

MODULE 11: MICROSOFT .NET FRAMEWORK INTEROPERABILITY

Module Overview

The Microsoft .NET Framework is an integral component of Microsoft Windows operating systems, that supports building and running the next generation of applications. It consists of Common Language Runtime which manages memory, execution, code safety, just-in-time compilation, additional system services, and a comprehensive class library that provides access to a wide range of system functionality. This includes operating system access, security, threading, text and XML manipulation, globalization, data access services, communication services, and more.

The .NET Framework is the main software development framework for Windows. Most Microsoft and third-party applications are developed in addition to .NET Framework. Microsoft Dynamics NAV 2013 is fully built in the .NET Framework and is a native .NET Framework application.

Microsoft Dynamics NAV 2013 and C/AL programming language include a set of features which allow for interoperability of C/AL code with the .NET Framework classes. You can take advantage of the .NET Framework interoperability so that Microsoft Dynamics NAV objects can interact with .NET Framework objects. In Microsoft Dynamics NAV objects, you can reference .NET Framework classes and call their members directly from C/AL code. The .NET Framework interoperability lets you use .NET Framework class library or third-party assemblies from the global assembly cache. You also can use your own custom assemblies that are deployed in the server or client executable directory. Use the .NET Framework interoperability to extend your solution beyond the boundaries of C/AL. Using .NET Framework interoperability features, your solution can perform any of the following functions:

- Communicate with third-party applications through web services.
- Integrate with Microsoft Office products.
- Manipulate text data with more flexibility than C/AL.
- Extend the functionality of the Microsoft Dynamics NAV 2013 client for Windows.



Note: *Even though client-side .NET Framework interoperability is not supported in Microsoft Dynamics NAV Portal Framework for Microsoft SharePoint 2010, server-side .NET Framework interoperability is fully supported without regard to the environment and display target.*

Objectives

The objectives are:

- Explain the .NET Interoperability features.
- Describe the concept of constructors.
- Communicate between client-side and server-side objects.
- Describe how to respond to events that are raised by .NET objects.
- Examine mapping between C/AL and .NET data types.
- Review the most important C/AL functions for managing .NET objects.
- Use arrays, collections, and enumerations.
- Explain how to stream data between C/AL and .NET objects.

The DotNet Data Type

The .NET Framework interoperability is achieved through the DotNet C/AL data type. By declaring a variable of DotNet data type, and subtyping it to a specific .NET Framework class, you can access all functionality of the referenced class. This is available to developers within the Visual Studio programming environment. The DotNet data type combines the power of the .NET Framework with the simplicity of coding in C/AL programming language. You can use the familiar syntax and development environment, while having full access to the .NET Framework.

The .NET Framework contains assemblies that can be either a part of the Global Assembly Cache (GAC) or can be custom assemblies that are deployed as a part of the application which is using them. In Microsoft Dynamics NAV 2013 you can use assemblies both from the GAC, and from the executable folder of Microsoft Dynamics NAV 2013 Service Tier or RoleTailored client.

The DotNet variable lets you do the following tasks:

- Access a specific .NET Framework class and its members.
- Respond to events that are raised by the referenced .NET Framework class.
- Target the Microsoft Dynamics NAV 2013 Server or the Microsoft Dynamics NAV 2013 client for Windows.

Declaring A DotNet Variable

To declare a DotNet variable, open the **C/AL Globals** or **C/AL Locals** window, enter the variable name, and select **DotNet** in the **Type** field. In the **Subtype** field, click the **AssistEdit(...)** button to open the **.NET Type List** window.

The **.NET Type List** window enables you to subtype a DotNet variable to a specific .NET Framework assembly and a specific class within a selected assembly. When declaring a DotNet variable, you first must select an assembly, and then a class within the selected assembly.

To select an assembly, look up the field **Assembly** to open the **Assembly List** window. This window shows the list of all available assemblies. Assemblies are grouped into the following two tabs.

Tab	Description
Dynamics NAV	Assemblies that are located in the local Add-ins folder of the Microsoft Dynamics NAV 2013 RoleTailored client installation are listed in the Dynamics NAV tab.
.NET	Assemblies that are installed in the GAC are listed in the .NET tab.



Note: The Dynamics NAV tab of the **.NET Type List** window shows only the assemblies that are present in the Add-ins folder of the RoleTailored client. If you want the declared variable to target the Microsoft Dynamics NAV 2013 Server, then you must make sure that the assembly that you referenced is also installed in the Add-ins folder of the Service tier.

After you select the assembly, click **OK** to return to the **.NET Type List** window, where you complete the declaration of the DotNet variable by selecting the required class from the **Type** list.



Note: NET Framework member names in C/AL code are case-sensitive. If you use the incorrect case when you call a member, then you receive an error when you compile the object. This behavior differs from other C/AL variables where you can mix cases and still compile the object. For other C/AL variables, the case is corrected automatically the next time that you open the object.

Deployment Options

To be accessed by Microsoft Dynamics NAV 2013, the .NET assemblies can be deployed either in the GAC or in the installation folder of either Microsoft Dynamics NAV 2013 Server or the RoleTailored client.

Global Assembly Cache (GAC)

GAC is a machine-wide, central repository of .NET assemblies. This repository is virtual, and assemblies can physically reside anywhere in the file system. Each assembly in the GAC is tagged with its strong name, version, and public key token. This makes it easy to reference a specific version of the assembly, and guarantees that compatibility of existing applications is never compromised after a new version of an assembly is installed.

After you install the .NET Framework, the GAC contains the Framework Class Library that Microsoft provides. However, you can deploy your own custom-built assemblies into the GAC and make them available to other applications. Many third-party products deploy their assemblies into the GAC.

An assembly that is deployed into the GAC can be consumed by any .NET Framework application. This includes being used as a DotNet variable in Microsoft Dynamics NAV 2013.

After an assembly is deployed into the GAC, it is available in the **.NET** tab of the **Assembly List** window.



Note: To learn more about the GAC and how to deploy assemblies into it, refer to the Developer and IT Pro Help documentation and to MSDN documentation online.

Local Application Folder

As an alternative to the GAC, you can deploy the assemblies in the Add-ins subfolder of the local application folder of Microsoft Dynamics NAV 2013 Server, the RoleTailored client, or both. This depends on whether DotNet variables that reference this assembly are targeting the server or the client.

The exact path of the Add-ins folder depends on the options that you chose during installation. However, the default path for the RoleTailored client is C:\Program Files (x86)\Microsoft Dynamics NAV\70\RoleTailored Client\Add-ins. The default path for the Server is C:\Program Files\Microsoft Dynamics NAV\70\Service\Add-ins.



Best Practice: For both the client and the server, put the assemblies into the subfolders of the Add-ins folder. This organizes multiple versions of the same assembly, and groups dependent assemblies.

Resolution Priority

When referencing classes that do not reside within the application, .NET applications first try to resolve the class by looking for it in the GAC. If the application you cannot find it in that location, then the application searches the local folder. The same is true of Microsoft Dynamics NAV 2013. Therefore, when C/AL code tries to create an instance of a .NET object, it first tries to load the assembly from the GAC. If it is not found there, it searches for the assembly in the Add-ins folder or its subfolders.



Best Practice: To guarantee maximum backward and forward compatibility of your custom C/AL code, it is best practice to deploy your assemblies into the GAC. This ensures that your C/AL code, which targets a specific version of a .NET assembly, does not stop working when a newer version of the same assembly is deployed.

Client-side and Server-side Execution

When declaring a DotNet variable, you can decide whether it targets the Microsoft Dynamics NAV 2013 client for Windows (client-side object) or Microsoft Dynamics NAV 2013 Server (server-side object). By default, all DotNet variables target the Microsoft Dynamics NAV 2013 Server and execute on the server-side.

To specify that a DotNet variable should target the Microsoft Dynamics NAV 2013 client for Windows, set the DotNet variable RunOnClient property to Yes.



Note: When a DotNet variable targets a specific tier, the instance of the referenced class is created in the specified tier. Any calls to its members execute under that tier. Make sure that the assembly that you reference is available in either the GAC or the local application folder on the physical server where it runs.

Events

An *event* is a message that is sent by an object to signal the occurrence of an action. The action might be caused by user interaction, such as a mouse click, or it might be triggered by some other program logic. The object that raises the event is called the event *publisher*. The object that captures the event and responds to it is called the event *subscriber*.

If a DotNet variable references a class that publishes events, you can subscribe to those events in C/AL code by setting the variable's WithEvents property to Yes. When you subscribe to events of a DotNet variable, a blank trigger is created in C/AL code for each event.

The DotNet variable is always subscribed to all events that the publisher exposes. You can define the event by adding code to the event trigger that is generated by C/SIDE. When the .NET object raises the event, the corresponding trigger in C/AL code is invoked and its code runs.



