Contents

Interoperating with unmanaged code

3/25/2019 • 2 minutes to read • Edit Online

The .NET Framework promotes interaction with COM components, COM+ services, external type libraries, and many operating system services. Data types, method signatures, and error-handling mechanisms vary between managed and unmanaged object models. To simplify interoperation between .NET Framework components and unmanaged code and to ease the migration path, the common language runtime conceals from both clients and servers the differences in these object models.

Code that executes under the control of the runtime is called managed code. Conversely, code that runs outside the runtime is called unmanaged code. COM components, ActiveX interfaces, and Windows API functions are examples of unmanaged code.

In this section

Exposing COM Components to the .NET Framework
Describes how to use COM components from .NET Framework applications.

Exposing .NET Framework Components to COM
Describes how to use .NET Framework components from COM applications.

Consuming Unmanaged DLL Functions
Describes how to call unmanaged DLL functions using platform invoke.

Interop Marshaling
Describes marshaling for COM interop and platform invoke.

How to: Map HRESULTs and Exceptions
Describes the mapping between exceptions and HRESULTs.

COM Wrappers
Describes the wrappers provided by COM interop.

Type Equivalence and Embedded Interop Types
Describes how type information for COM types is embedded in assemblies, and how the common language runtime determines the equivalence of embedded COM types.

How to: Generate Primary Interop Assemblies Using Tlbimp.exe
Describes how to produce primary interop assemblies using *Tlbimp.exe* (Type Library Importer).

How to: Register Primary Interop Assemblies
Describes how to register the primary interop assemblies before you can reference them in your projects.

Registration-Free COM Interop
Describes how COM interop can activate components without using the Windows registry.

How to: Configure .NET Framework-Based COM Components for Registration-Free Activation
Describes how to create an application manifest and how to create and embed a component manifest.

Exposing COM Components to the .NET Framework

5/9/2019 • 2 minutes to read • Edit Online

This section summarizes the process needed to expose an existing COM component to managed code. For details about writing COM servers that tightly integrate with the .NET Framework, see Design Considerations for Interoperation.

Existing COM components are valuable resources in managed code as middle-tier business applications or as isolated functionality. An ideal component has a primary interop assembly and conforms tightly to the programming standards imposed by COM.

To expose COM components to the .NET Framework

1. Import a type library as an assembly.

   The common language runtime requires metadata for all types, including COM types. There are several ways to obtain an assembly containing COM types imported as metadata.

2. Use COM types in managed Code.

   You can inspect COM types, activate instances, and invoke methods on the COM object the same way you do for any managed type.

3. Compile an interop project.

   The Windows Software Development Kit (SDK) provides compilers for several languages compliant with the Common Language Specification (CLS), including Visual Basic, C#, and C++.

4. Deploy an interop application.

   Interop applications are best deployed as strong-named, signed assemblies in the global assembly cache.

See also

- Interoperating with Unmanaged Code
- Design Considerations for Interoperation
- COM Interop Sample: .NET Client and COM Server
- Language Independence and Language-Independent Components
- Gacutil.exe (Global Assembly Cache Tool)

Importing a Type Library as an Assembly

4/28/2019 • 3 minutes to read • Edit Online

COM type definitions usually reside in a type library. In contrast, CLS-compliant compilers produce type metadata in an assembly. The two sources of type information are quite different. This topic describes techniques for generating metadata from a type library. The resulting assembly is called an interop assembly, and the type information it contains enables .NET Framework applications to use COM types.

There are two ways to make this type information available to your application:

- Using design-time-only interop assemblies: Beginning with the .NET Framework 4, you can instruct the compiler to embed type information from the interop assembly into your executable. The compiler embeds only the type information that your application uses. You do not have to deploy the interop assembly with your application. This is the recommended technique.
- Deploying interop assemblies: You can create a standard reference to the interop assembly. In this case, the interop assembly must be deployed with your application. If you employ this technique, and you are not using a private COM component, always reference the primary interop assembly (PIA) published by the author of the COM component you intend to incorporate in your managed code. For more information about producing and using primary interop assemblies, see Primary Interop Assemblies.

When you use design-time-only interop assemblies, you can embed type information from the primary interop assembly published by the author of the COM component. However, you do not have to deploy the primary interop assembly with your application.

Using design-time-only interop assemblies reduces the size of your application, because most applications do not use all the features of a COM component. The compiler is very efficient when it embeds type information; if your application uses only some of the methods on a COM interface, the compiler does not embed the unused methods. When an application that has embedded type information interacts with another such application, or interacts with an application that uses a primary interop assembly, the common language runtime uses type equivalence rules to determine whether two types with the same name represent the same COM type. You do not have to know these rules to use COM objects. However, if you are interested in the rules, see Type Equivalence and Embedded Interop Types.

Generating Metadata

COM type libraries can be stand-alone files that have a .tlb extension, such as Loanlib.tlb. Some type libraries are embedded in the resource section of a .dll or .exe file. Other sources of type library information are .olb and .ocx files.

After you locate the type library that contains the implementation of your target COM type, you have the following options for generating an interop assembly containing type metadata:

- Visual Studio

  Visual Studio automatically converts COM types in a type library to metadata in an assembly. For instructions, see How to: Add References to Type Libraries.

- Type Library Importer (Tlbimp.exe)

  The Type Library Importer provides command-line options to adjust metadata in the resulting interop file, imports types from an existing type library, and generates an interop assembly and a namespace. For instructions, see How to: Generate Interop Assemblies from Type Libraries.

- System.Runtime.InteropServices.TypeLibConverter class

  This class provides methods to convert coclasses and interfaces in a type library to metadata within an assembly. It produces the same metadata output as Tlbimp.exe. However, unlike Tlbimp.exe, the TypeLibConverter class can convert an in-memory type library to metadata.

- Custom wrappers

  When a type library is unavailable or incorrect, one option is to create a duplicate definition of the class or interface in managed source code. You then compile the source code with a compiler that targets the runtime to produce metadata in an assembly.

  To define COM types manually, you must have access to the following items:

  - Precise descriptions of the coclasses and interfaces being defined.

  - A compiler, such as the C# compiler, that can generate the appropriate .NET Framework class definitions.

  - Knowledge of the type library-to-assembly conversion rules.

  Writing a custom wrapper is an advanced technique. For additional information about how to generate a custom wrapper, see Customizing Standard Wrappers.

For more information about the COM interop import process, see Type Library to Assembly Conversion Summary.

See also

- TypeLibConverter
- Exposing COM Components to the .NET Framework
- Type Library to Assembly Conversion Summary
- Tlbimp.exe (Type Library Importer)
- Customizing Standard Wrappers
- Using COM Types in Managed Code
- Compiling an Interop Project
- Deploying an Interop Application
- How to: Add References to Type Libraries
- How to: Generate Interop Assemblies from Type Libraries

How to: Add References to Type Libraries

4/9/2019 • 2 minutes to read • Edit Online

Visual Studio generates an interop assembly containing metadata when you add a reference to a type library. If a primary interop assembly is available, Visual Studio uses the existing assembly before generating a new interop assembly.

To add a reference to a type library in Visual Studio

1. Install the COM DLL or EXE file on your computer, unless a Windows Setup.exe file performs the installation for you.

2. Choose **Project**, **Add Reference**.

3. In the Reference Manager, choose **COM**.

4. Select the type library from the list, or browse for the .tlb file.

5. Choose **OK**.

6. In Solution Explorer, open the shortcut menu for the reference you just added, and then choose **Properties**.

7. In the **Properties** window, make sure that the **Embed Interop Types** property is set to **True**. This causes Visual Studio to embed type information for COM types in your executables, eliminating the need to deploy primary interop assemblies with your app.

NOTE

The menu and dialog box options may vary depending on the version of Visual Studio you're using.

To add a reference to a type library for command-line compilation

1. Generate an interop assembly as described in How to: Generate Interop Assemblies from Type Libraries.

2. Use the /link (C# Compiler Options) or /link (Visual Basic) compiler option with the interop assembly name to embed type information for COM types in your executables.

See also

- Importing a Type Library as an Assembly
- Exposing COM Components to the .NET Framework
- Walkthrough: Embedding Types from Managed Assemblies in Visual Studio (C#)
- Walkthrough: Embedding Types from Managed Assemblies in Visual Studio (Visual Basic)
- /link (C# Compiler Options)
- /link (Visual Basic)

How to: Generate Interop Assemblies from Type Libraries

4/9/2019 • 2 minutes to read • Edit Online

The Type Library Importer (Tlbimp.exe) is a command-line tool that converts the coclasses and interfaces contained in a COM type library to metadata. This tool creates an interop assembly and namespace for the type information automatically. After the metadata of a class is available, managed clients can create instances of the COM type and call its methods, just as if it were a .NET instance. Tlbimp.exe converts an entire type library to metadata at once and cannot generate type information for a subset of the types defined in a type library.

To generate an interop assembly from a type library

1. Use the following command:

   tlbimp <*type-library-file*>

   Adding the **/out:** switch produces an interop assembly with an altered name, such as LOANLib.dll. Altering the interop assembly name can help distinguish it from the original COM DLL and prevent problems that can occur from having duplicate names.

Example

The following command produces the Loanlib.dll assembly in the `Loanlib` namespace.

```
tlbimp Loanlib.tlb
```

The following command produces an interop assembly with an altered name (LOANLib.dll).

```
tlbimp LoanLib.tlb /out: LOANLib.dll
```

See also

- Importing a Type Library as an Assembly
- Exposing COM Components to the .NET Framework

Compiling an Interop Project

5/15/2019 • 2 minutes to read • Edit Online

COM interop projects that reference one or more assemblies containing imported COM types are compiled like any other managed project. You can reference interop assemblies in a development environment such as Visual Studio, or you can reference them when you use a command-line compiler. In either case, to compile properly, the interop assembly must be in the same directory as the other project files.

There are two ways to reference interop assemblies:

- Embedded interop types: Beginning with the .NET Framework 4 and Visual Studio 2010, you can instruct the compiler to embed type information from an interop assembly into your executable. This is the recommended technique.
- Deploying interop assemblies: You can create a standard reference to an interop assembly. In this case, the interop assembly must be deployed with your application.

The differences between these two techniques are discussed in greater detail in Using COM Types in Managed Code.

Embedding interop types with Visual Studio is demonstrated in Walkthrough: Embedding Types from Managed Assemblies in Visual Studio (C#), and Walkthrough: Embedding Types from Managed Assemblies in Visual Studio (Visual Basic).

To reference an interop assembly with a command-line compiler and embed type information in your executables, use the /link (C# Compiler Options) or the /link (Visual Basic) compiler switch and specify the name of the interop assembly.

NOTE

Visual C++ applications cannot embed type information, but they can interoperate with applications or add-ins that do.

To compile an application that includes a primary interop assembly when it is deployed, use the **/reference** compiler switch and specify the name of the interop assembly.

See also

- Exposing COM Components to the .NET Framework
- Language Independence and Language-Independent Components
- Using COM Types in Managed Code
- Walkthrough: Embedding Types from Managed Assemblies in Visual Studio (C#)
- Walkthrough: Embedding Types from Managed Assemblies in Visual Studio (Visual Basic)
- Importing a Type Library as an Assembly

Deploying an Interop Application

4/28/2019 • 2 minutes to read • Edit Online

An interop application typically includes a .NET client assembly, one or more interop assemblies representing distinct COM type libraries, and one or more registered COM components. Visual Studio and the Windows Software Development Kit (SDK) provide tools to import and convert a type library to an interop assembly, as discussed in Importing a Type Library as an Assembly. There are two ways to deploy an interop application:

- By using embedded interop types: Beginning with the .NET Framework 4, you can instruct the compiler to embed type information from an interop assembly into your executable. The compiler embeds only the type information that your application uses. You do not have to deploy the interop assembly with your application. This is the recommended technique.
- By deploying interop assemblies: You can create a standard reference to an interop assembly. In this case, the interop assembly must be deployed with your application. If you employ this technique, and you are not using a private COM component, always reference the primary interop assembly (PIA) published by the author of the COM component you intend to incorporate in your managed code. For more information about producing and using primary interop assemblies, see Primary Interop Assemblies.

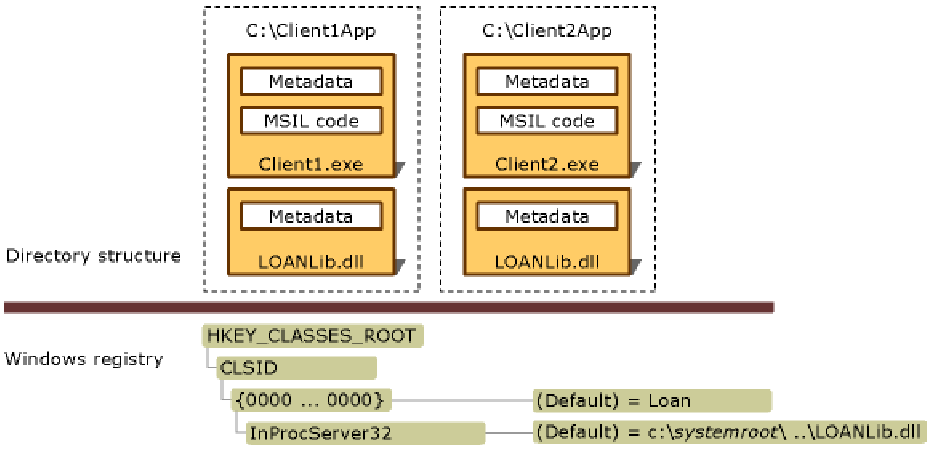
If you use embedded interop types, deployment is simple and straightforward. There is nothing special you need to do. The rest of this article describes the scenarios for deploying interop assemblies with your application.

Deploying Interop Assemblies

Assemblies can have strong names. A strong-named assembly includes the publisher's public key, which provides a unique identity. Assemblies that are produced by the Type Library Importer (Tlbimp.exe) can be signed by the publisher by using the **/keyfile** option. You can install signed assemblies into the global assembly cache. Unsigned assemblies must be installed on the user's machine as private assemblies.

Private Assemblies

To install an assembly to be used privately, both the application executable and the interop assembly that contains imported COM types must be installed in the same directory structure. The following illustration shows an unsigned interop assembly to be used privately by Client1.exe and Client2.exe, which reside in separate application directories. The interop assembly, which is called LOANLib.dll in this example, is installed twice.



All COM components associated with the application must be installed in the Windows registry. If Client1.exe and Client2.exe in the illustration are installed on different computers, you must register the COM components on both computers.

Shared Assemblies

Assemblies that are shared by multiple applications should be installed in a centralized repository called the global assembly cache. .NET clients can access the same copy of the interop assembly, which is signed and installed in the global assembly cache. For more information about producing and using primary interop assemblies, see Primary Interop Assemblies.

