive Function
`op_Equality`	= and ≡

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

.NET Method	APL Primitive Function
`op_Inequality`	≠ and ≠

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)
```

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 non-integer 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 18.0 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 18.0 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 18.0 Unicode\dyaedit.exe
C:\Program Files\Dyalog\Dyalog APL-64 18.0 Unicode\dyalog.exe
C:\Program Files\Dyalog\Dyalog APL-64 18.0 Unicode\dyalogc64_
unicode.exe  C:\Program Files\Dyalog\Dyalog APL-64 18.0
Unicode\dyalogrt.exe
```

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

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

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

```
      (d.GetFiles ⊂'*.exe').CreationTime
 05/03/2020 10:23:40  05/03/2020 10:23:28  14/11/2019 ...
```

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 18.0 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               14/11/2019 20:38:40
C:\...\...Unicode\bridge180-64_unicode.dll   05/03/2020 10:18:32
...
```

3.4.2 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.

EXAMPLE

This example will only work if your computer is configured to allow you to send email in this way and assumes that some additional files have been installed from https://www.nuget.org/packages/MailKit/.

```
∇ recip Send(subject msg);⎕USING;from;mail;to;builder;client
  ⎕USING←'' ',MimeKit.dll' ',MailKit.dll'
',BouncyCastle.Crypto.dll'
  from←⎕NEW MimeKit.MailboxAddress('john daintree (demo)'
'johnd@dyalog.com')
  to←⎕NEW MimeKit.MailboxAddress(''recip)
  mail←⎕NEW MimeKit.MimeMessage
  builder←⎕NEW MimeKit.BodyBuilder
  builder.TextBody←msg
  mail.Body←builder.ToMessageBody
  mail.Subject←subject
  mail.From.Add from
  mail.To.Add to
  client←⎕NEW MailKit.Net.Smtp.SmtpClient
  client.Connect'mail.dyalog.com' 587 0 ⎕NULL
  client.Send mail ⎕NULL ⎕NULL
∇
```

This could then be called as follows:

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

3.4.3 Web Scraping

.NET Core 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 Core'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
```

Note that 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 meaningless numbers in code, it is conventional to use names to represent particular values. These are known as *enumerated constants* or, more simply, as *enums*.

In .NET Core, 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 a 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` (96-bit integer) and `System.Int64` (64-bit integer) 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.

Index

V

W