Note: Events are only supported on global variables.

Client-side and Server-side Events

Events always run on the tier that the DotNet variable targets. When a DotNet variable targets the client, the events execute on the client (client-side events). When the variable targets the Microsoft Dynamics NAV 2013 Server, the events execute on the server (server-side events).

Client-side events are only supported in page objects; server-side events are supported in all object types that support C/AL code.

Synchronous and Asynchronous Events

Events that are published by .NET Framework objects can be classified as *synchronous* or *asynchronous*.

Synchronous events are raised immediately when something occurs in the running application code in the main execution thread of the publisher. The execution of the code in the publisher waits until the event code in the subscriber is completed.

Module 11: Microsoft .NET Framework Interoperability

Asynchronous events are raised in a separate thread of the publisher. The execution on the main thread of the publisher immediately continues. This means that time may pass after something occurs, before the subscriber is notified of the event. Another effect is that the publisher may go out of scope by the time that the subscriber finishes the execution of the event. An example of an asynchronous event is the timer event of the System.Timers.Timer class.

When asynchronous events are raised in a DotNet variable, they are added to the event queue of the .NET Framework, and first processed when the connection between the client and the Microsoft Dynamics NAV Server is idle. A connection is idle when no Microsoft Dynamics NAV objects are communicating over the connection. If there is a long process that is running over the connection, you might experience a delay in the running of event triggers.

To determine whether an event of a class that you subscribe to is synchronous or asynchronous, view the documentation of the class. When you create your own .NET classes, all events are synchronous unless you specifically design an asynchronous event.



Best Practice: When subscribing to asynchronous events of a DotNet variable, declare such variables in a single-instance codeunit. Those codeunits remain instantiated until you close the client or a company. DotNet variables declared in them are still active after asynchronous events complete.

Constructors

A *constructor* is a method that creates an instance of an object and prepares it for use. This is usually done by initializing its members and properties and allocating necessary resources. Constructors can accept parameters, just as any other method.



Note: In the .NET Framework Interoperability in C/AL, a constructor always has the same name as the class that the DotNet variable references.

Classes usually have a constructor without parameters. This is known as the *default constructor*. A class may have multiple constructors. These are known as *overloaded constructors*. They differ from the default constructor because they accept different numbers or types of parameters.



Note: It is not important if a default constructor or an overloaded constructor is called. All constructors create an instance of a class. The only difference is in the initialization behavior. Constructors with more parameters usually set more default properties.

The “Constructor List for a DotNet Variable in the C/AL Symbol Menu” figure shows the list of constructors that are available for a DotNet variable referencing System.Collections.Generic.List<T> class:

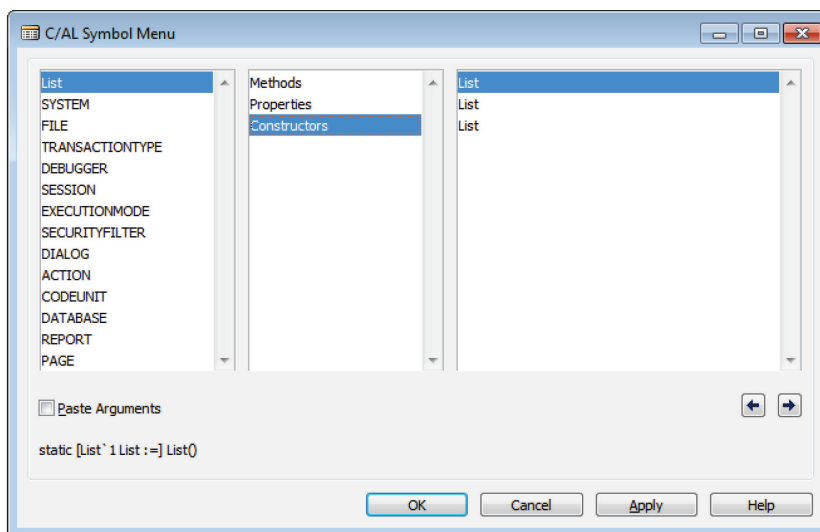


FIGURE 11.1: CONSTRUCTOR LIST FOR A DOTNET VARIABLE IN THE C/AL SYMBOL MENU

Static Classes and Members

In the .NET Framework, classes can be *static*. Static classes cannot be instantiated, and do not have a constructor. A DotNet variable that references a static class can be accessed immediately after it is declared. All other DotNet variables first must be instantiated by using a constructor.

The type information for a static class is loaded by the .NET Framework common language runtime (CLR) when the program that references the class is loaded. The program cannot specify exactly when the class is loaded. However, the class is guaranteed to be loaded and to have its fields initialized before the class is referenced for the first time in your program. A static class remains in memory for the lifetime of the application domain in which the program resides. All threads of the same application always have access to the same instance of the class.

Nonstatic classes can still have static members. To access static members the class does not have to be instantiated by using a constructor. You can access static members of a class whether you have called the constructor before or not.



Note: In Microsoft Dynamics NAV 2013, a static class or object is loaded once per Microsoft Dynamics NAV 2013 Server instance. The class or object is shared between all clients that are connected to the server instance. The data that is maintained by the static class or object is visible by all clients that use the type. You should consider this in your .NET Framework interoperability design to help avoid disclosing private information.

Demonstration: Declaring a DotNet Variable and Subscribing to Events

The following demonstration shows how to declare a DotNet variable and subscribe to its events. A DotNet variable is an instance of the System.IO.FileSystemWatcher class. It is configured to monitor a folder in the file system, and log any creation or deletion events for files.

Demonstration Steps

1. Create a table to keep track of the file system changes, and save it as table **90011, File System Log**.
 - a. Click **Tools > Object Designer**.
 - b. In **Object Designer**, click **Table**.
 - c. Click **New**.
 - d. In the **Table Designer**, define the following fields.

Field No.	Field Name	Data Type	Length
1	Event Time	DateTime	
2	Event Name	Text	250
3	File Name	Text	250

- e. Click **File > Save**.
 - f. In the **Save As** dialog window, in the **ID** field, enter "90011".
 - g. In the **Name** field, enter "File System Log".
 - h. Make sure that **Compiled** is selected, and then click **OK**.
 - i. Close the **Table Designer**.
2. Create a new list page for the **File System Log** table.
 - a. In **Object Designer**, click **Page**.
 - b. Click **New**.
 - c. In the **New Page** window, in the **Table** field, enter "File System Log".

- d. Select the **Create a page using a wizard** option, and select the **List** type.
 - e. Click **OK** to start the **List Page Wizard**.
 - f. Add all fields from the **Available Fields** list to the **Field Order** list.
 - g. Click **Finish** to close the **List Page Wizard**, and enter **Page Designer** for the new page.
 3. Add a new variable named *FileSystemLog* for the **File System Log** table. Add another variable named *FileWatcher*, of type DotNet and subtype System.IO.FileSystemWatcher from the System assembly.
 - a. Click **View > C/AL Globals**.
 - b. In the **Name** field, enter "FileSystemLog".
 - c. In the **Data Type** field, select Record, and in the **Subtype** field, enter "File System Log".
 - d. On the new row, in the **Name** field, enter "FileWatcher", in the **Data Type** field, select DotNet.
 - e. In the **Subtype** field, click **AssistEdit** to open the **.NET Type List** window.
 - f. Look up the **Assembly** field to open the **Assembly List** window.
 - g. In the **Assembly List** window, select the **.NET** tab, and then select System.
 - h. Click **OK** to accept the selection and close the **Assembly List** window.
 - i. In the **.NET Type List** window, select the System.IO.FileSystemWatcher class, and then click **OK**.
 4. Set the property on the *FileWatcher* variable to enable events.
 - a. Select the *FileWatcher* variable, and then click **View > Properties**.
 - b. In **FileWatcher – Properties** window, set WithEvents to **Yes**.
 - c. Close the FileWatcher – Properties window.
 - d. Close the **C/AL Globals** window.
 5. Create a function that accepts two parameters: one for the file path, and another for the event name. Then log the file system event in the **File System Log** table.
 - a. Click the **Functions** tab.
 - b. Create a new function, and name it **LogEvent**.
 - c. Select the **LogEvent** function and then click **Locals**.

d. Define the following parameters:

Var	Name	Data Type	Length
No	FileName	Text	250
No	EventName	Text	250

- e. Close the **C/AL Locals** window.
- f. Right-click the **LogEvent** function, and then click **Go To Definition** to open the **C/AL Editor** window at the **LogEvent** function trigger definition.
- g. In the **LogEvent** function trigger, enter the following code.