See also

- Exposing COM Components to the .NET Framework
- Importing a Type Library as an Assembly
- Using COM Types in Managed Code
- Compiling an Interop Project

COM Interop Sample: .NET Client and COM Server

4/8/2019 • 3 minutes to read • Edit Online

This sample demonstrates how a .NET client, built to access a COM server, creates an instance of a COM coclass and calls class members to perform mortgage calculations.

In this example, the client creates and calls an instance of the **Loan** coclass, passes four arguments (one of those four being equal to zero) to the instance, and displays the computations. Code fragments from this sample appear throughout this section.

.NET Client

```
Imports System
Imports Microsoft.VisualBasic
Imports LoanLib

Public Class LoanApp

    Public Shared Sub Main()
        Dim Args As String()
        Args = System.Environment.GetCommandLineArgs()

        Dim ln As New Loan()

        If Args.Length < 5 Then
            Console.WriteLine("Usage: ConLoan Balance Rate Term Payment")
            Console.WriteLine(" Either Balance, Rate, Term, or Payment " _
                              & "must be 0")
            Exit Sub
        End If

        ln.OpeningBalance = Convert.ToDouble(Args(1))
        ln.Rate = Convert.ToDouble(Args(2)) / 100.0
        ln.Term = Convert.ToInt16(Args(3))
        ln.Payment = Convert.ToDouble(Args(4))

        If ln.OpeningBalance = 0.0 Then
            ln.ComputeOpeningBalance()
        End If
        If ln.Rate = 0.0 Then
            ln.ComputeRate()
        End If
        If ln.Term = 0 Then
            ln.ComputeTerm()
        End If
        If ln.Payment = 0.0 Then
            ln.ComputePayment()
        End If
        Console.WriteLine("Balance = {0,10:0.00}", ln.OpeningBalance)
        Console.WriteLine("Rate    = {0,10:0.0%}", ln.Rate)
        Console.WriteLine("Term    = {0,10:0.00}", ln.Term)
        Console.WriteLine("Payment = {0,10:0.00}" & ControlChars.Cr, _
                          ln.Payment)

        Dim MorePmts As Boolean
        Dim Balance As Double = 0.0
        Dim Principal As Double = 0.0
        Dim Interest As Double = 0.0

        Console.WriteLine("{0,4}{1,10}{2,12}{3,10}{4,12}", _
           "Nbr", "Payment", "Principal", "Interest", "Balance")
        Console.WriteLine("{0,4}{1,10}{2,12}{3,10}{4,12}", _
           "---", "-------", "---------", "--------", "-------")

        MorePmts = ln.GetFirstPmtDistribution(ln.Payment, Balance, _
           Principal, Interest)

        Dim PmtNbr As Short

        While MorePmts

            Console.WriteLine("{0,4}{1,10:0.00}{2,12:0.00}{3,10:0.00}" _
              & "{4,12:0.00}", PmtNbr, ln.Payment, Principal, Interest,
              Balance)
            MorePmts = ln.GetNextPmtDistribution(ln.Payment, Balance, _
              Principal, Interest)
            PmtNbr += CShort(1)
        End While
    End Sub
End Class
```

```
using System;
using LoanLib;

public class LoanApp {
   public static void Main(String[] Args) {

      Loan ln = new Loan();

      if (Args.Length < 4)
      {
         Console.WriteLine("Usage: ConLoan Balance Rate Term Payment");
         Console.WriteLine("    Either Balance, Rate, Term, or Payment
            must be 0");
         return;
      }

      ln.OpeningBalance = Convert.ToDouble(Args[0]);
      ln.Rate = Convert.ToDouble(Args[1])/100.0;
      ln.Term = Convert.ToInt16(Args[2]);
      ln.Payment = Convert.ToDouble(Args[3]);

      if (ln.OpeningBalance == 0.00) ln.ComputeOpeningBalance();
      if (ln.Rate == 0.00) ln.ComputeRate();
      if (ln.Term == 0) ln.ComputeTerm();
      if (ln.Payment == 0.00) ln.ComputePayment();

      Console.WriteLine("Balance = {0,10:0.00}", ln.OpeningBalance);
      Console.WriteLine("Rate    = {0,10:0.0%}", ln.Rate);
      Console.WriteLine("Term    = {0,10:0.00}", ln.Term);
      Console.WriteLine("Payment = {0,10:0.00}\n", ln.Payment);

      bool MorePmts;
      double Balance = 0.0;
      double Principal = 0.0;
      double Interest = 0.0;

      Console.WriteLine("{0,4}{1,10}{2,12}{3,10}{4,12}", "Nbr", "Payment",
        "Principal", "Interest", "Balance");
      Console.WriteLine("{0,4}{1,10}{2,12}{3,10}{4,12}", "---", "-------",
        "---------", "--------", "-------");

      MorePmts = ln.GetFirstPmtDistribution(ln.Payment, out Balance,
        out Principal, out Interest);

      for (short PmtNbr = 1; MorePmts; PmtNbr++) {
         Console.WriteLine("{0,4}{1,10:0.00}{2,12:0.00}{3,10:0.00}
           {4,12:0.00}", PmtNbr, ln.Payment, Principal, Interest,
            Balance);
         MorePmts = ln.GetNextPmtDistribution(ln.Payment, ref Balance,
           out Principal, out Interest);
      }
    }
}
```

COM Server

```
// Loan.cpp : Implementation of CLoan
#include "stdafx.h"
#include "math.h"
#include "LoanLib.h"
#include "Loan.h"

static double Round(double value, short digits);

STDMETHODIMP CLoan::get_OpeningBalance(double *pVal)
```

```
{
    *pVal = OpeningBalance;
    return S_OK;
}

STDMETHODIMP CLoan::put_OpeningBalance(double newVal)
{
    OpeningBalance = newVal;
    return S_OK;
}

STDMETHODIMP CLoan::get_Rate(double *pVal)
{
    *pVal = Rate;
    return S_OK;
}

STDMETHODIMP CLoan::put_Rate(double newVal)
{
    Rate = newVal;
    return S_OK;
}

STDMETHODIMP CLoan::get_Payment(double *pVal)
{
    *pVal = Payment;
    return S_OK;
}

STDMETHODIMP CLoan::put_Payment(double newVal)
{
    Payment = newVal;
    return S_OK;
}

STDMETHODIMP CLoan::get_Term(short *pVal)
{
    *pVal = Term;
    return S_OK;
}

STDMETHODIMP CLoan::put_Term(short newVal)
{
    Term = newVal;
    return S_OK;
}

STDMETHODIMP CLoan::ComputePayment(double *pVal)
{
    Payment = Round(OpeningBalance * (Rate /
        (1 - pow((1 + Rate), -Term))),2);
    *pVal = Payment;
    return S_OK;
}

STDMETHODIMP CLoan::ComputeOpeningBalance(double *pVal)
{
    OpeningBalance = Round(Payment / (Rate /
        (1 - pow((1 + Rate), -Term))),2);
    *pVal = OpeningBalance ;
    return S_OK;
}

STDMETHODIMP CLoan::ComputeRate(double *pVal)
{
    double DesiredPayment = Payment;

    for (Rate = 0.001; Rate < 28.0; Rate += 0.001)
    {
        Payment = Round(OpeningBalance * (Rate /
         (1 - pow((1 + Rate), -Term))),2);

        if (Payment >= DesiredPayment)
            break;
    }

    *pVal = Rate;
    return S_OK;
}

STDMETHODIMP CLoan::ComputeTerm(short *pVal)
{
    double DesiredPayment = Payment;
    for (Term = 1; Term < 480 ; Term ++)
    {
        Payment = Round(OpeningBalance * (Rate /
         (1 - pow((1 + Rate), -Term))),2);
        if (Payment <= DesiredPayment)
            break;
    }
    *pVal = Term;
    return S_OK;
}

STDMETHODIMP CLoan::GetFirstPmtDistribution(double PmtAmt, double *Balance, double *PrinPortion, double
*IntPortion, VARIANT_BOOL *pVal)
{
    *Balance = OpeningBalance;
    GetNextPmtDistribution(PmtAmt, Balance, PrinPortion, IntPortion,
      pVal);
    return S_OK;
}

STDMETHODIMP CLoan::GetNextPmtDistribution(double PmtAmt, double *Balance, double *PrinPortion, double
*IntPortion, VARIANT_BOOL *pVal)
{
    *IntPortion = Round(*Balance * Rate, 2);
    *PrinPortion = Round(PmtAmt - *IntPortion, 2);
    *Balance = Round(*Balance - *PrinPortion, 2);

    if (*Balance <= 0.0)
        *pVal = FALSE;
    else
        *pVal = TRUE;
    return S_OK;
}

STDMETHODIMP CLoan::get_RiskRating(BSTR *pVal)
{
    *pVal = (BSTR)RiskRating;
    return S_OK;
}

STDMETHODIMP CLoan::put_RiskRating(BSTR newVal)
{
    RiskRating = newVal;
    return S_OK;
}
static double Round(double value, short digits)
{
    double factor = pow(10, digits);
    return floor(value * factor + 0.5)/factor;
}
```

See also

- Exposing COM Components to the .NET Framework

Exposing .NET Framework Components to COM

4/28/2019 • 2 minutes to read • Edit Online

Writing a .NET type and consuming that type from unmanaged code are distinct activities for developers. This section describes several tips for writing managed code that interoperates with COM clients:

- Qualifying .NET types for interoperation.

  All managed types, methods, properties, fields, and events that you want to expose to COM must be public. Types must have a public default constructor, which is the only constructor that can be invoked through COM.

- Applying interop attributes.

  Custom attributes within managed code can enhance the interoperability of a component.

- Packaging an assembly for COM.

  COM developers might require that you summarize the steps involved in referencing and deploying your assemblies.

Additionally, this section identifies the tasks related to consuming a managed type from a COM client.

To consume a managed type from COM

1. Register assemblies with COM.

   Types in an assembly (and type libraries) must be registered at design time. If an installer does not register the assembly, instruct COM developers to use Regasm.exe.

2. Reference .NET types from COM.

   COM developers can reference types in an assembly using the same tools and techniques they use today.

3. Call a .NET object.

   COM developers can call methods on the .NET object the same way they call methods on any unmanaged type. For example, the COM **CoCreateInstance** API activates .NET objects.

4. Deploy an application for COM access.

   A strong-named assembly can be installed in the global assembly cache and requires a signature from its publisher. Assemblies that are not strong named must be installed in the application directory of the client.

See also

- Interoperating with Unmanaged Code
- COM Interop Sample: COM Client and .NET Server

Qualifying .NET Types for Interoperation

4/28/2019 • 2 minutes to read • Edit Online

If you intend to expose types in an assembly to COM applications, consider the requirements of COM interop at design time. Managed types (class, interface, structure, and enumeration) seamlessly integrate with COM types when you adhere to the following guidelines:

- Classes should implement interfaces explicitly.

  Although COM interop provides a mechanism to automatically generate an interface containing all members of the class and the members of its base class, it is far better to provide explicit interfaces. The automatically generated interface is called the class interface. For guidelines, see Introducing the class interface.

  You can use Visual Basic, C#, and C++ to incorporate interface definitions in your code, instead of having to use Interface Definition Language (IDL) or its equivalent. For syntax details, see your language documentation.

- Managed types must be public.

  Only public types in an assembly are registered and exported to the type library. As a result, only public types are visible to COM.

  Managed types expose features to other managed code that might not be exposed to COM. For instance, parameterized constructors, static methods, and constant fields are not exposed to COM clients. Further, as the runtime marshals data in and out of a type, the data might be copied or transformed.

- Methods, properties, fields, and events must be public.

  Members of public types must also be public if they are to be visible to COM. You can restrict the visibility of an assembly, a public type, or public members of a public type by applying the ComVisibleAttribute. By default, all public types and members are visible.

- Types must have a public default constructor to be activated from COM.

  Managed, public types are visible to COM. However, without a public default constructor (a constructor with no arguments), COM clients cannot create the type. COM clients can still use the type if it is activated by some other means.

- Types cannot be abstract.

  Neither COM clients nor .NET clients can create abstract types.

When exported to COM, the inheritance hierarchy of a managed type is flattened. Versioning also differs between managed and unmanaged environments. Types exposed to COM do not have the same versioning characteristics as other managed types.

See also

- ComVisibleAttribute
- Exposing .NET Framework Components to COM
- Introducing the class interface
- Applying Interop Attributes
- Packaging an Assembly for COM

Applying Interop Attributes

4/8/2019 • 4 minutes to read • Edit Online

The System.Runtime.InteropServices namespace provides three categories of interop-specific attributes: those applied by you at design time, those applied by COM interop tools and APIs during the conversion process, and those applied either by you or COM interop.

If you are unfamiliar with the task of applying attributes to managed code, see Extending Metadata Using Attributes. Like other custom attributes, you can apply interop-specific attributes to types, methods, properties, parameters, fields, and other members.

Design-Time Attributes

You can adjust the outcome of the conversion process performed by COM interop tools and APIs by using design-time attributes. The following table describes the attributes that you can apply to your managed source code. COM interop tools, on occasion, might also apply the attributes described in this table.

ATTRIBUTE	DESCRIPTION
AutomationProxyAttribute	Specifies whether the type should be marshaled using the Automation marshaler or a custom proxy and stub.
ClassInterfaceAttribute	Controls the type of interface generated for a class.
CoClassAttribute	Identifies the CLSID of the original coclass imported from a type library. COM interop tools typically apply this attribute.
ComImportAttribute	Indicates that a coclass or interface definition was imported from a COM type library. The runtime uses this flag to know how to activate and marshal the type. This attribute prohibits the type from being exported back to a type library. COM interop tools typically apply this attribute.
ComRegisterFunctionAttribute	Indicates that a method should be called when the assembly is registered for use from COM, so that user-written code can be executed during the registration process.
ComSourceInterfacesAttribute	Identifies interfaces that are sources of events for the class. COM interop tools can apply this attribute.
ComUnregisterFunctionAttribute	Indicates that a method should be called when the assembly is unregistered from COM, so that user-written code can execute during the process.
ComVisibleAttribute	Renders types invisible to COM when the attribute value equals **false**. This attribute can be applied to an individual type or to an entire assembly to control COM visibility. By default, all managed, public types are visible; the attribute is not needed to make them visible.