LogEvent Function Trigger

```
FileSystemLog.INIT;  
  
FileSystemLog."Event Time" := CREATEDATETIME(TODAY,TIME);  
  
FileSystemLog."Event Name" := EventName;  
  
FileSystemLog."File Name" := FileName;  
  
FileSystemLog.INSERT;
```

6. Add the code to the OnOpenPage trigger that constructs an instance of the FileSystemWatcher class, and sets it to monitor the C:\ folder.
- a. In the OnOpenPage trigger, enter the following code:

OnOpenPage Code

```
FileWatcher := FileWatcher.FileSystemWatcher;  
  
FileWatcher.Path := 'C:\';  
  
FileWatcher.NotifyFilter := 317;  
  
FileWatcher.EnableRaisingEvents := TRUE;
```



Note: The *NotifyFilter* property value of 317 corresponds to the C# value of *NotifyFilters.FileName | NotifyFilters.Attributes | NotifyFilters.LastAccess | NotifyFilters.LastWrite | NotifyFilters.Security | NotifyFilters.Size*.

7. Add the code to appropriate event triggers to log the file system event for creation and deletion events.
 - a. In the bodies of both the FileWatcher::Created and FileWatcher::Deleted event triggers, enter the following code:

FileWatcher::Created and FileWatcher::Deleted Event Trigger Code

```
LogEvent(e.FullPath,FORMAT(e.ChangeType));
```

8. Save the page as **90012, File System Watcher**, and run it.
 - a. Click **File > Save**.
 - b. In the **Save** window, in the **ID** field, enter "90012".
 - c. In the **Name** field enter "File System Watcher".
 - d. Click **OK**.
 - e. Close **Page Designer**.
 - f. In **Object Designer**, select page **90012, File System Watcher**.
 - g. Click **Run**.
9. Create and delete several files in the C:\ folder, and then refresh the page to view the changes.
 - a. On the **Start** menu, click **Computer**.
 - b. In **Windows Explorer**, double-click **Local Disk (C:)**.
 - c. Right-click in the empty area in the **Windows Explorer** window.
 - d. In the popup menu, click **New > Text Document**.
 - e. Repeat Step d.



Note: This creates two new text documents, named *New Text Document* and *New Text Document (2)*.

- f. Select the *New Text Document* and *New Text Document (2)* text files, and then delete them.
- g. In the **File System Watcher** page, on the **Actions** tab in the ribbon, click **Refresh**.

The File System Watcher shows two creation and two deletion events, as shown in the "File System Watcher Page" figure.

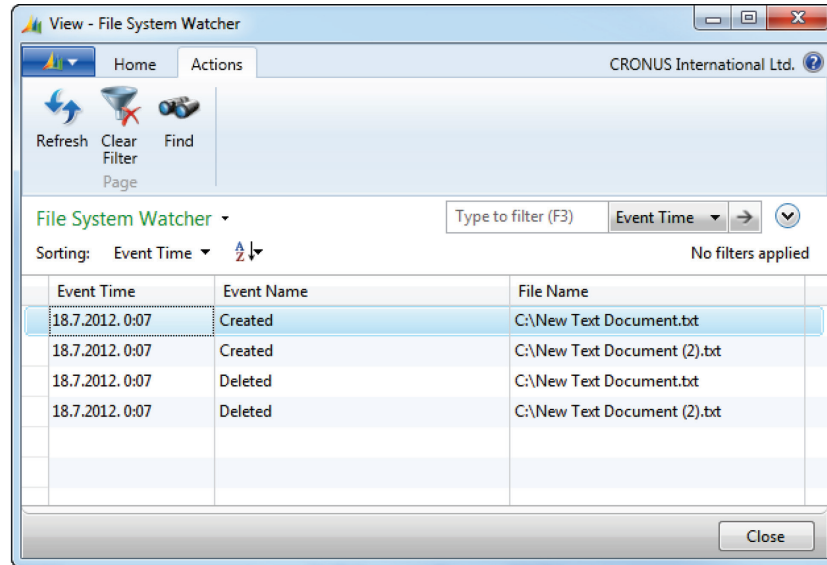


FIGURE 11.2: FILE SYSTEM WATCHER PAGE

Comparing Values

.NET Framework interoperability does not support direct use of operators to compare two DotNet variables.

The following code is not allowed.

Invalid comparison of DotNet variables

```
IF DotNet1 = DotNet2 THEN  
  
DoSomething;
```

If you want to compare two DotNet variables, use the Equals function on the DotNet.

The following example shows how to compare two DotNet variables.

Valid comparison of DotNet variables

```
IF DotNet1.Equals(DotNet2) THEN  
  
DoSomething;
```

You can perform comparisons by using .NET Framework methods and properties that return compatible C/AL types because these objects are implicitly converted to C/AL types before the comparison occurs.

The following example shows valid comparisons with DotNet data types and their members.

Comparison Examples

```
// DotNetList1 and DotNetList2 are of System.Collections.Generic.List<T> type  
  
DotNetList1 := DotNetList1.List;  
  
WHILE DotNetList1.Count < 5 DO  
  
    DotNetList1.Add(RANDOM(100));  
  
DotNetList2 := DotNetList2.List;  
  
WHILE DotNetList2.Count < DotNetList1.Count DO  
  
    DotNetList2.Add(RANDOM(100));
```

Data Type Mapping and Assignment

The .NET Framework data types are not the same as C/AL data types. When interoperating between C/AL and the .NET Framework, certain data mapping, conversion, and assignment rules are applied by both the C/AL compiler and Microsoft Dynamics NAV 2013 runtime.

In C/AL, some .NET Framework data types, such as Booleans, Integers, and Decimals are automatically converted to C/AL types. Because the types are converted, the .NET Framework versions of these types are not directly supported in C/AL.



For example, instead of using a .NET Framework Integer data type inside your C/AL code, you should use a C/AL Integer data type. When the C/AL Integer is passed as a parameter to a DotNet variable method, then the C/AL Integer is converted automatically to a .NET Framework Integer. When a DotNet variable method returns a .NET integer, it is converted automatically to C/AL Integer.

When assigning values between C/AL data types and the .NET Framework data types, there are data types which you can assign in either direction in any circumstances. On the other hand, there are C/AL and the .NET Framework data types which can only be assigned in one direction. Assigning these data types in another direction, may cause errors or data loss.

Bi-directional Mapping

A *right-to-left map* exists when data types between C/AL and the .NET Framework fully match and can be assigned between C/AL and the .NET Framework in both directions without any data loss or risk of a runtime error. Any data types that can be mapped right-to-left can be used easily in C/AL code for all types of operations. For example, a System.Bool always can be assigned between C/AL and the .NET Framework.




The following data types always can be assigned in both directions between C/AL and the .NET Framework without any errors or data loss.

.NET Framework Data Type	C/AL Data Type
System.Byte (0..255)	Char (0..255)
System.Int32 (-2,147,483,648..2,147,483,647)	Integer (-2,147,483,648..2,147,483,647)
System.Int64 (-9,223,372,036,854,775,808 ..9,223,372,036,854,775,807)	BigInteger (-9,223,372,036,854,775,808 ..9,223,372,036,854,775,807)
System.Bool (TRUE, FALSE)	Boolean (TRUE, FALSE)
System.Guid (128 bit number)	GUID (128 bit number)
System.Int32 (-2,147,483,648..2,147,483,647)	Option (-2,147,483,648..2,147,483,647)
	 Note: When assigning to and from the C/AL option variable, the value is mapped to System.Int32 data type.
System.IO.Stream	InStream OutStream
	 Note: Even though InStream and OutStream are freely assignable to and from System.IO.Stream, specific situations require specialized stream types, such as System.IO.MemoryStream.



Limited Mapping




Some C/AL data types and the .NET Framework data types do not fully match, and there are circumstances where assignment always works in one direction. However assignment in the opposite direction may result in a loss of data or even a run-time error. For example, you can always assign System.Enum to Integer, but assigning Integer to System.Enum may result in a run-time error. Especially around numeric types, .NET Framework provides more flexibility around how the data is stored. You must take additional precautionary steps when assigning data between such types where limitations in the mapping exist.

You can assign the following data types in both directions between C/AL and the .NET Framework. However, some data loss or errors may occur, depending on actual values.


.NET Framework Data Type	C/AL Data Type
System.Single (±3.402823e38) System.Double (±1.79769313486232e308) System.Decimal (±79,228,162,514,264,337,593,543,950,335)	Decimal (±999,999,999,999,999.99..999,999,999,999,999.99) <hr/>  Note: Assigning between System.Single, System.Double or System.Decimal to Decimal is supported in both directions. When assigning from C/AL to the .NET Framework, some precision may be lost. When assigning from .NET to C/AL, precision may be lost, or an error can occur if the .NET Framework variable value is out of bounds of the C/AL variable.
System.Char (0..65535) <hr/>  Note: Unicode character that is represented internally as a 16-bit unsigned integer.	Integer (±2,147,483,647) <hr/>  Note: Assigning from .NET Framework to C/AL is always possible. Assigning from C/AL to .NET Framework results in errors if the value that is passed is out of bounds of System.Char.