ATTRIBUTE	DESCRIPTION
DispIdAttribute	Specifies the COM dispatch identifier (DISPID) of a method or field. This attribute contains the DISPID for the method, field, or property it describes. COM interop tools can apply this attribute.
FieldOffsetAttribute	Indicates the physical position of each field within a class when used with the **StructLayoutAttribute**, and the **LayoutKind** is set to Explicit.
GuidAttribute	Specifies the globally unique identifier (GUID) of a class, interface, or an entire type library. The string passed to the attribute must be a format that is an acceptable constructor argument for the type **System.Guid**. COM interop tools can apply this attribute.
IDispatchImplAttribute	Indicates which **IDispatch** interface implementation the common language runtime uses when exposing dual interfaces and dispinterfaces to COM.
InAttribute	Indicates that data should be marshaled in to the caller. Can be used to attribute parameters.
InterfaceTypeAttribute	Controls how a managed interface is exposed to COM clients (Dual, IUnknown-derived, or IDispatch only). COM interop tools can apply this attribute.
LCIDConversionAttribute	Indicates that an unmanaged method signature expects an LCID parameter. COM interop tools can apply this attribute.
MarshalAsAttribute	Indicates how the data in fields or parameters should be marshaled between managed and unmanaged code. The attribute is always optional because each data type has default marshaling behavior. COM interop tools can apply this attribute.
OptionalAttribute	Indicates that a parameter is optional. COM interop tools can apply this attribute.
OutAttribute	Indicates that the data in a field or parameter must be marshaled from a called object back to its caller.
PreserveSigAttribute	Suppresses the HRESULT or retval signature transformation that normally takes place during interoperation calls. The attribute affects marshaling as well as type library exporting. COM interop tools can apply this attribute.
ProgIdAttribute	Specifies the ProgID of a .NET Framework class. Can be used to attribute classes.

ATTRIBUTE	DESCRIPTION
StructLayoutAttribute	Controls the physical layout of the fields of a class. COM interop tools can apply this attribute.

Conversion-Tool Attributes

The following table describes attributes that COM interop tools apply during the conversion process. You do not apply these attributes at design time.

ATTRIBUTE	DESCRIPTION
ComAliasNameAttribute	Indicates the COM alias for a parameter or field type. Can be used to attribute parameters, fields, or return values.
ComConversionLossAttribute	Indicates that information about a class or interface was lost when it was imported from a type library to an assembly.
ComEventInterfaceAttribute	Identifies the source interface and the class that implements the methods of the event interface.
ImportedFromTypeLibAttribute	Indicates that the assembly was originally imported from a COM type library. This attribute contains the type library definition of the original type library.
TypeLibFuncAttribute	Contains the **FUNCFLAGS** that were originally imported for this function from the COM type library.
TypeLibTypeAttribute	Contains the **TYPEFLAGS** that were originally imported for this type from the COM type library.
TypeLibVarAttribute	Contains the **VARFLAGS** that were originally imported for this variable from the COM type library.

See also

- System.Runtime.InteropServices
- Exposing .NET Framework Components to COM
- Attributes
- Qualifying .NET Types for Interoperation
- Packaging an Assembly for COM

Packaging an Assembly for COM

4/18/2019 • 2 minutes to read • Edit Online

COM developers can benefit from the following information about the managed types they plan to incorporate in their application:

- A list of types that COM applications can consume

  Some managed types are invisible to COM; some are visible but not creatable; and some are both visible and creatable. An assembly can comprise any combination of invisible, visible, not creatable, and creatable types. For completeness, identify the types in an assembly that you intend to expose to COM, especially when those types are a subset of the types exposed to the .NET Framework.

  For additional information, see Qualifying .NET Types for Interoperation.

- Versioning instructions

  Managed classes that implement the class interface (a COM interop-generated interface) are subject to versioning restrictions.

  For guidelines on using the class interface, see Introducing the class interface.

- Deployment instructions

  Strong-named assemblies that are signed by a publisher can be installed into the global assembly cache. Unsigned assemblies must be installed on the user's machine as private assemblies.

  For additional information, see Assembly Security Considerations.

- Type library inclusion

  Most types require a type library when consumed by a COM application. You can generate a type library or have COM developers perform this task. The Windows Software Development Kit (SDK) provides the following options for generating a type library:

  - Type Library Exporter
  - TypeLibConverter Class
  - Assembly Registration Tool
  - .NET Services Installation Tool

  Regardless of the mechanism you choose, only public types defined in the assembly you supply are included in the generated type library.

  You can package a type library as a separate file or embed it as Win32 resource file within a .NET-based application. Microsoft Visual Basic 6.0 performed this task for you automatically; however, when using Microsoft Visual Basic 2005, you must embed your type library manually. For instructions, see How to: Embed Type Libraries as Win32 Resources in .NET-Based Applications.

Type Library Exporter

The Type Library Exporter (Tlbexp.exe) is a command-line tool that converts the classes and interfaces contained in an assembly to a COM type library. Once the type information of the class is available, COM clients can create an instance of the .NET class and call the methods of the instance, just as if it were a COM object. Tlbexp.exe converts an entire assembly at one time. You cannot use Tlbexp.exe to generate type information for a subset of the types defined in an assembly.

TypeLibConverter Class

The TypeLibConverter class, located in the **System.Runtime.Interop** namespace, converts the classes and interfaces contained in an assembly to a COM type library. This API produces the same type information as the Type Library Exporter, described in the previous section.

The **TypeLibConverter class** implements the ITypeLibConverter.

Assembly Registration Tool

The Assembly Registration Tool (Regasm.exe) can generate and register a type library when you apply the **/tlb:** option. COM clients require that type libraries be installed in the Windows registry. Without this option, Regasm.exe only registers the types in an assembly, not the type library. Registering the types in an assembly and registering the type library are distinct activities.

.NET Services Installation Tool

The .NET Services Installation Tool (Regsvcs.exe) adds managed classes to Windows 2000 Component Services and combines several tasks within a single tool. In addition to loading and registering an assembly, Regsvcs.exe can generate, register, and install the type library into an existing COM+ 1.0 application.

See also

- TypeLibConverter
- ITypeLibConverter
- Exposing .NET Framework Components to COM
- Qualifying .NET Types for Interoperation
- Introducing the class interface
- Assembly Security Considerations
- Tlbexp.exe (Type Library Exporter)
- Registering Assemblies with COM
- How to: Embed Type Libraries as Win32 Resources in Applications

Registering Assemblies with COM

4/28/2019 • 2 minutes to read • Edit Online

You can run a command-line tool called the Assembly Registration Tool (Regasm.exe) to register or unregister an assembly for use with COM. Regasm.exe adds information about the class to the system registry so COM clients can use the .NET Framework class transparently. The RegistrationServices class provides the equivalent functionality.

A managed component must be registered in the Windows registry before it can be activated from a COM client. The following table shows the keys that Regasm.exe typically adds to the Windows registry. (000000 indicates the actual GUID value.)

GUID	DESCRIPTION	REGISTRY KEY
CLSID	Class identifier	HKEY_CLASSES_ROOT\CLSID\{000...000}
IID	Interface identifier	HKEY_CLASSES_ROOT\Interface\{000...000}
LIBID	Library identifier	HKEY_CLASSES_ROOT\TypeLib\{000...000}
ProgID	Programmatic identifier	HKEY_CLASSES_ROOT\000...000

Under the HKCR\CLSID\{0000...0000} key, the default value is set to the ProgID of the class, and two new named values, Class and Assembly, are added. The runtime reads the Assembly value from the registry and passes it on to the runtime assembly resolver. The assembly resolver attempts to locate the assembly, based on assembly information such as the name and version number. For the assembly resolver to locate an assembly, the assembly has to be in one of the following locations:

- The global assembly cache (must be a strong-named assembly).
- In the application directory. Assemblies loaded from the application path are only accessible from that application.
- Along an file path specified with the **/codebase** option to Regasm.exe.

Regasm.exe also creates the InProcServer32 key under the HKCR\CLSID\{0000...0000} key. The default value for the key is set to the name of the DLL that initializes the common language runtime (Mscoree.dll).

Examining Registry Entries

COM interop provides a standard class factory implementation to create an instance of any .NET Framework class. Clients can call **DllGetClassObject** on the managed DLL to get a class factory and create objects, just as they would with any other COM component.

For the `InprocServer32` subkey, a reference to Mscoree.dll appears in place of a traditional COM type library to indicate that the common language runtime creates the managed object.

See also

- Exposing .NET Framework Components to COM
- How to: Reference .NET Types from COM
- Calling a .NET Object
- Deploying an Application for COM Access

How to: Reference .NET Types from COM

4/28/2019 • 2 minutes to read • Edit Online

From the point of view of client and server code, the differences between COM and the .NET Framework are largely invisible. Microsoft Visual Basic clients can view a .NET object in the object browser, which exposes the object methods and syntax, properties, and fields exactly as if it were any other COM object.

The process for importing a type library is slightly more complicated for C++ clients, although you use the same tools to export metadata to a COM type library. To reference .NET object members from an unmanaged C++ client, reference the TLB file (produced with Tlbexp.exe) with the **#import** directive. When referencing a type library from C++, you must either specify the **raw_interfaces_only** option or import the definitions in the base class library, Mscorlib.tlb.

To import a library

- Specify the **raw_interfaces_only** option in the **#import** directive. For example:

  ```
  #import "..\LoanLib\LoanLib.tlb" raw_interfaces_only
  ```

  -or-

- Include an #import directive for Mscorlib.tlb. For example:

  ```
  #import "mscorlib.tlb"
  #import "..\LoanLib\LoanLib.tlb"
  ```

See also

- Exposing .NET Framework Components to COM
- Registering Assemblies with COM
- Calling a .NET Object
- Deploying an Application for COM Access

COM Interop Sample: COM Client and .NET Server

4/8/2019 • 5 minutes to read • Edit Online

This sample demonstrates the interoperation of a COM Client and a .NET Server that performs mortgage calculations. In this example, the client creates and calls an instance of the managed `Loan` class, passes four arguments (one of those four being equal to zero) to the instance, and displays the computations. Code examples from this sample appear throughout this section.

COM Client

```
// ConLoan.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#import "..\LoanLib\LoanLib.tlb" raw_interfaces_only
using namespace LoanLib;

int main(int argc, char* argv[])
{
    HRESULT hr = CoInitialize(NULL);

    ILoanPtr pILoan(__uuidof(Loan));

    if (argc < 5)
    {
        printf("Usage: ConLoan Balance Rate Term Payment\n");
        printf("    Either Balance, Rate, Term, or Payment must be 0\n");
        return -1;
    }

    double openingBalance = atof(argv[1]);
    double rate = atof(argv[2])/100.0;
    short  term = atoi(argv[3]);
    double payment = atof(argv[4]);

    pILoan->put_OpeningBalance(openingBalance);
    pILoan->put_Rate(rate);
    pILoan->put_Term(term);
    pILoan->put_Payment(payment);

    if (openingBalance == 0.00)
         pILoan->ComputeOpeningBalance(&openingBalance);
    if (rate == 0.00) pILoan->ComputeRate(&rate);
    if (term == 0) pILoan->ComputeTerm(&term);
    if (payment == 0.00) pILoan->ComputePayment(&payment);

    printf("Balance = %.2f\n", openingBalance);
    printf("Rate    = %.1f%%\n", rate*100);
    printf("Term    = %.2i\n", term);
    printf("Payment = %.2f\n", payment);

    VARIANT_BOOL MorePmts;
    double Balance = 0.0;
    double Principal = 0.0;
    double Interest = 0.0;

    printf("%4s%10s%12s%10s%12s\n", "Nbr", "Payment", "Principal", "Interest", "Balance");
    printf("%4s%10s%12s%10s%12s\n", "---", "-------", "---------",
"--------", "-------");

    pILoan->GetFirstPmtDistribution(payment, &Balance, &Principal, &Interest, &MorePmts);

    for (short PmtNbr = 1; MorePmts; PmtNbr++)
    {
        printf("%4i%10.2f%12.2f%10.2f%12.2f\n",
        PmtNbr, payment, Principal, Interest, Balance);

        pILoan->GetNextPmtDistribution(payment, &Balance, &Principal, &Interest, &MorePmts);
    }

    CoUninitialize();
    return 0;
}
```