Module 11: Microsoft .NET Framework Interoperability

.NET Framework Data Type	C/AL Data Type
System.SByte (-128..127) System.Int16 (-32768..32767) System.UInt16 (0..65335) System.UInt32 (0.. 4,294,967,295) System.UInt64 (0..18,446,744,073,709,551,615)	Integer (-2,147,483,648..2,147,483,647) BigInteger (-9,223,372,036,854,775,808 ..9,223,372,036,854,775,807) <hr/>  Note: <i>The .NET Framework integer types of various data lengths (8, 16, 32, or 64 bytes) and various sign types (signed or unsigned) can be assigned to and from C/AL Integer and BigInteger types, as long as the value that is assigned from is not out of bounds of the variable that it is being assigned to. If the value is out of bounds, a run-time error occurs.</i> <hr/>
System.String	Text (up to 2 gigabytes) BigText (up to 2 gigabytes) Code (0 to 1024 bytes.) <hr/>  Note: <i>A Code variable always can be assigned to System.String. A System.String variable can be assigned only to Code if it contains less than 1024 characters.</i> <p><i>When assigning System.String to Code, it turns to uppercase. When it is assigned from Code to System.String, uppercase is retained.</i></p> <hr/>

.NET Framework Data Type	C/AL Data Type
<p>System.DateTime 1 January year 1 .. 31 December 9999</p>	<p>DateTime 3 January year 1 .. 31 December 9999</p> <hr/> <p> Note: Assigning dates between .NET Framework and C/AL result in possible data loss when dates earlier than January 3, year 1 are assigned to C/AL datetime. In addition, .NET Framework does not have a concept of 0D. This is represented as January 1, year 1 in the .NET Framework.</p> <hr/> <p>DateTime 3 January year 1 .. 31 December 9999</p> <p>Time (00:00:00.000 .. 23:59:59.999)</p> <hr/> <p> Note: Because the .NET Framework only has DateTime, C/AL Date, and Time, variable values can be passed on to .NET Framework only as DateTime. Use CREATEDATETIME(Date,OT) when passing the Date variable, or CREATEDATETIME(OD,Time) when passing the Time variable.</p>
<p>System.TimeSpan (resolution is 100 nanoseconds)</p>	<p>Duration (resolution is 1 millisecond)</p> <hr/> <p> Note: Duration is internally represented as a 64-bit unsigned integer. Because of resolution differences, there may be some data loss when assigning from .NET Framework to C/AL.</p>


Module 11: Microsoft .NET Framework Interoperability

.NET Framework Data Type	C/AL Data Type
System.Enum	Integer (±2,147,483,647)
	 Note: <i>System.Enum resembles Option because it stores a list of possible values. Although Option always is represented internally as an Integer in C/AL, in the .NET Framework, System.Enum can be of any integer data type. Additionally, System.Enum allows you to explicitly assign an integer value to a specific enum value. This makes it likely to result in a run-time error if you assign an arbitrary Integer value to System.Enum.</i>

Mapping of System.String and System.DateTime

Most of the .NET Framework types are automatically converted to C/AL data types when the DotNet variable is instantiated, such as when you use it as a parameter and a return value of the .NET Framework class methods. Therefore, the DotNet variables cannot be used if they reference those types explicitly. For example, if you declare a DotNet variable of type System.Int32, you cannot use that variable in code to assign values to or from it.

Exceptions to this rule are System.String and System.DateTime types that are not converted automatically to C/AL data types, and can be explicitly referenced by DotNet variables. For example, if you declare a DotNetVariable of type System.String, you can use it in code to assign values to or from it, and you can call all System.String methods or access its properties. However, if a .NET Framework method returns a value of type System.String or System.DateTime, they are implicitly converted to C/AL Text or DateTime types.

 **Note:** *Even though you can use System.String and System.DateTime to assign values to and from C/AL variables or constants, you still cannot use these variables in comparison operations. Everything that applies to comparing DotNet variables, applies to comparing DotNet variables with C/AL variables. You must still use specific .NET Framework comparison methods, or assign the values to C/AL variables first, and then compare the C/AL variables.*

The following code example shows valid and invalid data type assignments and comparisons.

Data Type mapping for System.String and System.DateTime

```
// DotNetString is DotNet(System.String)

// CALText is Text[1024]

DotNetString := 'This is a System.String';

DotNetString := DotNetString.ToUpper;

CALText := DotNetString;

// Invalid comparison

IF (CALText = DotNetString) THEN;

// Valid comparison

IF DotNetString.Equals(CALText) THEN;

// Valid comparison

// ToString returns a System.String, but C/AL

// compiler implicitly converts it to Text

IF CALText = DotNetString.ToString THEN;

// DotNetDateTime is DotNet(System.DateTime)

// CALDateTime is DateTime

DotNetDateTime := CREATEDATETIME(TODAY,0T);

DotNetDateTime := DotNetDateTime.AddDays(1);

CALDateTime := DotNetDateTime;

// Invalid comparison

IF CALDateTime = DotNetDateTime THEN;

// Valid comparison

IF DotNetDateTime.Equals(CALDateTime) THEN;

// Valid comparison

// AddDays returns a System.DateTime, but C/AL
```

```
// compiler implicitly converts it to DateTime
IF CALDateTime = DotNetDateTime.AddDays(0) THEN;
```

Assign Instance References

DotNet variables are always of reference type, and never of value type. Assigning a DotNet variable to another DotNet variable creates another reference to the same object in memory. Changing any properties or values in the first variable results in applying the same change to the second variable.

The following code example shows the assignment of one DotNet variable to another.

Assigning DotNet variables

```
// List1 and List2 are System.Collections.Generic.List<T>
// From the mscorlib assembly
List1 := List1.List;
List2 := List1;
List1.Add('First');
List2.Add('Second');
List2 := List2.List;
List2.Add('Third');
MESSAGE('List1.Count = %1',List1.Count);
MESSAGE('List2.Count = %1',List2.Count);
```

In the code that was mentioned earlier, the two DotNet variables are of the type `System.Collections.Generic.List<T>`. The first list is instantiated, and assigned to the second variable. Both variables reference the same list object in memory. An element then is added to the first list, and an element is added to the second list. However, because both variables now point to the same list, there are two elements in the list now. Regardless of which variable (*List1* or *List2*) you use to access the object, you always access the same instance of `System.Collections.Generic.List<T>`. The *List2* variable then is instantiated, at which stage it loses the reference to the first list, and receives a new reference to a new instance. When you add an element to that list, nothing is added to *List1*, because *List1* and *List2* now point to two separate objects. You can easily verify that by showing the Count property of both variables.

The following “References and Instances Diagram” shows how objects are instantiated, and how variables act as references only to other objects.

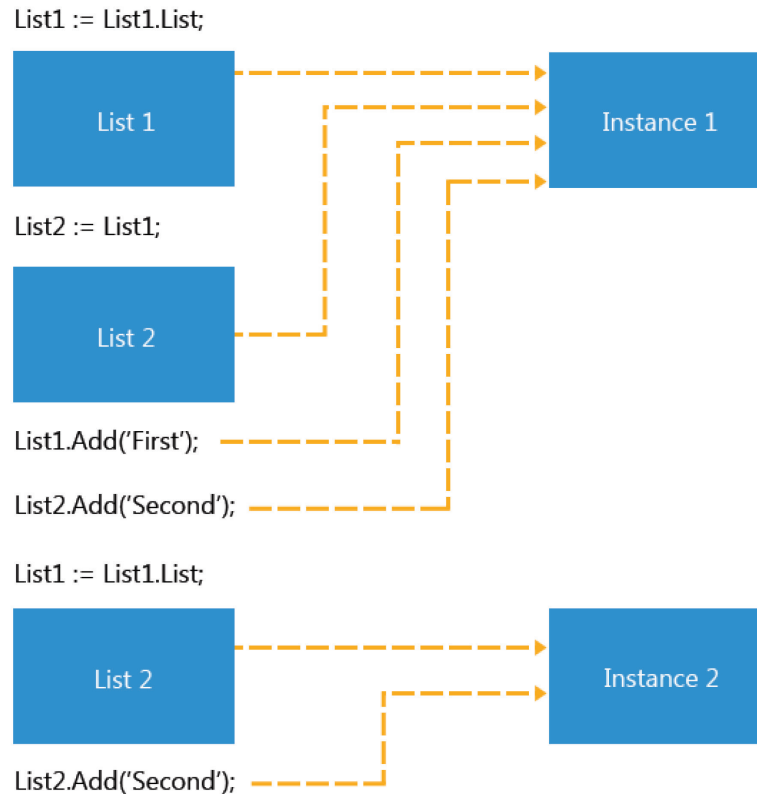


FIGURE 11.3: REFERENCES AND INSTANCES DIAGRAM

Assignment Between DotNet Subtypes

The .NET Framework is a strongly typed framework where each variable must be of a specific type. The type must be known at the time of declaration of the variable.

You can assign one DotNet variable to another even if they are of different .NET Framework types. This can happen only if the .NET Framework type of the variable being assigned from is compatible with the .NET Framework type of the variable being assigned to. Typically, the type compatibility in the .NET Framework is achieved through an object programming concept called *inheritance*. In the .NET Framework, types can inherit other types. There are no limits to the levels of inheritance. Any descendant type always inherits all its ancestor types.

For the purposes of assignment, ancestors are always compatible with their descendants, but descendants are not always fully compatible with their ancestors. Types are incompatible across inheritance trees. Variables can only be assigned if both types are on the same inheritance branch.

When assigning between types, you can always assign a descendant-type variable to ancestor-type variable. However, assigning an ancestor-type variable to a descendant-type variable is not considered safe. If the code accesses descendant type members on a reference of ancestor type, the error is detected only during runtime.

The following example shows how ancestor and descendant types can be mutually assigned, and how assigning ancestors to descendants is not safe.

Assignment between types

```
// All types are from System.Xml.Linq namespace
XDocument := XDocument.Parse('<xml attribute="first" />');
XNode := XDocument.Root.FirstAttribute;
MESSAGE('%1',XNode.ToString);
XElement := XNode;
MESSAGE('%1',XElement.Document.ToString);
MESSAGE('%1',XElement.HasElements);
```

In the code that was mentioned earlier, `XNode` is the ancestor of `XAttribute`. Therefore, assignment to `XNode` succeeds. However, the later assignment of the same `XNode` to a variable of type `XElement` is not safe, because the actual type is `XAttribute`, which is not fully compatible with `XElement`. The first subsequent `MESSAGE` call succeeds, because `Document` property is a member of `XNode` ancestor. Therefore, it is visible in both `XElement` and `XAttribute`. However, the second `MESSAGE` call fails, because `HasElements` property is a member of `XElement` (the declared type), but is not a member of `XAttribute` (the actual type).

System.Object

All types in the .NET Framework inherit from `System.Object`. `System.Object` is a common ancestor of all .NET Framework types. Therefore, assignment of any .NET Framework type to a variable of type `System.Object` is considered safe.

As in any other .NET Framework programming language, in the .NET Framework interoperability, you can assign any DotNet variable to a DotNet variable of type

System.Object. In the assignment operations, you cannot assign any native C/AL data types to a DotNet variable of type System.Object. However, you can pass the variables of simple C/AL data and several complex data types as parameters to any .NET Framework object methods to replace a System.Object parameter.

Local and Global Scope

A local DotNet variable is disposed of by default when it goes out of scope. This happens after the execution of the function where it is declared finishes. The disposal does not occur immediately, but is taken care of by the *Garbage Collector* component of the .NET Framework. The Garbage Collector cleans up any unused resources that are left over by objects that either do not have any active variables referencing them, or that are explicitly disposed of through the IDisposable interface.

All DotNet variables implement the IDisposable interface. Microsoft Dynamics NAV 2013 runtime call their Dispose method as soon as they go out of scope. If you assign a local DotNet variable to a global one, when the local variable is disposed, the global one references a potentially non-existing object.

If you do not want your local DotNet variables to be disposed when they go out of scope, set the SuppressDispose property to Yes on each local DotNet variable that you do not want disposed. This lets you use the local variable outside the scope where it was instantiated.



Note: You typically set the SuppressDispose on a variable that refers to a type of system resource that has a special disposal pattern.

Serializing Data Between Client And Server

When you write C/AL code, you may assign a variable with RunOnClient set to Yes (a client-side object), to a variable with RunOnClient set to No (a server-side object). The opposite is also true. This is possible, but only if the object being referenced supports serialization.

In .NET Framework, serialization converts an object into a format that you can use to store the current state of the object, so that you can restore the object later together with the identical state. You typically use serialization when you transmit an object over a network connection, or when you store it in a database.

Microsoft Dynamics NAV 2013 .NET Framework Interoperability uses serialization for communication between client-side .NET Framework objects and server-side .NET Framework objects. The serialization occurs under the following conditions:

- When a server-side object variable is assigned to a variable that is assigned as client-side. The opposite is also true.
- When a server-side object variable is passed as a parameter in a method call from the server to a client-side object. The opposite is also true.

Serialization requires that the .NET Framework types that are referenced by the DotNet variables are serializable. Many types in the Microsoft .NET Framework class library are already serializable. If you are using a .NET Framework type that cannot be serialized, then you must change the type to make it serializable. You can do so by applying the `System.SerializableAttribute` of type attribute, or by implementing the `System.Runtime.Serialization.ISerializable` interface.



Note: *If you want to learn more about serialization in the .NET Framework objects, and how to make your custom .NET Framework objects serializable, refer to the Developer and IT Pro Help, or to the MSDN documentation online.*

.NET Framework interoperability C/AL Functions

Several .NET Framework concepts are completely foreign to C/AL. For example, there is no concept of *null* in C/AL. In the .NET Framework *null* is an important concept that is frequently used and depended upon.

In order to provide access to such concepts, C/AL includes several system functions that help interoperability with the .NET Framework. These system functions make it simpler to write code in C/AL, which basically targets the .NET Framework exclusively.

null and ISNULL function

In the .NET Framework, most types are reference types. This means that the variable does not hold the actual value, but holds the reference to an object instead. There are very few value types in the .NET Framework. They include Boolean, numeric, and DateTime types. These are all built-in types. Examples of reference types are classes, arrays, and strings. A programmer cannot declare a new value type.

In C# (a .NET Framework programming language), *null* is a keyword that represents a null reference. This means a reference that does not refer to any object. When a reference-type variable is declared, *null* is its default value.



Note: *If you are more familiar with Visual Basic .NET, then you should be aware that Visual Basic uses the Nothing keyword instead of the null keyword.*

If you try to access a DotNet variable that holds the null value, which is typically before it was instantiated, a run-time error is thrown. To avoid that situation, check whether the variable holds no reference by calling the ISNULL function. This function takes a DotNet variable as a parameter, and returns TRUE if the variable has a null value, or FALSE if it references a valid object.

The following code example shows how to use the ISNULL function.

ISNULL function

```
AddToList(List : DotNet "System.Collections.Generic.List`1";Value : Text[1024])  
  
IF ISNULL(List) THEN  
  
    List := List.List;  
  
List.Add(Value);
```

System.Type .NET Framework Type

Reflection is the capability of classes to gain insight into their own structure or the structure of other classes. Reflection is an important concept in the .NET Framework. At the core of that concept is the data type, which is represented by the System.Type class. System.Type class represents type declarations. These include the following types:

- Class types
- Interface types
- Array types
- Value types
- Enumeration types
- Type parameters
- Generic type definitions

In addition to reflection, type information is also very important because .NET Framework is a strongly typed framework. Each variable must be of a specific type, and the type must be known at the time of declaration of the variable.

GETDOTNETTYPE

The GETDOTNETTYPE C/AL function returns the .NET Framework type of a specified variable. It takes a variable or a constant of any type as a parameter, and returns the System.Type value that represents the .NET Framework type of the expression that is passed as a parameter to the GETDOTNETTYPE function.

The following code examples show possible uses and resulting values of GETDOTNETTYPE

GETDOTNETTYPE examples

```
// Returns System.Boolean
MESSAGE('%1',GETDOTNETTYPE(1 = 2));

// Returns System.String
MESSAGE('%1',GETDOTNETTYPE('some text'));

// Returns System.Int32
MESSAGE('%1',GETDOTNETTYPE(Cust.Blocked));

// Returns System.Xml.Linq.XDocument
MESSAGE('%1',GETDOTNETTYPE(XDocument));

// InStr is InStream

// Returns System.IO.Stream
MESSAGE('%1',GETDOTNETTYPE(InStr));
```

GETDOTNETTYPE function can be used in C/AL in the situations where *typeof* keyword would be used in C#.

In the following example, the GETDOTNETTYPE is used to create an instance of an array that holds five elements of C/AL type Decimal.