.NET Server

```
Imports System
Imports System.Reflection

<Assembly: AssemblyKeyFile("sample.snk")>
Namespace LoanLib

    Public Interface ILoan
        Property OpeningBalance() As Double
        Property Rate() As Double
        Property Payment() As Double
        Property Term() As Short
        Property RiskRating() As String
        Function ComputePayment() As Double
        Function ComputeOpeningBalance() As Double
        Function ComputeRate() As Double
        Function ComputeTerm() As Short
        Function GetFirstPmtDistribution(PmtAmt As Double, _
           ByRef Balance As Double, ByRef PrinPortion As Double, _
           ByRef IntPortion As Double) As Boolean
        Function GetNextPmtDistribution(PmtAmt As Double, _
           ByRef Balance As Double, ByRef PrinPortion As Double, _
           ByRef IntPortion As Double) As Boolean
    End Interface

    Public Class Loan
        Implements ILoan
        Private m_openingBalance As Double
        Private m_rate As Double
        Private m_payment As Double
        Private m_term As Short
        Private m_riskRating As String

        Public Property OpeningBalance() As Double _
        Implements ILoan.OpeningBalance

            Get
                Return m_openingBalance
            End Get
            Set
                m_openingBalance = value
            End Set
        End Property

        Public Property Rate() As Double _
        Implements ILoan.Rate

            Get
                Return m_rate
            End Get
            Set
                m_rate = value
            End Set
        End Property

        Public Property Payment() As Double _
        Implements ILoan.Payment

            Get
                Return m_payment
            End Get
            Set
                m_payment = value
            End Set
        End Property

        Public Property Term() As Short _
        Implements ILoan.Term

            Get
                Return m_term
            End Get
            Set
                m_term = value
            End Set
        End Property

        Public Property RiskRating() As String _
        Implements ILoan.RiskRating

            Get
                Return m_riskRating
            End Get
            Set
                m_riskRating = value
            End Set
        End Property

        Public Function ComputePayment() As Double _
        Implements ILoan.ComputePayment

            Payment = Util.Round(OpeningBalance *(Rate / _
               (1 - Math.Pow(1 + Rate, - Term))), 2)
            Return Payment
        End Function

        Public Function ComputeOpeningBalance() As Double _
        Implements ILoan.ComputeOpeningBalance

            OpeningBalance = Util.Round(Payment /(Rate / _
               (1 - Math.Pow(1 + Rate, - Term))), 2)
            Return OpeningBalance
        End Function

        Public Function ComputeRate() As Double _
        Implements ILoan.ComputeRate

            Dim DesiredPayment As Double = Payment

            For m_rate = 0.001 To 28.0 - 0.001 Step 0.001
                Payment = Util.Round(OpeningBalance *(Rate / _
                   (1 - Math.Pow(1 + Rate, - Term))), 2)

                If Payment >= DesiredPayment Then
                    Exit For
                End If
            Next
            Return Rate
        End Function

        Public Function ComputeTerm() As Short _
        Implements ILoan.ComputeTerm

            Dim DesiredPayment As Double = Payment

            For m_term = 1 To 479
                Payment = Util.Round(OpeningBalance *(Rate / _
                   (1 - Math.Pow(1 + Rate, - Term))), 2)

                If Payment <= DesiredPayment Then
                    Exit For
                End If
            Next
            Return Term
        End Function

        Public Function GetFirstPmtDistribution(PmtAmt As Double, _
        ByRef Balance As Double, ByRef PrinPortion As Double, _
        ByRef IntPortion As Double) As Boolean _
        Implements ILoan.GetFirstPmtDistribution
```

```
        Implements ILoan.GetFirstPmtDistribution

            Balance = OpeningBalance
            Return GetNextPmtDistribution(PmtAmt, Balance, PrinPortion, _
               IntPortion)
        End Function

        Public Function GetNextPmtDistribution(PmtAmt As Double, _
        ByRef Balance As Double, ByRef PrinPortion As Double, _
        ByRef IntPortion As Double) As Boolean _
        Implements ILoan.GetNextPmtDistribution

            IntPortion = Util.Round(Balance * Rate, 2)
            PrinPortion = Util.Round(PmtAmt - IntPortion, 2)
            Balance = Util.Round(Balance - PrinPortion, 2)

            If Balance <= 0.0 Then
                Return False
            End If
            Return True
        End Function
    End Class

    Friend Class Util

        Public Shared Function Round(value As Double, digits As Short) _
                                      As Double
            Dim factor As Double = Math.Pow(10, digits)
            Return Math.Round((value * factor)) / factor
        End Function

    End Class

End Namespace
```

```
using System;
using System.Reflection;

[assembly:AssemblyKeyFile("sample.snk")]
namespace LoanLib {

    public interface ILoan {
        double OpeningBalance{get; set;}
        double Rate{get; set;}
        double Payment{get; set;}
        short  Term{get; set;}
        String RiskRating{get; set;}

        double ComputePayment();
        double ComputeOpeningBalance();
        double ComputeRate();
        short ComputeTerm();
        bool GetFirstPmtDistribution(double PmtAmt, ref double Balance,
            out double PrinPortion, out double IntPortion);
        bool GetNextPmtDistribution(double PmtAmt, ref double Balance,
            out double PrinPortion, out double IntPortion);
    }

    public class Loan : ILoan {
        private double openingBalance;
        private double rate;
        private double payment;
        private short  term;
        private String riskRating;

        public double OpeningBalance {
            get { return openingBalance; }
            set { openingBalance = value; }
        }

        public double Rate {
            get { return rate; }
            set { rate = value; }
        }

        public double Payment {
            get { return payment; }
            set { payment = value; }
        }

        public short Term {
            get { return term; }
            set { term = value; }
        }

        public String RiskRating {
            get { return riskRating; }
            set { riskRating = value; }
        }

        public double ComputePayment() {
             Payment = Util.Round(OpeningBalance * (Rate / (1 –
                        Math.Pow((1 + Rate), -Term))), 2);
             return Payment;
        }

        public double ComputeOpeningBalance() {
            OpeningBalance = Util.Round(Payment / (Rate / (1 - Math.Pow((1
                              + Rate), -Term))), 2);
             return OpeningBalance;
        }

        public double ComputeRate() {
            double DesiredPayment = Payment;

            for (Rate = 0.001; Rate < 28.0; Rate += 0.001) {
                Payment = Util.Round(OpeningBalance * (Rate / (1 –
                           Math.Pow((1 + Rate), -Term))), 2);

                if (Payment >= DesiredPayment)
                    break;
            }
            return Rate;
        }

        public short ComputeTerm() {
            double DesiredPayment = Payment;

            for (Term = 1; Term < 480 ; Term ++) {
                Payment = Util.Round(OpeningBalance * (Rate / (1 –
                           Math.Pow((1 + Rate), -Term))),2);

                if (Payment <= DesiredPayment)
                    break;
            }

            return Term;
        }

        public bool GetFirstPmtDistribution(double PmtAmt, ref double
            Balance, out double PrinPortion, out double IntPortion) {
             Balance = OpeningBalance;
             return GetNextPmtDistribution(PmtAmt, ref Balance, out
             PrinPortion, out IntPortion);
        }

        public bool GetNextPmtDistribution(double PmtAmt, ref double
        public bool GetNextPmtDistribution(double PmtAmt, ref double
           Balance, out double PrinPortion, out double IntPortion) {
            IntPortion = Util.Round(Balance * Rate, 2);
            PrinPortion = Util.Round(PmtAmt - IntPortion,2);
            Balance = Util.Round(Balance - PrinPortion,2);

            if (Balance <= 0.0)
                return false;

            return true;
        }
     }

    internal class Util {
        public static double Round(double value, short digits) {
            double factor = Math.Pow(10, digits);
            return Math.Round(value * factor) / factor;
         }
    }
}
```

See also

- Exposing .NET Framework Components to COM

Consuming Unmanaged DLL Functions

4/9/2019 • 2 minutes to read • Edit Online

Platform invoke is a service that enables managed code to call unmanaged functions implemented in dynamic link libraries (DLLs), such as those in the Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

This section introduces tasks associated with consuming unmanaged DLL functions and provides more information about platform invoke. In addition to the following tasks, there are general considerations and a link providing additional information and examples.

To consume exported DLL functions

1. Identify functions in DLLs.

   Minimally, you must specify the name of the function and name of the DLL that contains it.

2. Create a class to hold DLL functions.

   You can use an existing class, create an individual class for each unmanaged function, or create one class that contains a set of related unmanaged functions.

3. Create prototypes in managed code.

   [Visual Basic] Use the **Declare** statement with the **Function** and **Lib** keywords. In some rare cases, you can use the **DllImportAttribute** with the **Shared Function** keywords. These cases are explained later in this section.

   [C#] Use the **DllImportAttribute** to identify the DLL and function. Mark the method with the **static** and **extern** modifiers.

   [C++] Use the **DllImportAttribute** to identify the DLL and function. Mark the wrapper method or function with **extern "C"**.

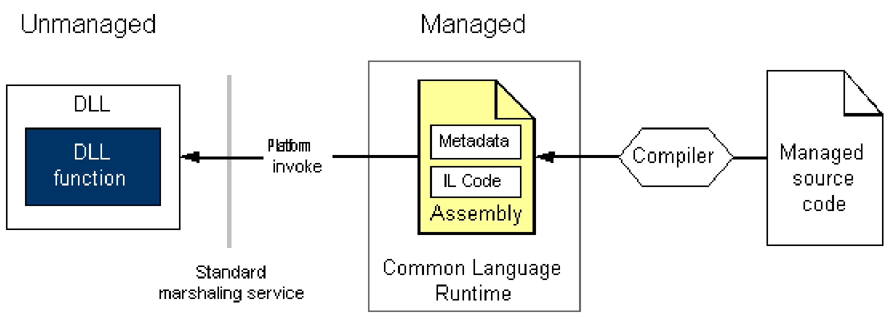
4. Call a DLL function.

   Call the method on your managed class as you would any other managed method. Passing structures and implementing callback functions are special cases.

For examples that demonstrate how to construct .NET-based declarations to be used with platform invoke, see Marshaling Data with Platform Invoke.

A closer look at platform invoke

Platform invoke relies on metadata to locate exported functions and marshal their arguments at run time. The following illustration shows this process.



When platform invoke calls an unmanaged function, it performs the following sequence of actions:

1. Locates the DLL containing the function.
2. Loads the DLL into memory.
3. Locates the address of the function in memory and pushes its arguments onto the stack, marshaling data as required.

   NOTE

   Locating and loading the DLL, and locating the address of the function in memory occur only on the first call to the function.

4. Transfers control to the unmanaged function.

Platform invoke throws exceptions generated by the unmanaged function to the managed caller.

See also

- Interoperating with Unmanaged Code
- Platform Invoke Examples
- Interop Marshaling

Identifying Functions in DLLs

4/28/2019 • 2 minutes to read • Edit Online

The identity of a DLL function consists of the following elements:

- Function name or ordinal
- Name of the DLL file in which the implementation can be found

For example, specifying the **MessageBox** function in the User32.dll identifies the function (**MessageBox**) and its location (User32.dll, User32, or user32). The Microsoft Windows application programming interface (Windows API) can contain two versions of each function that handles characters and strings: a 1-byte character ANSI version and a 2-byte character Unicode version. When unspecified, the character set, represented by the CharSet field, defaults to ANSI. Some functions can have more than two versions.

MessageBoxA is the ANSI entry point for the **MessageBox** function; **MessageBoxW** is the Unicode version. You can list function names for a specific DLL, such as user32.dll, by running a variety of command-line tools. For example, you can use `dumpbin /exports user32.dll` or `link /dump /exports user32.dll` to obtain function names.

You can rename an unmanaged function to whatever you like within your code as long as you map the new name to the original entry point in the DLL. For instructions on renaming an unmanaged DLL function in managed source code, see the Specifying an Entry Point.

Platform invoke enables you to control a significant portion of the operating system by calling functions in the Windows API and other DLLs. In addition to the Windows API, there are numerous other APIs and DLLs available to you through platform invoke.

The following table describes several commonly used DLLs in the Windows API.

DLL	DESCRIPTION OF CONTENTS
GDI32.dll	Graphics Device Interface (GDI) functions for device output, such as those for drawing and font management.
Kernel32.dll	Low-level operating system functions for memory management and resource handling.
User32.dll	Windows management functions for message handling, timers, menus, and communications.

For complete documentation on the Windows API, see the Platform SDK. For examples that demonstrate how to construct .NET-based declarations to be used with platform invoke, see Marshaling Data with Platform Invoke.

See also

- Consuming Unmanaged DLL Functions
- Specifying an Entry Point
- Creating a Class to Hold DLL Functions
- Creating Prototypes in Managed Code
- Calling a DLL Function

Creating a Class to Hold DLL Functions

4/28/2019 • 2 minutes to read • Edit Online

Wrapping a frequently used DLL function in a managed class is an effective approach to encapsulate platform functionality. Although it is not mandatory to do so in every case, providing a class wrapper is convenient because defining DLL functions can be cumbersome and error-prone. If you are programming in Visual Basic or C#, you must declare DLL functions within a class or Visual Basic module.

Within a class, you define a static method for each DLL function you want to call. The definition can include additional information, such as the character set or the calling convention used in passing method arguments; by omitting this information, you select the default settings. For a complete list of declaration options and their default settings, see Creating Prototypes in Managed Code.

Once wrapped, you can call the methods on the class as you call static methods on any other class. Platform invoke handles the underlying exported function automatically.

When designing a managed class for platform invoke, consider the relationships between classes and DLL functions. For example, you can:

- Declare DLL functions within an existing class.
- Create an individual class for each DLL function, keeping functions isolated and easy to find.
- Create one class for a set of related DLL functions to form logical groupings and reduce overhead.

You can name the class and its methods as you please. For examples that demonstrate how to construct .NET-based declarations to be used with platform invoke, see Marshaling Data with Platform Invoke.

See also

- Consuming Unmanaged DLL Functions
- Identifying Functions in DLLs
- Creating Prototypes in Managed Code
- Calling a DLL Function

Creating Prototypes in Managed Code

5/6/2019 • 5 minutes to read • Edit Online

This topic describes how to access unmanaged functions and introduces several attribute fields that annotate method definition in managed code. For examples that demonstrate how to construct .NET-based declarations to be used with platform invoke, see Marshaling Data with Platform Invoke.

Before you can access an unmanaged DLL function from managed code, you need to know the name of the function and the name of the DLL that exports it. With this information, you can begin to write the managed definition for an unmanaged function that is implemented in a DLL. Furthermore, you can adjust the way that platform invoke creates the function and marshals data to and from the function.

NOTE

Windows API functions that allocate a string enable you to free the string by using a method such as `LocalFree`. Platform invoke handles such parameters differently. For platform invoke calls, make the parameter an `IntPtr` type instead of a `String` type. Use methods that are provided by the System.Runtime.InteropServices.Marshal class to convert the type to a string manually and free it manually.

Declaration Basics

Managed definitions to unmanaged functions are language-dependent, as you can see in the following examples. For more complete code examples, see Platform Invoke Examples.

```
Friend Class NativeMethods
    Friend Declare Auto Function MessageBox Lib "user32.dll" (
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer
End Class
```

To apply the BestFitMapping, CallingConvention, ExactSpelling, PreserveSig, SetLastError, or ThrowOnUnmappableChar fields to a Visual Basic declaration, you must use the DllImportAttribute attribute instead of the `Declare` statement.

```
Imports System.Runtime.InteropServices

Friend Class NativeMethods
    <DllImport("user32.dll", CharSet:=CharSet.Auto)>
    Friend Shared Function MessageBox(
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer
    End Function
End Class
```

```
using System;
using System.Runtime.InteropServices;

internal static class NativeMethods
{
    [DllImport("user32.dll")]
    internal static extern int MessageBox(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);
}
```

```
using namespace System;
using namespace System::Runtime::InteropServices;

[DllImport("user32.dll")]
extern "C" int MessageBox(
    IntPtr hWnd, String* lpText, String* lpCaption, unsigned int uType);
```

Adjusting the Definition

Whether you set them explicitly or not, attribute fields are at work defining the behavior of managed code. Platform invoke operates according to the default values set on various fields that exist as metadata in an assembly. You can alter this default behavior by adjusting the values of one or more fields. In many cases, you use the DllImportAttribute to set a value.

The following table lists the complete set of attribute fields that pertain to platform invoke. For each field, the table includes the default value and a link to information on how to use these fields to define unmanaged DLL functions.

FIELD	DESCRIPTION
BestFitMapping	Enables or disables best-fit mapping.
CallingConvention	Specifies the calling convention to use in passing method arguments. The default is `WinAPI`, which corresponds to `__stdcall` for the 32-bit Intel-based platforms.
CharSet	Controls name mangling and the way that string arguments should be marshaled to the function. The default is `CharSet.Ansi`.
EntryPoint	Specifies the DLL entry point to be called.
ExactSpelling	Controls whether an entry point should be modified to correspond to the character set. The default value varies by programming language.
PreserveSig	Controls whether the managed method signature should be transformed into an unmanaged signature that returns an HRESULT and has an additional [out, retval] argument for the return value. The default is `true` (the signature should not be transformed).

FIELD	DESCRIPTION
SetLastError	Enables the caller to use the `Marshal.GetLastWin32Error` API function to determine whether an error occurred while executing the method. In Visual Basic, the default is `true`; in C# and C++, the default is `false`.
ThrowOnUnmappableChar	Controls throwing of an exception on an unmappable Unicode character that is converted to an ANSI "?" character.

For detailed reference information, see DllImportAttribute.

Platform invoke security considerations

The `Assert`, `Deny`, and `PermitOnly` members of the SecurityAction enumeration are referred to as *stack walk modifiers*. These members are ignored if they are used as declarative attributes on platform invoke declarations and COM Interface Definition Language (IDL) statements.

Platform Invoke Examples

The platform invoke samples in this section illustrate the use of the `RegistryPermission` attribute with the stack walk modifiers.

In the following example, the SecurityAction `Assert`, `Deny`, and `PermitOnly` modifiers are ignored.

```
[DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
[RegistryPermission(SecurityAction.Assert, Unrestricted = true)]
    private static extern bool CallRegistryPermissionAssert();

[DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
[RegistryPermission(SecurityAction.Deny, Unrestricted = true)]
    private static extern bool CallRegistryPermissionDeny();

[DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
[RegistryPermission(SecurityAction.PermitOnly, Unrestricted = true)]
    private static extern bool CallRegistryPermissionDeny();
```

However, the `Demand` modifier in the following example is accepted.

```
[DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
[RegistryPermission(SecurityAction.Demand, Unrestricted = true)]
    private static extern bool CallRegistryPermissionDeny();
```

SecurityAction modifiers do work correctly if they are placed on a class that contains (wraps) the platform invoke call.

```
      [RegistryPermission(SecurityAction.Demand, Unrestricted = true)]
public ref class PInvokeWrapper
{
public:
[DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
    private static extern bool CallRegistryPermissionDeny();
};
```

```
[RegistryPermission(SecurityAction.Demand, Unrestricted = true)]
class PInvokeWrapper
{
[DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
    private static extern bool CallRegistryPermissionDeny();
}
```

SecurityAction modifiers also work correctly in a nested scenario where they are placed on the caller of the platform invoke call:

```
      {
public ref class PInvokeWrapper
public:
    [DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
    private static extern bool CallRegistryPermissionDeny();

    [RegistryPermission(SecurityAction.Demand, Unrestricted = true)]
    public static bool CallRegistryPermission()
    {
     return CallRegistryPermissionInternal();
    }
};
```

```
class PInvokeScenario
{
    [DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
    private static extern bool CallRegistryPermissionInternal();

    [RegistryPermission(SecurityAction.Assert, Unrestricted = true)]
    public static bool CallRegistryPermission()
    {
     return CallRegistryPermissionInternal();
    }
}
```

COM Interop Examples

The COM interop samples in this section illustrate the use of the `RegistryPermission` attribute with the stack walk modifiers.

The following COM interop interface declarations ignore the `Assert`, `Deny`, and `PermitOnly` modifiers, similarly to the platform invoke examples in the previous section.

```
[ComImport, Guid("12345678-43E6-43c9-9A13-47F40B338DE0")]
interface IAssertStubsItf
{
[RegistryPermission(SecurityAction.Assert, Unrestricted = true)]
    bool CallRegistryPermission();
[FileIOPermission(SecurityAction.Assert, Unrestricted = true)]
    bool CallFileIoPermission();
}

[ComImport, Guid("12345678-43E6-43c9-9A13-47F40B338DE0")]
interface IDenyStubsItf
{
[RegistryPermission(SecurityAction.Deny, Unrestricted = true)]
    bool CallRegistryPermission();
[FileIOPermission(SecurityAction.Deny, Unrestricted = true)]
    bool CallFileIoPermission();
}

[ComImport, Guid("12345678-43E6-43c9-9A13-47F40B338DE0")]
interface IAssertStubsItf
{
[RegistryPermission(SecurityAction.PermitOnly, Unrestricted = true)]
    bool CallRegistryPermission();
[FileIOPermission(SecurityAction.PermitOnly, Unrestricted = true)]
    bool CallFileIoPermission();
}
```

Additionally, the `Demand` modifier is not accepted in COM interop interface declaration scenarios, as shown in the following example.

```
[ComImport, Guid("12345678-43E6-43c9-9A13-47F40B338DE0")]
interface IDemandStubsItf
{
[RegistryPermission(SecurityAction.Demand, Unrestricted = true)]
    bool CallRegistryPermission();
[FileIOPermission(SecurityAction.Demand, Unrestricted = true)]
    bool CallFileIoPermission();
}
```

See also

- Consuming Unmanaged DLL Functions
- Specifying an Entry Point
- Specifying a Character Set
- Platform Invoke Examples
- Platform Invoke Security Considerations
- Identifying Functions in DLLs
- Creating a Class to Hold DLL Functions
- Calling a DLL Function

Specifying an Entry Point

5/6/2019 • 2 minutes to read • Edit Online

An entry point identifies the location of a function in a DLL. Within a managed project, the original name or ordinal entry point of a target function identifies that function across the interoperation boundary. Further, you can map the entry point to a different name, effectively renaming the function.

Following is a list of possible reasons to rename a DLL function:

- To avoid using case-sensitive API function names
- To comply with existing naming standards
- To accommodate functions that take different data types (by declaring multiple versions of the same DLL function)
- To simplify using APIs that contain ANSI and Unicode versions

This topic demonstrates how to rename a DLL function in managed code.

Renaming a Function in Visual Basic

Visual Basic uses the **Function** keyword in the **Declare** statement to set the DllImportAttribute.EntryPoint field. The following example shows a basic declaration.

```
Friend Class NativeMethods
    Friend Declare Auto Function MessageBox Lib "user32.dll" (
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer
End Class
```

You can replace the **MessageBox** entry point with **MsgBox** by including the **Alias** keyword in your definition, as shown in the following example. In both examples the **Auto** keyword eliminates the need to specify the character-set version of the entry point. For more information about selecting a character set, see Specifying a Character Set.

```
Friend Class NativeMethods
    Friend Declare Auto Function MsgBox _
        Lib "user32.dll" Alias "MessageBox" (
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer
End Class
```

Renaming a Function in C# and C++

You can use the DllImportAttribute.EntryPoint field to specify a DLL function by name or ordinal. If the name of the function in your method definition is the same as the entry point in the DLL, you do not have to explicitly identify the function with the **EntryPoint** field. Otherwise, use one of the following attribute forms to indicate a name or ordinal:

```
[DllImport("DllName", EntryPoint = "Functionname")]
[DllImport("DllName", EntryPoint = "#123")]
```

Notice that you must prefix an ordinal with the pound sign (#).

The following example demonstrates how to replace **MessageBoxA** with **MsgBox** in your code by using the **EntryPoint** field.

```
using System;
using System.Runtime.InteropServices;

internal static class NativeMethods
{
    [DllImport("user32.dll", EntryPoint = "MessageBoxA")]
    internal static extern int MessageBox(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);
}
```

```
using namespace System;
using namespace System::Runtime::InteropServices;

typedef void* HWND;
[DllImport("user32", EntryPoint = "MessageBoxA")]
extern "C" int MsgBox(
    HWND hWnd, String* lpText, String* lpCaption, unsigned int uType);
```

See also

- DllImportAttribute
- Creating Prototypes in Managed Code
- Platform Invoke Examples
- Marshaling Data with Platform Invoke

Specifying a Character Set

5/9/2019 • 3 minutes to read • Edit Online

The DllImportAttribute.CharSet field controls string marshaling and determines how platform invoke finds function names in a DLL. This topic describes both behaviors.

Some APIs export two versions of functions that take string arguments: narrow (ANSI) and wide (Unicode). The Windows API, for instance, includes the following entry-point names for the **MessageBox** function:

- **MessageBoxA**

  Provides 1-byte character ANSI formatting, distinguished by an "A" appended to the entry-point name. Calls to **MessageBoxA** always marshal strings in ANSI format.

- **MessageBoxW**

  Provides 2-byte character Unicode formatting, distinguished by a "W" appended to the entry-point name. Calls to **MessageBoxW** always marshal strings in Unicode format.

String Marshaling and Name Matching

The `CharSet` field accepts the following values:

Ansi (default value)

- String marshaling

  Platform invoke marshals strings from their managed format (Unicode) to ANSI format.

- Name matching

  When the DllImportAttribute.ExactSpelling field is `true`, as it is by default in Visual Basic, platform invoke searches only for the name you specify. For example, if you specify **MessageBox**, platform invoke searches for **MessageBox** and fails when it cannot locate the exact spelling.

  When the `ExactSpelling` field is `false`, as it is by default in C++ and C#, platform invoke searches for the unmangled alias first (**MessageBox**), then the mangled name (**MessageBoxA**) if the unmangled alias is not found. Notice that ANSI name-matching behavior differs from Unicode name-matching behavior.

Unicode

- String marshaling

  Platform invoke copies strings from their managed format (Unicode) to Unicode format.

- Name matching

  When the `ExactSpelling` field is `true`, as it is by default in Visual Basic, platform invoke searches only for the name you specify. For example, if you specify **MessageBox**, platform invoke searches for **MessageBox** and fails if it cannot locate the exact spelling.

  When the `ExactSpelling` field is `false`, as it is by default in C++ and C#, platform invoke searches for the mangled name first (**MessageBoxW**), then the unmangled alias (**MessageBox**) if the mangled name is not found. Notice that Unicode name-matching behavior differs from ANSI name-matching behavior.

Auto

- Platform invoke chooses between ANSI and Unicode formats at run time, based on the target platform.

Specifying a Character Set in Visual Basic

The following example declares the **MessageBox** function three times, each time with different character-set behavior. You can specify character-set behavior in Visual Basic by adding the **Ansi**, **Unicode**, or **Auto** keyword to the declaration statement.

If you omit the character-set keyword, as is done in the first declaration statement, the DllImportAttribute.CharSet field defaults to the ANSI character set. The second and third statements in the example explicitly specify a character set with a keyword.

```
Friend Class NativeMethods
    Friend Declare Function MessageBoxA Lib "user32.dll" (
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer

    Friend Declare Unicode Function MessageBoxW Lib "user32.dll" (
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer

    Friend Declare Auto Function MessageBox Lib "user32.dll" (
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer
End Class
```

Specifying a Character Set in C# and C++

The DllImportAttribute.CharSet field identifies the underlying character set as ANSI or Unicode. The character set controls how string arguments to a method should be marshaled. Use one of the following forms to indicate the character set:

```
[DllImport("DllName", CharSet = CharSet.Ansi)]
[DllImport("DllName", CharSet = CharSet.Unicode)]
[DllImport("DllName", CharSet = CharSet.Auto)]
```

```
[DllImport("DllName", CharSet = CharSet::Ansi)]
[DllImport("DllName", CharSet = CharSet::Unicode)]
[DllImport("DllName", CharSet = CharSet::Auto)]
```

The following example shows three managed definitions of the **MessageBox** function attributed to specify a character set. In the first definition, by its omission, the `CharSet` field defaults to the ANSI character set.

```
using System;
using System.Runtime.InteropServices;

internal static class NativeMethods
{
    [DllImport("user32.dll")]
    internal static extern int MessageBoxA(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);

    [DllImport("user32.dll", CharSet = CharSet.Unicode)]
    internal static extern int MessageBoxW(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);

    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    internal static extern int MessageBox(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);
}
```

```
typedef void* HWND;

// Can use MessageBox or MessageBoxA.
[DllImport("user32")]
extern "C" int MessageBox(
    HWND hWnd, String* lpText, String* lpCaption, unsigned int uType);

// Can use MessageBox or MessageBoxW.
[DllImport("user32", CharSet = CharSet::Unicode)]
extern "C" int MessageBoxW(
    HWND hWnd, String* lpText, String* lpCaption, unsigned int uType);

// Must use MessageBox.
[DllImport("user32", CharSet = CharSet::Auto)]
extern "C" int MessageBox(
    HWND hWnd, String* lpText, String* lpCaption, unsigned int uType);
```

See also

- DllImportAttribute
- Creating Prototypes in Managed Code
- Platform Invoke Examples
- Marshaling Data with Platform Invoke

Platform Invoke Examples

4/8/2019 • 2 minutes to read • Edit Online

The following examples demonstrate how to define and call the **MessageBox** function in User32.dll, passing a simple string as an argument. In the examples, the DllImportAttribute.CharSet field is set to **Auto** to let the target platform determine the character width and string marshaling.

```
using namespace System::Runtime::InteropServices;

typedef void* HWND;

[DllImport("user32", CharSet=CharSet::Auto)]
extern "C" IntPtr MessageBox(HWND hWnd,
                             String* pText,
                             String* pCaption,
                             unsigned int uType);

void main()
{
     String* pText = L"Hello World!";
     String* pCaption = L"Platform Invoke Sample";
     MessageBox(0, pText, pCaption, 0);
}
```

```
using System;
using System.Runtime.InteropServices;

public class Win32 {
     [DllImport("user32.dll", CharSet=CharSet.Auto)]
     public static extern IntPtr MessageBox(int hWnd, String text,
                     String caption, uint type);
}

public class HelloWorld {
    public static void Main() {
       Win32.MessageBox(0, "Hello World", "Platform Invoke Sample", 0);
    }
}
```

```
Imports System.Runtime.InteropServices

Public Class Win32
    Declare Auto Function MessageBox Lib "user32.dll" _
       (ByVal hWnd As Integer, ByVal txt As String, _
       ByVal caption As String, ByVal Typ As Integer) As IntPtr
End Class

Public Class HelloWorld
    Public Shared Sub Main()
        Win32.MessageBox(0, "Hello World", "Platform Invoke Sample", 0)
    End Sub
End Class
```

For additional examples, see Marshaling Data with Platform Invoke.

See also

- DllImportAttribute
- Creating Prototypes in Managed Code
- Specifying a Character Set

Calling a DLL Function

5/4/2018 • 2 minutes to read • Edit Online

Although calling unmanaged DLL functions is nearly identical to calling other managed code, there are differences that can make DLL functions seem confusing at first. This section introduces topics that describe some of the unusual calling-related issues.

Structures that are returned from platform invoke calls must be data types that have the same representation in managed and unmanaged code. Such types are called *blittable types* because they do not require conversion (see Blittable and Non-Blittable Types). To call a function that has a non-blittable structure as its return type, you can define a blittable helper type of the same size as the non-blittable type and convert the data after the function returns.

In This Section

Passing Structures
Identifies the issues of passing data structures with a predefined layout.

Callback Functions
Provides basic information about callback functions.

How to: Implement Callback Functions
Describes how to implement callback functions in managed code.

Related Sections

Consuming Unmanaged DLL Functions
Describes how to call unmanaged DLL functions using platform invoke.

Marshaling Data with Platform Invoke
Describes how to declare method parameters and pass arguments to functions exported by unmanaged libraries.