GETDOTNETTYPE Example

```
// Dec is Decimal

// Array is System.Array

Array := Array.CreateInstance(GETDOTNETTYPE(Dec),5);
```

CANLOADTYPE

The CANLOADTYPE function checks whether a .NET type can be loaded by Microsoft Dynamics NAV 2013. You can use this function to check whether an assembly is installed on the target computer, or if you have sufficient security permissions to use the assembly before accessing the assembly through a DotNet variable in the C/AL code.

The following example shows how to use the CANLOADTYPE function:

CANLOADTYPE Example

```
IF CANLOADTYPE(FileWatcher) THEN BEGIN

    FileWatcher := FileWatcher.FileSystemWatcher;

    FileWatcher.Path := 'C:\';

    FileWatcher.NotifyFilter := 317;

    FileWatcher.EnableRaisingEvents := TRUE;

END ELSE

    MESSAGE('The file system watcher functionality is not available.');
```

Lab 11.1: Use a Dictionary Object

Scenario

Isaac, a developer at Cronus International Ltd., is developing a report that shows the profit and loss statement for a specific customer. He has to calculate the sum of all transactions for each General Ledger (G/L) account where **Income/Balance** is **Income Statement**, and where the G/L entries are related to transactions of the specified customer. He decides to use the `System.Collections.Generic.Dictionary` class because of its fast performance for random data access to develop the algorithm that is used by the required report.

Exercise 1: Declare and Instantiate a Dictionary

Exercise Scenario

Isaac creates a new codeunit, and declares and instantiates a `System.Collections.Generic.Dictionary` variable. He then writes a function that accepts a parameter of the same type and a customer number. Finally, he calls that function from the `OnRun` trigger for testing. The dictionary variable uses `G/L Account Nos.` as key, and the total amount per `G/L Account` as value.

Task 1: Creating a new codeunit

High Level Steps

1. Create a new codeunit and save it as **90014, G/L Dictionary By Customer**.
2. Create a new function named **PopulateDictionary** which accepts two parameters. Name one parameter *Dict* of type `DotNet` and subtype `System.Collections.Generic.Dictionary`2`. Name the second parameter *CustNo* of type `Code[20]`.

Detailed Steps

1. Create a new codeunit and save it as **90014, G/L Dictionary By Customer**.
 - a. Click **Tools > Object Designer**.
 - b. In the **Object Designer** window, click **Codeunit**, and then click **New**.
 - c. In the **C/AL Editor** window, click **File > Save**.
 - d. In the **ID** field, enter "90014".
 - e. In the **Name** field, enter "G/L Dictionary By Customer".
 - f. Click **OK**.
2. Create a new function named **PopulateDictionary** which accepts two parameters. Name one parameter *Dict* of type `DotNet` and subtype `System.Collections.Generic.Dictionary`2`. Name the second parameter *CustNo* of type `Code[20]`.

- a. Click **View > C/AL Globals**.
- b. In the **Functions** tab, enter "PopulateDictionary", and then click **Locals**.
- c. In the **Name** field, enter "Dict", and then select DotNet as **DataType**.
- d. In the **Subtype** field, click the **AssistEdit** button.
- e. In the **.NET Type List** window, look up the **Assembly** field.
- f. In the **Assembly List** window, in the **.NET** tab, select mscorlib, and then click **OK**.
- g. In the **.NET Type List** window, select System.Collections.Generic.Dictionary`2, and then click **OK**.
- h. In the **C/AL Locals** window, create a new parameter.
- i. In the **Name** field, enter "CustNo", and then select Code as **DataType**.
- j. In the **Length** field, enter "20".
- k. Close the **C/AL Locals** window, and then close the **C/AL Globals** window.
- l. Click **File > Save**.

Task 2: Write code to Test the PopulateDictionary function

High Level Steps

1. Declare a global variable of type System.Collections.Generic.Dictionary`2, and then name it *DictTest*.
2. Declare a global variable of type Record, and subtype G/L Account, and name it *GLAcc*.
3. Write code in the OnRun trigger to call the PopulateDictionary function that uses **DictTest** and constant "10000" as parameters. Make sure that you create an instance of the DictTest variable before passing it as a parameter.
4. Add code to the OnRun trigger to iterate through all G/L Accounts. It shows a message if the DictTest contains a key equal to the G/L Account No. Show the value of the cached key in the *DictTest* dictionary.

Detailed Steps

1. Declare a global variable of type System.Collections.Generic.Dictionary`2, and then name it *DictTest*.
 - a. Click **View > C/AL Globals**.
 - b. In the **Variables** tab, in the **Name** field, enter "DictTest", then select DotNet as **DataType**.
 - c. In the **Subtype** field, click **AssistEdit**.

- d. In the **.NET Type List** window, look up the **Assembly** field.
 - e. In the **Assembly List** window, in the .NET tab, select mscorlib, and then click **OK**.
 - f. In the **.NET Type List** window, select System.Collections.Generic.Dictionary`2, and then click **OK**.
2. Declare a global variable of type Record, and subtype G/L Account, and name it *GLAcc*.
 - a. In the **C/AL Globals** window, create a new variable.
 - b. In the **Name** field, enter "GLAcc".
 - c. Select Record as **DataType**, and then in the **Subtype** field, type "15".
 - d. Close the **C/AL Globals** window.
 3. Write code in the OnRun trigger to call the PopulateDictionary function that uses **DictTest** and constant "10000" as parameters. Make sure that you create an instance of the DictTest variable before passing it as a parameter.
 - a. In the OnRun trigger, write the following code:

OnRun Trigger

```
DictTest := DictTest.Dictionary;  
PopulateDictionary(DictTest,'10000');
```

4. Add code to the OnRun trigger to iterate through all G/L Accounts. It shows a message if the DictTest contains a key equal to the G/L Account No. Show the value of the cached key in the *DictTest* dictionary.
 - a. Append the following code to the OnRun trigger:

Additional OnRun Trigger Code

```
GLAcc.RESET;  
  
IF GLAcc.FINDSET THEN  
  
    REPEAT  
  
        IF DictTest.ContainsKey(GLAcc."No.") THEN  
  
            MESSAGE('Total for %1 is %2',  
  
                GLAcc."No.",
```

```
DictTest.Item(GLAcc."No.");  
UNTIL GLAcc.NEXT = 0;
```

Results

A new codeunit object, **90014, G/L Dictionary By Customer**, that has a function that populates G/L accounts that are involved in transactions for a specific customer.

Exercise 2: Populate the dictionary

Exercise Scenario

Isaac writes the code in the **PopulateDictionary** function that iterates through all G/L Entries. This function analyzes whether there are **Customer Ledger Entries** that belong to the specified customer. If there are, then the total amount of the **G/L Entry** is stored in the dictionary.

Task 1: Iterate through all G/L Accounts

High Level Steps

1. In **PopulateDictionary** function, make sure that the *Dict* parameter is not null. If it is null, instantiate a new `System.Collections.Generic.Dictionary`. Otherwise, make sure that *Dict* is empty.
2. Append code to the **PopulateDictionary** function to make sure that there are no filters on the global variable *GLAcc*. Then write code which iterates through all *GLAcc* that have Income/Balance equal to Income Statement.

Detailed Steps

1. In **PopulateDictionary** function, make sure that the *Dict* parameter is not null. If it is null, instantiate a new `System.Collections.Generic.Dictionary`. Otherwise, make sure that *Dict* is empty.
 - a. Write the following code in the **PopulateDictionary** function.

Instantiate or clear the dictionary

```
IF ISNULL(Dict) THEN  
  
    Dict := Dict.Dictionary  
  
ELSE  
  
    Dict.Clear;
```


2. Append code to the **PopulateDictionary** function to make sure that there are no filters on the global variable *GLAcc*. Then write code which iterates through all *GLAcc* that have Income/Balance equal to Income Statement.
 - a. Append the following code to the **PopulateDictionary** function.

Iterate through income statement G/L accounts

```
GLAcc.RESET;

GLAcc.SETRANGE("Income/Balance",
  GLAcc."Income/Balance"::"Income Statement");

IF GLAcc.FINDSET THEN

  REPEAT

  UNTIL GLAcc.NEXT = 0;
```

Task 2: Iterate through all G/L Entries for current G/L Account

High Level Steps

1. Declare a global variable of type Record, and subtype G/L Entry. Name it *GLEntry*.
2. Within the REPEAT..UNTIL block for the *GLAcc* variable, iterate through all G/L Entries for current G/L Account.

Detailed Steps

1. Declare a global variable of type Record, and subtype G/L Entry. Name it *GLEntry*.
 - a. In the **C/AL Globals** window, create a new variable. In the **Name** field, enter "GLEntry", and then select Record as the **Data Type**.
 - b. Then in **Subtype** field, type "17".
2. Within the REPEAT..UNTIL block for the *GLAcc* variable, iterate through all G/L Entries for current G/L Account.
 - a. Within the REPEAT..UNTIL block for the *GLAcc* variable, write the following code.