Passing Structures

5/6/2019 • 4 minutes to read • Edit Online

Many unmanaged functions expect you to pass, as a parameter to the function, members of structures (user-defined types in Visual Basic) or members of classes that are defined in managed code. When passing structures or classes to unmanaged code using platform invoke, you must provide additional information to preserve the original layout and alignment. This topic introduces the StructLayoutAttribute attribute, which you use to define formatted types. For managed structures and classes, you can select from several predictable layout behaviors supplied by the **LayoutKind** enumeration.

Central to the concepts presented in this topic is an important difference between structure and class types. Structures are value types and classes are reference types — classes always provide at least one level of memory indirection (a pointer to a value). This difference is important because unmanaged functions often demand indirection, as shown by the signatures in the first column of the following table. The managed structure and class declarations in the remaining columns show the degree to which you can adjust the level of indirection in your declaration.Declarations are provided for both Visual Basic and Visual C#.

UNMANAGED SIGNATURE	MANAGED DECLARATION: NO INDIRECTION `STRUCTURE MYTYPE` `STRUCT MYTYPE;`	MANAGED DECLARATION: ONE LEVEL OF INDIRECTION `CLASS MYTYPE` `CLASS MYTYPE;`
`DoWork(MyType x);` Demands zero levels of indirection.	`DoWork(ByVal x As MyType)` `DoWork(MyType x)` Adds zero levels of indirection.	Not possible because there is already one level of indirection.
`DoWork(MyType* x);` Demands one level of indirection.	`DoWork(ByRef x As MyType)` `DoWork(ref MyType x)` Adds one level of indirection.	`DoWork(ByVal x As MyType)` `DoWork(MyType x)` Adds zero levels of indirection.
`DoWork(MyType** x);` Demands two levels of indirection.	Not possible because **ByRef ByRef** or `ref` `ref` cannot be used.	`DoWork(ByRef x As MyType)` `DoWork(ref MyType x)` Adds one level of indirection.

The table describes the following guidelines for platform invoke declarations:

- Use a structure passed by value when the unmanaged function demands no indirection.
- Use either a structure passed by reference or a class passed by value when the unmanaged function demands one level of indirection.
- Use a class passed by reference when the unmanaged function demands two levels of indirection.

Declaring and Passing Structures

The following example shows how to define the `Point` and `Rect` structures in managed code, and pass the types as parameter to the **PtInRect** function in the User32.dll file. **PtInRect** has the following unmanaged signature:

```
BOOL PtInRect(const RECT *lprc, POINT pt);
```

Notice that you must pass the Rect structure by reference, since the function expects a pointer to a RECT type.

```
Imports System.Runtime.InteropServices

<StructLayout(LayoutKind.Sequential)> Public Structure Point
    Public x As Integer
    Public y As Integer
End Structure

Public Structure <StructLayout(LayoutKind.Explicit)> Rect
    <FieldOffset(0)> Public left As Integer
    <FieldOffset(4)> Public top As Integer
    <FieldOffset(8)> Public right As Integer
    <FieldOffset(12)> Public bottom As Integer
End Structure

Friend Class NativeMethods
    Friend Declare Auto Function PtInRect Lib "user32.dll" (
        ByRef r As Rect, p As Point) As Boolean
End Class
```

```
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public struct Point {
    public int x;
    public int y;
}

[StructLayout(LayoutKind.Explicit)]
public struct Rect {
    [FieldOffset(0)] public int left;
    [FieldOffset(4)] public int top;
    [FieldOffset(8)] public int right;
    [FieldOffset(12)] public int bottom;
}

internal static class NativeMethods
{
    [DllImport("User32.dll")]
    internal static extern bool PtInRect(ref Rect r, Point p);
}
```

Declaring and Passing Classes

You can pass members of a class to an unmanaged DLL function, as long as the class has a fixed member layout. The following example demonstrates how to pass members of the `MySystemTime` class, which are defined in sequential order, to the **GetSystemTime** in the User32.dll file. **GetSystemTime** has the following unmanaged signature:

```
void GetSystemTime(SYSTEMTIME* SystemTime);
```

Unlike value types, classes always have at least one level of indirection.

```
Imports System.Runtime.InteropServices

<StructLayout(LayoutKind.Sequential)> Public Class MySystemTime
    Public wYear As Short
    Public wMonth As Short
    Public wDayOfWeek As Short
    Public wDay As Short
    Public wHour As Short
    Public wMinute As Short
    Public wSecond As Short
    Public wMiliseconds As Short
End Class

Friend Class NativeMethods
    Friend Declare Auto Sub GetSystemTime Lib "Kernel32.dll" (
        sysTime As MySystemTime)
    Friend Declare Auto Function MessageBox Lib "User32.dll" (
        hWnd As IntPtr, lpText As String, lpCaption As String, uType As UInteger) As Integer
End Class

Public Class TestPlatformInvoke
    Public Shared Sub Main()
        Dim sysTime As New MySystemTime()
        NativeMethods.GetSystemTime(sysTime)

        Dim dt As String
        dt = "System time is:" & ControlChars.CrLf & _
              "Year: " & sysTime.wYear & _
              ControlChars.CrLf & "Month: " & sysTime.wMonth & _
              ControlChars.CrLf & "DayOfWeek: " & sysTime.wDayOfWeek & _
              ControlChars.CrLf & "Day: " & sysTime.wDay
        NativeMethods.MessageBox(IntPtr.Zero, dt, "Platform Invoke Sample", 0)
    End Sub
End Class
```

```
[StructLayout(LayoutKind.Sequential)]
public class MySystemTime {
    public ushort wYear;
    public ushort wMonth;
    public ushort wDayOfWeek;
    public ushort wDay;
    public ushort wHour;
    public ushort wMinute;
    public ushort wSecond;
    public ushort wMilliseconds;
}
internal static class NativeMethods
{
    [DllImport("Kernel32.dll")]
    internal static extern void GetSystemTime(MySystemTime st);

    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    internal static extern int MessageBox(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);
}

public class TestPlatformInvoke
{
    public static void Main()
    {
        MySystemTime sysTime = new MySystemTime();
        NativeMethods.GetSystemTime(sysTime);

        string dt;
        dt = "System time is: \n" +
              "Year: " + sysTime.wYear + "\n" +
              "Month: " + sysTime.wMonth + "\n" +
              "DayOfWeek: " + sysTime.wDayOfWeek + "\n" +
              "Day: " + sysTime.wDay;
        NativeMethods.MessageBox(IntPtr.Zero, dt, "Platform Invoke Sample", 0);
    }
}
```

See also

- Calling a DLL Function
- StructLayoutAttribute
- FieldOffsetAttribute

Callback Functions

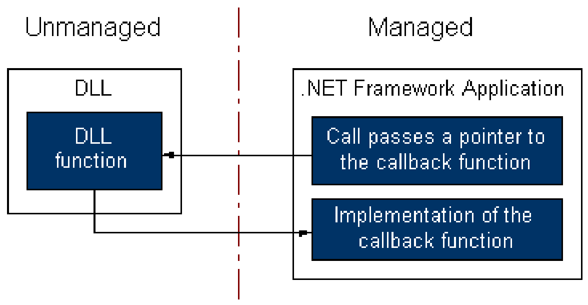
4/8/2019 • 2 minutes to read • Edit Online

A callback function is code within a managed application that helps an unmanaged DLL function complete a task. Calls to a callback function pass indirectly from a managed application, through a DLL function, and back to the managed implementation. Some of the many DLL functions called with platform invoke require a callback function in managed code to run properly.

To call most DLL functions from managed code, you create a managed definition of the function and then call it. The process is straightforward.

Using a DLL function that requires a callback function has some additional steps. First, you must determine whether the function requires a callback by looking at the documentation for the function. Next, you have to create the callback function in your managed application. Finally, you call the DLL function, passing a pointer to the callback function as an argument.

The following illustration summarizes the callback function and implementation steps:


Unmanaged
Managed
DLL
DLL function
.NET Framework Application
Call passes a pointer to the callback function
Implementation of the callback function

Callback functions are ideal for use in situations in which a task is performed repeatedly. Another common usage is with enumeration functions, such as **EnumFontFamilies**, **EnumPrinters**, and **EnumWindows** in the Windows API. The **EnumWindows** function enumerates through all existing windows on your computer, calling the callback function to perform a task on each window. For instructions and an example, see How to: Implement Callback Functions.

See also

- How to: Implement Callback Functions
- Calling a DLL Function

How to: Implement Callback Functions

4/9/2019 • 2 minutes to read • Edit Online

The following procedure and example demonstrate how a managed application, using platform invoke, can print the handle value for each window on the local computer. Specifically, the procedure and example use the **EnumWindows** function to step through the list of windows and a managed callback function (named CallBack) to print the value of the window handle.

To implement a callback function

1. Look at the signature for the **EnumWindows** function before going further with the implementation. **EnumWindows** has the following signature:

   ```
   BOOL EnumWindows(WNDENUMPROC lpEnumFunc, LPARAM lParam)
   ```

   One clue that this function requires a callback is the presence of the **lpEnumFunc** argument. It is common to see the **lp** (long pointer) prefix combined with the **Func** suffix in the name of arguments that take a pointer to a callback function. For documentation about Win32 functions, see the Microsoft Platform SDK.

2. Create the managed callback function. The example declares a delegate type, called `CallBack`, which takes two arguments (**hwnd** and **lparam**). The first argument is a handle to the window; the second argument is application-defined. In this release, both arguments must be integers.

   Callback functions generally return nonzero values to indicate success and zero to indicate failure. This example explicitly sets the return value to **true** to continue the enumeration.

3. Create a delegate and pass it as an argument to the **EnumWindows** function. Platform invoke converts the delegate to a familiar callback format automatically.

4. Ensure that the garbage collector does not reclaim the delegate before the callback function completes its work. When you pass a delegate as a parameter, or pass a delegate contained as a field in a structure, it remains uncollected for the duration of the call. So, as is the case in the following enumeration example, the callback function completes its work before the call returns and requires no additional action by the managed caller.

   If, however, the callback function can be invoked after the call returns, the managed caller must take steps to ensure that the delegate remains uncollected until the callback function finishes. For detailed information about preventing garbage collection, see Interop Marshaling with Platform Invoke.

Example

```vb
Imports System
Imports System.Runtime.InteropServices

Public Delegate Function CallBack( _
hwnd As Integer, lParam As Integer) As Boolean

Public Class EnumReportApp

    Declare Function EnumWindows Lib "user32" ( _
       x As CallBack, y As Integer) As Integer

    Public Shared Sub Main()
        EnumWindows(AddressOf EnumReportApp.Report, 0)
    End Sub 'Main

    Public Shared Function Report(hwnd As Integer, lParam As Integer) _
    As Boolean
        Console.Write("Window handle is ")
        Console.WriteLine(hwnd)
        Return True
    End Function 'Report
End Class 'EnumReportApp
```

```csharp
using System;
using System.Runtime.InteropServices;

public delegate bool CallBack(int hwnd, int lParam);

public class EnumReportApp
{
    [DllImport("user32")]
    public static extern int EnumWindows(CallBack x, int y);

    public static void Main()
    {
        CallBack myCallBack = new CallBack(EnumReportApp.Report);
        EnumWindows(myCallBack, 0);
    }

    public static bool Report(int hwnd, int lParam)
    {
        Console.Write("Window handle is ");
        Console.WriteLine(hwnd);
        return true;
    }
}
```

```
using namespace System;
using namespace System::Runtime::InteropServices;

// A delegate type.
delegate bool CallBack(int hwnd, int lParam);

// Managed type with the method to call.
ref class EnumReport
{
// Report the window handle.
public:
    [DllImport("user32")]
    static int EnumWindows(CallBack^ x, int y);

    static void Main()
    {
        EnumReport^ er = gcnew EnumReport;
        CallBack^ myCallBack = gcnew CallBack(&EnumReport::Report);
        EnumWindows(myCallBack, 0);
    }

    static bool Report(int hwnd, int lParam)
    {
       Console::Write(L"Window handle is ");
       Console::WriteLine(hwnd);
       return true;
    }
};

int main()
{
    EnumReport::Main();
}
```

See also

- Callback Functions
- Calling a DLL Function

Interop Marshaling

5/9/2019 • 6 minutes to read • Edit Online

Interop marshaling governs how data is passed in method arguments and return values between managed and unmanaged memory during calls. Interop marshaling is a run-time activity performed by the common language runtime's marshaling service.

Most data types have common representations in both managed and unmanaged memory. The interop marshaler handles these types for you. Other types can be ambiguous or not represented at all in managed memory.

An ambiguous type can have either multiple unmanaged representations that map to a single managed type, or missing type information, such as the size of an array. For ambiguous types, the marshaler provides a default representation and alternative representations where multiple representations exist. You can supply explicit instructions to the marshaler on how it is to marshal an ambiguous type.

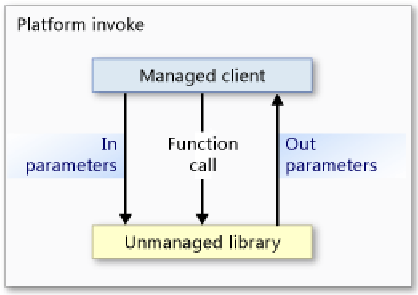
This overview contains the following sections:

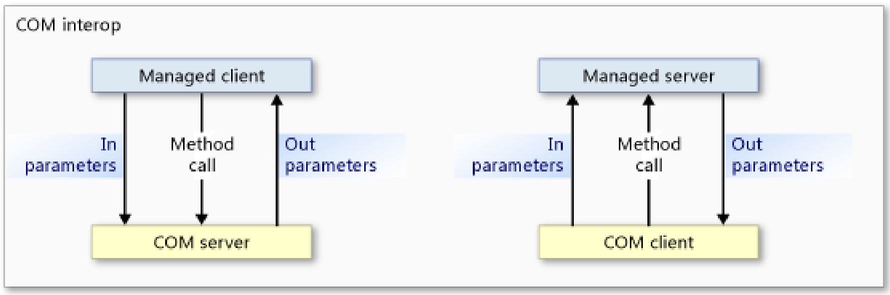
Platform Invoke and COM Interop Models

The common language runtime provides two mechanisms for interoperating with unmanaged code:

- Platform invoke, which enables managed code to call functions exported from an unmanaged library.
- COM interop, which enables managed code to interact with Component Object Model (COM) objects through interfaces.

Both platform invoke and COM interop use interop marshaling to accurately move method arguments between caller and callee and back, if required. As the following illustration shows, a platform invoke method call flows from managed to unmanaged code and never the other way, except when callback functions are involved. Even though platform invoke calls can flow only from managed to unmanaged code, data can flow in both directions as input or output parameters. COM interop method calls can flow in either direction.





At the lowest level, both mechanisms use the same interop marshaling service; however, certain data types are supported exclusively by COM interop or platform invoke. For details, see Default Marshaling Behavior.

Back to top

Marshaling and COM Apartments

The interop marshaler marshals data between the common language runtime heap and the unmanaged heap. Marshaling occurs whenever the caller and callee cannot operate on the same instance of data. The interop marshaler makes it possible for the caller and callee to appear to be operating on the same data even if they have their own copy of the data.

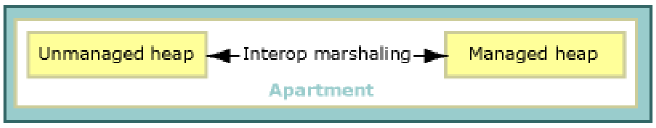
COM also has a marshaler that marshals data between COM apartments or different COM processes. When calling between managed and unmanaged code within the same COM apartment, the interop marshaler is the only marshaler involved. When calling between managed code and unmanaged code in a different COM apartment or a different process, both the interop marshaler and the COM marshaler are involved.

COM Clients and Managed Servers

An exported managed server with a type library registered by the Regasm.exe (Assembly Registration Tool) has a `ThreadingModel` registry entry set to `Both`. This value indicates that the server can be activated in a single-threaded apartment (STA) or a multithreaded apartment (MTA). The server object is created in the same apartment as its caller, as shown in the following table.

COM CLIENT	.NET SERVER	MARSHALING REQUIREMENTS
STA	`Both` becomes STA.	Same-apartment marshaling.
MTA	`Both` becomes MTA.	Same-apartment marshaling.

Because the client and server are in the same apartment, the interop marshaling service automatically handles all data marshaling. The following illustration shows the interop marshaling service operating between managed and unmanaged heaps within the same COM-style apartment.



If you plan to export a managed server, be aware that the COM client determines the apartment of the server. A managed server called by a COM client initialized in an MTA must ensure thread safety.

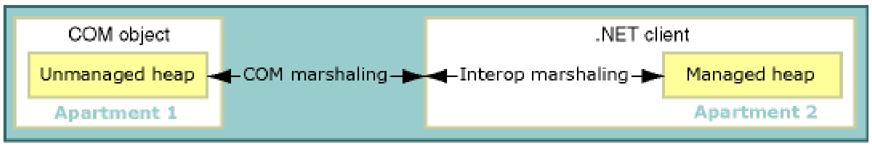
Managed Clients and COM Servers

The default setting for managed client apartments is MTA; however, the application type of the .NET client can change the default setting. For example, a Visual Basic client apartment setting is STA. You can use the System.STAThreadAttribute, the System.MTAThreadAttribute, the Thread.ApartmentState property, or the Page.AspCompatMode property to examine and change the apartment setting of a managed client.

The author of the component sets the thread affinity of a COM server. The following table shows the combinations of apartment settings for .NET clients and COM servers. It also shows the resulting marshaling requirements for the combinations.

.NET CLIENT	COM SERVER	MARSHALING REQUIREMENTS
MTA (default)	MTA	Interop marshaling.
	STA	Interop and COM marshaling.
STA	MTA	Interop and COM marshaling.
	STA	Interop marshaling.

When a managed client and unmanaged server are in the same apartment, the interop marshaling service handles all data marshaling. However, when client and server are initialized in different apartments, COM marshaling is also required. The following illustration shows the elements of a cross-apartment call.



For cross-apartment marshaling, you can do the following:

- Accept the overhead of the cross-apartment marshaling, which is noticeable only when there are many calls across the boundary. You must register the type library of the COM component for calls to successfully cross the apartment boundary.
- Alter the main thread by setting the client thread to STA or MTA. For example, if your C# client calls many STA COM components, you can avoid cross-apartment marshaling by setting the main thread to STA.

> **NOTE**
>
> Once the thread of a C# client is set to STA, calls to MTA COM components will require cross-apartment marshaling.

For instructions on explicitly selecting an apartment model, see Managed and Unmanaged Threading.

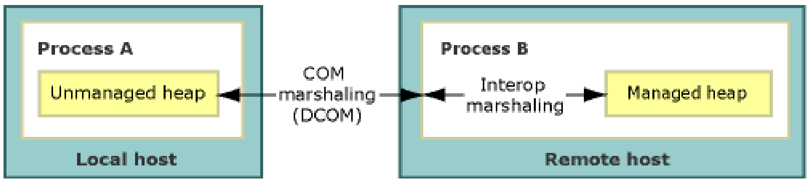
Back to top

Marshaling Remote Calls

As with cross-apartment marshaling, COM marshaling is involved in each call between managed and unmanaged code whenever the objects reside in separate processes. For example:

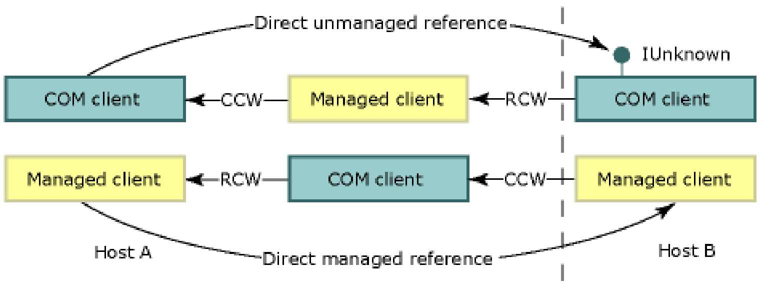
- A COM client that invokes a managed server on a remote host uses distributed COM (DCOM).
- A managed client that invokes a COM server on a remote host uses DCOM.

The following illustration shows how interop marshaling and COM marshaling provide communications channels across process and host boundaries.



Preserving Identity

The common language runtime preserves the identity of managed and unmanaged references. The following illustration shows the flow of direct unmanaged references (top row) and direct managed references (bottom row) across process and host boundaries.



In this illustration:

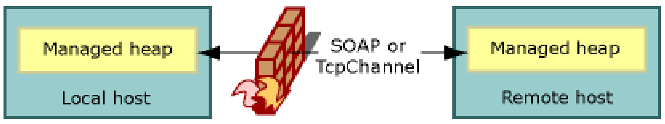
- An unmanaged client gets a reference to a COM object from a managed object that gets this reference from a remote host. The remoting mechanism is DCOM.
- A managed client gets a reference to a managed object from a COM object that gets this reference from a remote host. The remoting mechanism is DCOM.

> **NOTE**
>
> The exported type library of the managed server must be registered.

The number of process boundaries between caller and callee is irrelevant; the same direct referencing occurs for in-process and out-of-process calls.

Managed Remoting

The runtime also provides managed remoting, which you can use to establish a communications channel between managed objects across process and host boundaries. Managed remoting can accommodate a firewall between the communicating components, as the following illustration shows.



Remote calls across firewalls using SOAP or the TcpChannel class

Some unmanaged calls can be channeled through SOAP, such as the calls between serviced components and COM.

Back to top

Related Topics

TITLE	DESCRIPTION
Default Marshaling Behavior	Describes the rules that the interop marshaling service uses to marshal data.
Marshaling Data with Platform Invoke	Describes how to declare method parameters and pass arguments to functions exported by unmanaged libraries.
Marshaling Data with COM Interop	Describes how to customize COM wrappers to alter marshaling behavior.
How to: Migrate Managed-Code DCOM to WCF	Describes how to migrate from DCOM to WCF.
How to: Map HRESULTs and Exceptions	Describes how to map custom exceptions to HRESULTs and provides the complete mapping from each HRESULT to its comparable exception class in the .NET Framework.
Interoperating Using Generic Types	Describes which actions are supported when using generic types for COM interoperability.
Interoperating with Unmanaged Code	Describes interoperability services provided by the common language runtime.
Advanced COM Interoperability	Provides links to more information about incorporating COM components into your .NET Framework application.
Design Considerations for Interoperation	Provides tips for writing integrated COM components.

Back to top

Reference

System.Runtime.InteropServices

Back to top

Default Marshaling Behavior

5/6/2019 • 14 minutes to read • Edit Online

Interop marshaling operates on rules that dictate how data associated with method parameters behaves as it passes between managed and unmanaged memory. These built-in rules control such marshaling activities as data type transformations, whether a callee can change data passed to it and return those changes to the caller, and under which circumstances the marshaler provides performance optimizations.

This section identifies the default behavioral characteristics of the interop marshaling service. It presents detailed information on marshaling arrays, Boolean types, char types, delegates, classes, objects, strings, and structures.

NOTE

Marshaling of generic types is not supported. For more information see, Interoperating Using Generic Types.

Memory management with the interop marshaler

The interop marshaler always attempts to free memory allocated by unmanaged code. This behavior complies with COM memory management rules, but differs from the rules that govern native C++.

Confusion can arise if you anticipate native C++ behavior (no memory freeing) when using platform invoke, which automatically frees memory for pointers. For example, calling the following unmanaged method from a C++ DLL does not automatically free any memory.

Unmanaged signature

```
BSTR MethodOne (BSTR b) {
     return b;
}
```

However, if you define the method as a platform invoke prototype, replace each **BSTR** type with a String type, and call `MethodOne`, the common language runtime attempts to free `b` twice. You can change the marshaling behavior by using IntPtr types rather than **String** types.

The runtime always uses the **CoTaskMemFree** method to free memory. If the memory you are working with was not allocated with the **CoTaskMemAlloc** method, you must use an **IntPtr** and free the memory manually using the appropriate method. Similarly, you can avoid automatic memory freeing in situations where memory should never be freed, such as when using the **GetCommandLine** function from Kernel32.dll, which returns a pointer to kernel memory. For details on manually freeing memory, see the Buffers Sample.

Default marshaling for classes

Classes can be marshaled only by COM interop and are always marshaled as interfaces. In some cases the interface used to marshal the class is known as the class interface. For information about overriding the class interface with an interface of your choice, see Introducing the class interface.

Passing Classes to COM

When a managed class is passed to COM, the interop marshaler automatically wraps the class with a COM proxy and passes the class interface produced by the proxy to the COM method call. The proxy then delegates all calls on the class interface back to the managed object. The proxy also exposes other interfaces that are not explicitly implemented by the class. The proxy automatically implements interfaces such as **IUnknown** and **IDispatch** on behalf of the class.

Passing Classes to .NET Code

Coclasses are not typically used as method arguments in COM. Instead, a default interface is usually passed in place of the coclass.

When an interface is passed into managed code, the interop marshaler is responsible for wrapping the interface with the proper wrapper and passing the wrapper to the managed method. Determining which wrapper to use can be difficult. Every instance of a COM object has a single, unique wrapper, no matter how many interfaces the object implements. For example, a single COM object that implements five distinct interfaces has only one wrapper. The same wrapper exposes all five interfaces. If two instances of the COM object are created, then two instances of the wrapper are created.

For the wrapper to maintain the same type throughout its lifetime, the interop marshaler must identify the correct wrapper the first time an interface exposed by the object is passed through the marshaler. The marshaler identifies the object by looking at one of the interfaces the object implements.

For example, the marshaler determines that the class wrapper should be used to wrap the interface that was passed into managed code. When the interface is first passed through the marshaler, the marshaler checks whether the interface is coming from a known object. This check occurs in two situations:

- An interface is being implemented by another managed object that was passed to COM elsewhere. The marshaler can readily identify interfaces exposed by managed objects and is able to match the interface with the managed object that provides the implementation. The managed object is then passed to the method and no wrapper is needed.

- An object that has already been wrapped is implementing the interface. To determine whether this is the case, the marshaler queries the object for its **IUnknown** interface and compares the returned interface to the interfaces of other objects that are already wrapped. If the interface is the same as that of another wrapper, the objects have the same identity and the existing wrapper is passed to the method.

If an interface is not from a known object, the marshaler does the following:

1. The marshaler queries the object for the **IProvideClassInfo2** interface. If provided, the marshaler uses the CLSID returned from **IProvideClassInfo2.GetGUID** to identify the coclass providing the interface. With the CLSID, the marshaler can locate the wrapper from the registry if the assembly has previously been registered.

2. The marshaler queries the interface for the **IProvideClassInfo** interface. If provided, the marshaler uses the **ITypeInfo** returned from **IProvideClassInfo.GetClassinfo** to determine the CLSID of the class exposing the interface. The marshaler can use the CLSID to locate the metadata for the wrapper.

3. If the marshaler still cannot identify the class, it wraps the interface with a generic wrapper class called **System.__ComObject**.

Default marshaling for delegates

A managed delegate is marshaled as a COM interface or as a function pointer, based on the calling mechanism:

- For platform invoke, a delegate is marshaled as an unmanaged function pointer by default.

- For COM interop, a delegate is marshaled as a COM interface of type **_Delegate** by default. The **_Delegate** interface is defined in the Mscorlib.tlb type library and contains the Delegate.DynamicInvoke method, which enables you to call the method that the delegate references.

The following table shows the marshaling options for the managed delegate data type. The MarshalAsAttribute attribute provides several UnmanagedType enumeration values to marshal delegates.

ENUMERATION TYPE	DESCRIPTION OF UNMANAGED FORMAT
UnmanagedType.FunctionPtr	An unmanaged function pointer.
UnmanagedType.Interface	An interface of type **_Delegate**, as defined in Mscorlib.tlb.

Consider the following example code in which the methods of `DelegateTestInterface` are exported to a COM type library. Notice that only delegates marked with the **ref** (or **ByRef**) keyword are passed as In/Out parameters.

```
using System;
using System.Runtime.InteropServices;

public interface DelegateTest {
void m1(Delegate d);
void m2([MarshalAs(UnmanagedType.Interface)] Delegate d);
void m3([MarshalAs(UnmanagedType.Interface)] ref Delegate d);
void m4([MarshalAs(UnmanagedType.FunctionPtr)] Delegate d);
void m5([MarshalAs(UnmanagedType.FunctionPtr)] ref Delegate d);
}
```

Type library representation

```
importlib("mscorlib.tlb");
interface DelegateTest : IDispatch {
[id(…)] HRESULT m1([in] _Delegate* d);
[id(…)] HRESULT m2([in] _Delegate* d);
[id(…)] HRESULT m3([in, out] _Delegate** d);
[id()] HRESULT m4([in] int d);
[id()] HRESULT m5([in, out] int *d);
   };
```

A function pointer can be dereferenced, just as any other unmanaged function pointer can be dereferenced.

In this example, when the two delegates are marshaled as UnmanagedType.FunctionPtr, the result is an `int` and a pointer to an `int`. Because delegate types are being marshaled, `int` here represents a pointer to a void (`void*`), which is the address of the delegate in memory. In other words, this result is specific to 32-bit Windows systems, since `int` here represents the size of the function pointer.

NOTE

A reference to the function pointer to a managed delegate held by unmanaged code does not prevent the common language runtime from performing garbage collection on the managed object.