Iterate through G/L Entries for a G/L Account

```
GLEntry.RESET;  
  
GLEntry.SETCURRENTKEY("G/L Account No.");  
  
GLEntry.SETRANGE("G/L Account No.",GLAcc."No.");  
  
IF GLEntry.FINDSET THEN  
  
    REPEAT  
  
        UNTIL GLEntry.NEXT = 0;
```

Task 3: Check whether there are Customer Ledger Entries for a specified customer

High Level Steps

1. Declare a global variable of type Record, and subtype Cust. Ledger Entry. Name it *CustLedgEntry*.
2. Within the REPEAT..UNTIL block of the *GLEntry* variable, check whether there are customer ledger entries with the same Transaction No. that belong to the customer that is specified by the *CustNo* parameter.

Detailed Steps

1. Declare a global variable of type Record, and subtype Cust. Ledger Entry. Name it *CustLedgEntry*.
 - a. In the **C/AL Globals** window, create a new variable. In the **Name** field, type "CustLedgEntry".
 - b. Select Record as **Data Type**, and then in **Subtype** field enter "21".
2. Within the REPEAT..UNTIL block of the *GLEntry* variable, check whether there are customer ledger entries with the same Transaction No. that belong to the customer that is specified by the *CustNo* parameter.
 - a. Within the REPEAT..UNTIL block of the *GLEntry* variable, write the following code.

Check matching customer transactions

```
CustLedgEntry.RESET;  
  
CustLedgEntry.SETRANGE("Customer No.",CustNo);  
  
CustLedgEntry.SETRANGE("Transaction No.",GLEntry."Transaction No.");  
  
IF NOT CustLedgEntry.ISEMPTY THEN BEGIN  
  
END;
```

Task 4: Store the total in the dictionary

High Level Steps

1. Declare a global variable of type Decimal. Name it *AccTotal*.
2. Check whether the current G/L Account No. is already in the dictionary. If so, retrieve the value into the *AccTotal* variable. If no, set *AccTotal* to zero.
3. If it exists, remove the key from the dictionary, and then add the actual G/L Entry to the *AccTotal* amount. Store the *AccTotal* amount in the dictionary with the key of G/L Account No.

Detailed Steps

1. Declare a global variable of type Decimal. Name it *AccTotal*.
 - a. In the **C/AL Globals** window, create a new variable.
 - b. In the **Name** field, enter "AccTotal".
 - c. Select Decimal as **DataType**.
2. Check whether the current G/L Account No. is already in the dictionary. If so, retrieve the value into the *AccTotal* variable. If no, set *AccTotal* to zero.
 - a. Within the BEGIN..END block of the Customer Ledger Entry check, and write the following code.

Check if key exists in the dictionary

```
IF Dict.ContainsKey(GLAcc."No.") THEN  
  
    AccTotal := Dict.Item(GLAcc."No.")  
  
ELSE  
  
    AccTotal := 0;
```

3. If it exists, remove the key from the dictionary, and then add the actual G/L Entry to the *AccTotal* amount. Store the *AccTotal* amount in the dictionary with the key of G/L Account No.
 - a. Append the following code after the code that was added in the previous task and within the same BEGIN..END block.

Store the value into the dictionary.

```
AccTotal := AccTotal + GLEntry.Amount;  
  
IF Dict.ContainsKey(GLAcc."No.") THEN  
  
    Dict.Remove(GLAcc."No.");  
  
Dict.Add(GLAcc."No.",AccTotal);
```



Note: In C# or Visual Basic .NET, you do not remove the key/value pair from the dictionary first. Because of the syntactic specifics of C/AL, this is the simplest way to replace an existing key in the dictionary with a new value.

Task 5: Testing the function

High Level Steps

1. Run the codeunit to test it.

Detailed Steps

1. Run the codeunit to test it.
 - a. Click **File** > **Save**.
 - b. Click **File** > **Run**.

Results

Functional PopulateDictionary function

Streaming

Streaming is an important concept in the .NET Framework. There are several .NET Framework types that facilitate management of different types of data streams. In Microsoft Dynamics NAV 2013, there are two data types which handle streams: the *InStream* and the *OutStream*. These data types directly map to the *System.IO.Stream* .NET Framework type. They can be assigned to and from the DotNet variables of the *System.IO.Stream* type or its descendant types.


In Microsoft Dynamics NAV 2013 the data types that handle streams do not differentiate between the types of streams. These data types provide an abstract programming interface that relieves developers from understanding the intrinsic details of a specific stream implementation. In the .NET Framework there are different types of streams that address specific scenarios that involve data streaming.

Types of Streams in .NET

The base type which handles streams in the .NET Framework is *System.IO.Stream*. It is an abstract type, which means that objects of this kind cannot be directly instantiated. However, you can use variables of type *System.IO.Stream* to provide a common set of methods and properties for managing streams in a generic way, much like *InStream* and *OutStream* in C/AL.

There are two primary streaming types in the .NET Framework. These facilitate specific data streaming scenarios. These types are as follows.

.NET Framework Type	Description
System.IO.MemoryStream	Manages streams whose backing data store is memory.
System.IO.FileStream	Exposes a stream programming interface around files that allow stream-based read and write access to files that are stored in the file system.

 **Note:** There are many other less frequently used stream types in the .NET Framework that address other specific scenarios, such as data compression, network transfer, printing, or encryption.

Pass Streams between C/AL and .NET

In C/AL, there are two types of stream types. The *InStream* type handles reading the data from the underlying data source. The *OutStream* type handles writing data into the underlying data source. In C/AL, those two data types are incompatible, and they expose different sets of functions. Variables of these two types cannot be assigned between the types.

Unlike C/AL, there is one base stream type, the *System.IO.Stream*. This stream type handles data access in both directions. Variables of this kind can be freely assigned to one another, regardless of which direction of data access they are intended to support.

DotNet variables that reference the System.IO.Stream or one of its descendants can be assigned to variables of type InStream and OutStream. The opposite is also true. Usually, the DotNet variables of System.IO.Stream type can be used to replace InStream or OutStream variables in C/AL code.



Note: You cannot use variables of System.IO.Stream type as parameters to CREATEINSTREAM or CREATEOUTSTREAM functions. In those two instances, the actual InStream or OutStream variable must be used. For all other purposes, you can replace the InStream and OutStream with System.IO.Stream.

Following are typical situations when you use .NET Framework stream types in C/AL:

- To import or export data to or from Microsoft Dynamics NAV 2013 database directly from or to a .NET Framework object through XMLports
- To stream data from .NET Framework objects to files or BLOB fields in Microsoft Dynamics NAV 2013 database
- To pass data between different .NET Framework objects

Using System.IO.Stream with XMLports

You can use variables of System.IO.Stream type, or its descendants, to specify the source or destination of XMLports by using **SETSOURCE** and **SETDESTINATION** functions calls. For both functions to work, the DotNet variable of type System.IO.Stream first must be instantiated. In addition, you can provide an instance of a System.IO.Stream descendant to **IMPORT** or **EXPORT** functions of XMLPORT system object.

The following example shows how to export data from an XMLport to a file using System.IO.MemoryStream.

Stream to file from XMLport through .NET Framework

```
// Stream is System.IO.Stream

// MemStream is System.IO.MemoryStream

Stream := MemStream.MemoryStream;

XMLPORT.EXPORT(XMLPORT::"Export Contact",Stream);

// FileContact is File

// Outstr is OutStream
```

Module 11: Microsoft .NET Framework Interoperability

```
FileContact.CREATE('C:\Temp\Contacts.xml');  
  
FileContact.CREATEOUTSTREAM(Outstr);  
  
Stream.Seek(0,0);  
  
COPYSTREAM(Outstr,Stream);  
  
FileContact.CLOSE;
```

Before you use .NET Framework streams with read operations or with **COPYSTREAM** function, it is important to position the stream at the beginning by using the Seek method of System.IO.Stream type. In the example that was mentioned earlier, the **COPYSTREAM** function did not copy any data from the System.IO.Stream object into the Outstr object. This is because after the stream was written to use **XMLPORT.EXPORT** function, the stream is positioned at the end.



Note: *System.IO.Stream object can be used interchangeably to replace both InStream and OutStream objects. The same DotNet variable of this .NET Framework type can act as both the input and output stream with the same instance of the System.IO.Stream object.*

Passing C/AL Streams to .NET Framework Objects

You can use InStream and OutStream C/AL types to replace System.IO.Stream type when passing variables as parameters from C/AL to .NET Framework objects. When passing InStream and OutStream variables to .NET Framework objects in this manner, you must understand how the .NET Framework code uses the stream object that you passed to it. If the .NET Framework object *reads* the data, you should pass the InStream. If the .NET Framework object *writes* the data, you should pass the OutStream.

Both InStream and OutStream map to System.IO.Stream, and the .NET Framework does not differentiate between read or write stream types. If you pass the OutStream object to replace an InStream object, or if the code within the .NET Framework object being called does not explicitly position the stream object to the beginning, the read operation fails. When an OutStream object is mapped to the System.IO.Stream, the System.IO.Stream object is always positioned at the end of the stream.

For example, you can use an InStream object to pass data from Microsoft Dynamics NAV 2013 database to an XDocument .NET Framework object.

Use InStream instead of System.IO.Stream in XDocument.Load method

```
// ObjectMetadata is Record 2000000071, Object Metadata
// Instr is InStream
// XDoc is Dot Net, System.Xml.Linq.XDocument

ObjectMetadata.GET(ObjectMetadata."Object Type"::Page,22);

ObjectMetadata.CALCFIELDS(Metadata);

ObjectMetadata.Metadata.CREATEINSTREAM(Instr);

XDoc := XDoc.Load(Instr);

MESSAGE('%1',XDoc.ToString);
```

Demonstration: Use Streams to Import through XMLports

The following demonstration shows how to convert a string variable into a stream. You then use the resulting **System.IO.MemoryStream** instance as the source for the **XMLPORT.IMPORT** function to import plain text data into Microsoft Dynamics NAV 2013 database.

Demonstration Steps

1. Create a new codeunit and save it as codeunit 90015, **Import Fault Codes From String**.
 - a. Click **Tools > Object Designer**.
 - b. Click **Codeunit**, and then click **New**.
 - c. In the **C/AL Editor** window, click **File > Save**.
 - d. In the **Save As** window, in the **ID** field, type "90015".
 - e. In the **Name** field, type "Import Fault Codes From String".
 - f. Click **OK**.

2. Declare the variables.
 - a. Click **View > C/AL Globals**
 - b. On the **Variables** tab, declare the following variables.

Name	Data Type	Subtype
MemStream	DotNet	System.IO.MemoryStream
Encoding	DotNet	System.Text.ASCIIEncoding

Module 11: Microsoft .NET Framework Interoperability

Name	Data Type	Subtype
Data	Text	
Tab	Char	
Cr	Char	
Lf	Char	



Note: You can find the `System.IO.MemoryStream` class in the `mscorlib` assembly.

3. In OnRun trigger, write code that assigns the following values: 9 to Tab, 13 to Cr and 15 to Lf. Then write code that stores three lines of plain text data, according to the format that is defined in XMLport 5901, **Import IRIS to Fault Codes**.
 - a. In the OnRun trigger, enter the following code.

Defining the plain text data for import

```
Tab := 9;

Cr := 13;

Lf := 10;

Data :=

STRSUBSTNO('%1%2Description for %1%2%3%4','TEST1',Tab,Cr,Lf) +

STRSUBSTNO('%1%2Description for %1%2%3%4','TEST2',Tab,Cr,Lf) +

STRSUBSTNO('%1%2Description for %1%2%3%4','TEST3',Tab);
```

4. Append code to OnRun trigger, to convert the string to a `System.IO.MemoryString`, by using the `System.Text.Encoding.AsciiEncoding` type. Then import the resulting stream into Microsoft Dynamics NAV 2013 database through XMLport 5901, **Import IRIS to Fault Codes**.
 - a. Append the following code to OnRun trigger:

Convert string to stream, and import the stream

```
MemStream :=

MemStream.MemoryStream(

Encoding.ASCII.GetBytes(Data));
```

```
MemStream.Seek(0,0);  
  
XMLPORT.IMPORT(  
  
XMLPORT::"Import IRIS to Fault Codes",MemStream);
```

5. Run the codeunit, and verify that the data was imported into table **5918, Fault Code**.
 - a. Click **File > Run**.
 - b. Click **Tools > Object Designer**, and then click **Table**.
 - c. Select table **5918, Fault Code**, and then click **Run**.
 - d. In the **Edit – Fault Code** window, scroll to the bottom and verify that the three lines were added.

Module Review

Module Review and Takeaways

Microsoft .NET Framework is a powerful and rich programming framework that lets you access a wide choice of types and methods to extend the functionality of Microsoft Dynamics NAV 2013 customizations. With .NET Framework Interoperability you can integrate with the Framework Class Library that is provided by Microsoft, with third-party applications or objects built in addition to .NET Framework. You can integrate to your own .NET Framework objects that you developed in addition to the .NET Framework by using Visual Studio. You use the DotNet type Microsoft Dynamics NAV 2013 to reference the .NET Framework types.

You can use .NET Framework Interoperability to extend the functionality of both the Microsoft Dynamics NAV 2013 Server and the client. It is even possible to share data between the server and the client through serialization.

When declaring global DotNet variables, you can subscribe to any events that are published by the referenced .NET Framework type. The events in the .NET Framework can be synchronous or asynchronous. Special precaution steps must be taken if subscribing to asynchronous events.

The .NET Framework uses a much wider range of built-in data types. Even though certain types are converted automatically to C/AL types, not all .NET Framework types map directly to simple C/AL types.

Pay special attention when you map numeric data. The .NET Framework String and DateTime types are not converted automatically to C/AL types. This lets you use the rich set of functions to manipulate text and dates.

C/AL includes several functions that facilitate coding with DotNet variables without developing custom .NET Framework types. Use these functions for simple tasks, such as verifying whether a DotNet variable references a valid object, or to return specific .NET Framework type information.

In Microsoft Dynamics NAV 2013, there is strong interoperability between C/AL and the .NET Framework stream types. This lets you easily pass the data between InStream and OutStream and the .NET Framework types, such as System.IO.MemoryStream, to import and export data to and from Microsoft Dynamics NAV 2013.

Test Your Knowledge

Test your knowledge with the following questions.

1. Variables of DotNet type can target the Microsoft Dynamics NAV server only.

True

False

2. Which property do you set to automatically create event triggers for events of a DotNet variable?

3. You can only call the default constructor of a .NET class. C/AL does not support overloaded constructors.

True

False

4. Which function do you call to test at the run time whether you can use a .NET class in your code?

GETDOTNETTYPE

CANLOADTYPE

You cannot test this at run time. If you cannot use a .NET class, the compile-time error occurs.

5. Which function can you call to test if a DotNet variable references a valid instance of an object?

Module 11: Microsoft .NET Framework Interoperability

6. .NET Framework strings are Unicode. However, when you map a .NET string to C/AL text data type, Unicode information is lost, and SQL Server database collation rules determine how the string is converted to text.

True

False

7. Which .NET Framework class can you use in place of InStream or OutStream types in C/AL code?

8. In C/AL use native .NET Framework classes in and any .NET Framework classes that you develop using Microsoft Visual Studio.

True

False

9. If you want to use your custom .NET Framework class from C/AL code to run on the server, where do you need to deploy it?

Global Assembly Cache on the server or Service\Add-ins directory on the server.

Global Assembly Cache on both the client and the server or Service\Add-ins directory on both the client and the server.

Service\Add-ins directory on the server only. You cannot use classes from the Global Assembly Cache in C/AL.

Test Your Knowledge Solutions

Module Review and Takeaways

1. Variables of DotNet type can target the Microsoft Dynamics NAV server only.
 True
 False

2. Which property do you set to automatically create event triggers for events of a DotNet variable?

MODEL ANSWER:

WithEvents

3. You can only call the default constructor of a .NET class. C/AL does not support overloaded constructors.
 True
 False
4. Which function do you call to test at the run time whether you can use a .NET class in your code?
 GETDOTNETTYPE
 CANLOADTYPE
 You cannot test this at run time. If you cannot use a .NET class, the compile-time error occurs.

5. Which function can you call to test if a DotNet variable references a valid instance of an object?

MODEL ANSWER:

ISNULL

6. .NET Framework strings are Unicode. However, when you map a .NET string to C/AL text data type, Unicode information is lost, and SQL Server database collation rules determine how the string is converted to text.
 True
 False

Module 11: Microsoft .NET Framework Interoperability

7. Which .NET Framework class can you use in place of InStream or OutStream types in C/AL code?

MODEL ANSWER:

System.IO.Stream

8. In C/AL use native .NET Framework classes in and any .NET Framework classes that you develop using Microsoft Visual Studio.

True

False

9. If you want to use your custom .NET Framework class from C/AL code to run on the server, where do you need to deploy it?

Global Assembly Cache on the server or Service\Add-ins directory on the server.

Global Assembly Cache on both the client and the server or Service\Add-ins directory on both the client and the server.

Service\Add-ins directory on the server only. You cannot use classes from the Global Assembly Cache in C/AL.