For example, the following code is incorrect because the reference to the `cb` object, passed to the `SetChangeHandler` method, does not keep `cb` alive beyond the life of the `Test` method. Once the `cb` object is garbage collected, the function pointer passed to `SetChangeHandler` is no longer valid.

```
public class ExternalAPI {
   [DllImport("External.dll")]
   public static extern void SetChangeHandler(
      [MarshalAs(UnmanagedType.FunctionPtr)]ChangeDelegate d);
}
public delegate bool ChangeDelegate([MarshalAs(UnmanagedType.LPWStr) string S);
public class CallBackClass {
   public bool OnChange(string S){ return true;}
}
internal class DelegateTest {
   public static void Test() {
      CallBackClass cb = new CallBackClass();
      // Caution: The following reference on the cb object does not keep the
      // object from being garbage collected after the Main method
      // executes.
      ExternalAPI.SetChangeHandler(new ChangeDelegate(cb.OnChange));
   }
}
```

To compensate for unexpected garbage collection, the caller must ensure that the `cb` object is kept alive as long as the unmanaged function pointer is in use. Optionally, you can have the unmanaged code notify the managed code when the function pointer is no longer needed, as the following example shows.

```
internal class DelegateTest {
   CallBackClass cb;
   // Called before ever using the callback function.
   public static void SetChangeHandler() {
      cb = new CallBackClass();
      ExternalAPI.SetChangeHandler(new ChangeDelegate(cb.OnChange));
   }
   // Called after using the callback function for the last time.
   public static void RemoveChangeHandler() {
      // The cb object can be collected now. The unmanaged code is
      // finished with the callback function.
      cb = null;
   }
}
```

Default marshaling for value types

Most value types, such as integers and floating-point numbers, are blittable and do not require marshaling. Other non-blittable types have dissimilar representations in managed and unmanaged memory and do require marshaling. Still other types require explicit formatting across the interoperation boundary.

This section provides information on the following formatted value types:

- Value Types Used in Platform Invoke
- Value Types Used in COM Interop

In addition to describing formatted types, this topic identifies System Value Types that have unusual marshaling behavior.

A formatted type is a complex type that contains information that explicitly controls the layout of its members in memory. The member layout information is provided using the StructLayoutAttribute attribute. The layout can be one of the following LayoutKind enumeration values:

- **LayoutKind.Automatic**

  Indicates that the common language runtime is free to reorder the members of the type for efficiency. However, when a value type is passed to unmanaged code, the layout of the members is predictable. An attempt to marshal such a structure automatically causes an exception.

- **LayoutKind.Sequential**

  Indicates that the members of the type are to be laid out in unmanaged memory in the same order in which they appear in the managed type definition.

- **LayoutKind.Explicit**

  Indicates that the members are laid out according to the FieldOffsetAttribute supplied with each field.

Value Types Used in Platform Invoke

In the following example the `Point` and `Rect` types provide member layout information using the **StructLayoutAttribute**.

```
Imports System.Runtime.InteropServices
<StructLayout(LayoutKind.Sequential)> Public Structure Point
   Public x As Integer
   Public y As Integer
End Structure
<StructLayout(LayoutKind.Explicit)> Public Structure Rect
   <FieldOffset(0)> Public left As Integer
   <FieldOffset(4)> Public top As Integer
   <FieldOffset(8)> Public right As Integer
   <FieldOffset(12)> Public bottom As Integer
End Structure
```

```
using System.Runtime.InteropServices;
[StructLayout(LayoutKind.Sequential)]
public struct Point {
   public int x;
   public int y;
}

[StructLayout(LayoutKind.Explicit)]
public struct Rect {
   [FieldOffset(0)] public int left;
   [FieldOffset(4)] public int top;
   [FieldOffset(8)] public int right;
   [FieldOffset(12)] public int bottom;
}
```

When marshaled to unmanaged code, these formatted types are marshaled as C-style structures. This provides an easy way of calling an unmanaged API that has structure arguments. For example, the `POINT` and `RECT` structures can be passed to the Microsoft Windows API **PtInRect** function as follows:

```
BOOL PtInRect(const RECT *lprc, POINT pt);
```

You can pass structures using the following platform invoke definition:

```
Friend Class NativeMethods
    Friend Declare Auto Function PtInRect Lib "User32.dll" (
        ByRef r As Rect, p As Point) As Boolean
End Class
```

```
internal static class NativeMethods
{
   [DllImport("User32.dll")]
   internal static extern bool PtInRect(ref Rect r, Point p);
}
```

The `Rect` value type must be passed by reference because the unmanaged API is expecting a pointer to a `RECT` to be passed to the function. The `Point` value type is passed by value because the unmanaged API expects the `POINT` to be passed on the stack. This subtle difference is very important. References are passed to unmanaged code as pointers. Values are passed to unmanaged code on the stack.

NOTE

When a formatted type is marshaled as a structure, only the fields within the type are accessible. If the type has methods, properties, or events, they are inaccessible from unmanaged code.

Classes can also be marshaled to unmanaged code as C-style structures, provided they have fixed member layout. The member layout information for a class is also provided with the StructLayoutAttribute attribute. The main difference between value types with fixed layout and classes with fixed layout is the way in which they are marshaled to unmanaged code. Value types are passed by value (on the stack) and consequently any changes made to the members of the type by the callee are not seen by the caller. Reference types are passed by reference (a reference to the type is passed on the stack); consequently, all changes made to blittable-type members of a type by the callee are seen by the caller.

NOTE

If a reference type has members of non-blittable types, conversion is required twice: the first time when an argument is passed to the unmanaged side and the second time on return from the call. Due to this added overhead, In/Out parameters must be explicitly applied to an argument if the caller wants to see changes made by the callee.

In the following example, the `SystemTime` class has sequential member layout and can be passed to the Windows API **GetSystemTime** function.

```
<StructLayout(LayoutKind.Sequential)> Public Class SystemTime
   Public wYear As System.UInt16
   Public wMonth As System.UInt16
   Public wDayOfWeek As System.UInt16
   Public wDay As System.UInt16
   Public wHour As System.UInt16
   Public wMinute As System.UInt16
   Public wSecond As System.UInt16
   Public wMilliseconds As System.UInt16
End Class
```

```
[StructLayout(LayoutKind.Sequential)]
   public class SystemTime {
   public ushort wYear;
   public ushort wMonth;
   public ushort wDayOfWeek;
   public ushort wDay;
   public ushort wHour;
   public ushort wMinute;
   public ushort wSecond;
   public ushort wMilliseconds;
}
```

The **GetSystemTime** function is defined as follows:

```
void GetSystemTime(SYSTEMTIME* SystemTime);
```

The equivalent platform invoke definition for **GetSystemTime** is as follows:

```
Friend Class NativeMethods
    Friend Declare Auto Sub GetSystemTime Lib "Kernel32.dll" (
        ByVal sysTime As SystemTime)
End Class
```

```
internal static class NativeMethods
{
   [DllImport("Kernel32.dll", CharSet = CharSet.Auto)]
   internal static extern void GetSystemTime(SystemTime st);
}
```

Notice that the `SystemTime` argument is not typed as a reference argument because `SystemTime` is a class, not a value type. Unlike value types, classes are always passed by reference.

The following code example shows a different `Point` class that has a method called `SetXY`. Because the type has sequential layout, it can be passed to unmanaged code and marshaled as a structure. However, the `SetXY` member is not callable from unmanaged code, even though the object is passed by reference.

```
<StructLayout(LayoutKind.Sequential)> Public Class Point
   Private x, y As Integer
   Public Sub SetXY(x As Integer, y As Integer)
      Me.x = x
      Me.y = y
   End Sub
End Class
```

```
[StructLayout(LayoutKind.Sequential)]
public class Point {
   int x, y;
   public void SetXY(int x, int y){
      this.x = x;
      this.y = y;
   }
}
```

Value Types Used in COM Interop

Formatted types can also be passed to COM interop method calls. In fact, when exported to a type library, value types are automatically converted to structures. As the following example shows, the `Point` value type becomes a type definition (typedef) with the name `Point`. All references to the `Point` value type elsewhere in the type library are replaced with the `Point` typedef.

Type library representation

```
typedef struct tagPoint {
   int x;
   int y;
} Point;
interface _Graphics {
   …
   HRESULT SetPoint ([in] Point p)
   HRESULT SetPointRef ([in,out] Point *p)
   HRESULT GetPoint ([out,retval] Point *p)
}
```

The same rules used to marshal values and references to platform invoke calls are used when marshaling through COM interfaces. For example, when an instance of the `Point` value type is passed from the .NET Framework to COM, the `Point` is passed by value. If the `Point` value type is passed by reference, a pointer to a `Point` is passed on the stack. The interop marshaler does not support higher levels of indirection (**Point** **) in either direction.

NOTE

Structures having the LayoutKind enumeration value set to **Explicit** cannot be used in COM interop because the exported type library cannot express an explicit layout.

System Value Types

The System namespace has several value types that represent the boxed form of the runtime primitive types. For example, the value type System.Int32 structure represents the boxed form of **ELEMENT_TYPE_I4**. Instead of marshaling these types as structures, as other formatted types are, you marshal them in the same way as the primitive types they box. **System.Int32** is therefore marshaled as **ELEMENT_TYPE_I4** instead of as a structure containing a single member of type **long**. The following table contains a list of the value types in the **System** namespace that are boxed representations of primitive types.

SYSTEM VALUE TYPE	ELEMENT TYPE
System.Boolean	**ELEMENT_TYPE_BOOLEAN**
System.SByte	**ELEMENT_TYPE_I1**
System.Byte	**ELEMENT_TYPE_UI1**
System.Char	**ELEMENT_TYPE_CHAR**
System.Int16	**ELEMENT_TYPE_I2**
System.UInt16	**ELEMENT_TYPE_U2**
System.Int32	**ELEMENT_TYPE_I4**
System.UInt32	**ELEMENT_TYPE_U4**
System.Int64	**ELEMENT_TYPE_I8**
System.UInt64	**ELEMENT_TYPE_U8**
System.Single	**ELEMENT_TYPE_R4**

SYSTEM VALUE TYPE	ELEMENT TYPE
System.Double	**ELEMENT_TYPE_R8**
System.String	**ELEMENT_TYPE_STRING**
System.IntPtr	**ELEMENT_TYPE_I**
System.UIntPtr	**ELEMENT_TYPE_U**

Some other value types in the **System** namespace are handled differently. Because the unmanaged code already has well-established formats for these types, the marshaler has special rules for marshaling them. The following table lists the special value types in the **System** namespace, as well as the unmanaged type they are marshaled to.

SYSTEM VALUE TYPE	IDL TYPE
System.DateTime	**DATE**
System.Decimal	**DECIMAL**
System.Guid	**GUID**
System.Drawing.Color	**OLE_COLOR**

The following code shows the definition of the unmanaged types **DATE**, **GUID**, **DECIMAL**, and **OLE_COLOR** in the Stdole2 type library.

Type library representation

```
typedef double DATE;
typedef DWORD OLE_COLOR;

typedef struct tagDEC {
    USHORT    wReserved;
    BYTE      scale;
    BYTE      sign;
    ULONG     Hi32;
    ULONGLONG Lo64;
} DECIMAL;

typedef struct tagGUID {
    DWORD Data1;
    WORD  Data2;
    WORD  Data3;
    BYTE  Data4[ 8 ];
} GUID;
```

The following code shows the corresponding definitions in the managed `IValueTypes` interface.

```
Public Interface IValueTypes
   Sub M1(d As System.DateTime)
   Sub M2(d As System.Guid)
   Sub M3(d As System.Decimal)
   Sub M4(d As System.Drawing.Color)
End Interface
```

```
public interface IValueTypes {
   void M1(System.DateTime d);
   void M2(System.Guid d);
   void M3(System.Decimal d);
   void M4(System.Drawing.Color d);
}
```

Type library representation

```
[…]
interface IValueTypes : IDispatch {
   HRESULT M1([in] DATE d);
   HRESULT M2([in] GUID d);
   HRESULT M3([in] DECIMAL d);
   HRESULT M4([in] OLE_COLOR d);
};
```

See also

- Blittable and Non-Blittable Types
- Copying and Pinning
- Default Marshaling for Arrays
- Default Marshaling for Objects
- Default Marshaling for Strings

Blittable and Non-Blittable Types

5/9/2019 • 2 minutes to read • Edit Online

Most data types have a common representation in both managed and unmanaged memory and do not require special handling by the interop marshaler. These types are called *blittable types* because they do not require conversion when they are passed between managed and unmanaged code.

Structures that are returned from platform invoke calls must be blittable types. Platform invoke does not support non-blittable structures as return types.

The following types from the System namespace are blittable types:

- System.Byte
- System.SByte
- System.Int16
- System.UInt16
- System.Int32
- System.UInt32
- System.Int64
- System.UInt64
- System.IntPtr
- System.UIntPtr
- System.Single
- System.Double

The following complex types are also blittable types:

- One-dimensional arrays of blittable types, such as an array of integers. However, a type that contains a variable array of blittable types is not itself blittable.
- Formatted value types that contain only blittable types (and classes if they are marshaled as formatted types). For more information about formatted value types, see Default marshaling for value types.

Object references are not blittable. This includes an array of references to objects that are blittable by themselves. For example, you can define a structure that is blittable, but you cannot define a blittable type that contains an array of references to those structures.

As an optimization, arrays of blittable types and classes that contain only blittable members are pinned instead of copied during marshaling. These types can appear to be marshaled as In/Out parameters when the caller and callee are in the same apartment. However, these types are actually marshaled as In parameters, and you must apply the InAttribute and OutAttribute attributes if you want to marshal the argument as an In/Out parameter.

Some managed data types require a different representation in an unmanaged environment. These non-blittable data types must be converted into a form that can be marshaled. For example, managed strings are non-blittable types because they must be converted into string objects before they can be marshaled.

The following table lists non-blittable types from the System namespace. Delegates, which are data structures that refer to a static method or to a class instance, are also non-blittable.

NON-BLITTABLE TYPE	DESCRIPTION
System.Array	Converts to a C-style array or a `SAFEARRAY`.
System.Boolean	Converts to a 1, 2, or 4-byte value with `true` as 1 or -1.
System.Char	Converts to a Unicode or ANSI character.
System.Class	Converts to a class interface.
System.Object	Converts to a variant or an interface.
System.Mdarray	Converts to a C-style array or a `SAFEARRAY`.
System.String	Converts to a string terminating in a null reference or to a BSTR.
System.Valuetype	Converts to a structure with a fixed memory layout.
System.Szarray	Converts to a C-style array or a `SAFEARRAY`.

Class and object types are supported only by COM interop. For corresponding types in Visual Basic, C#, and C++, see the Class Library Overview.

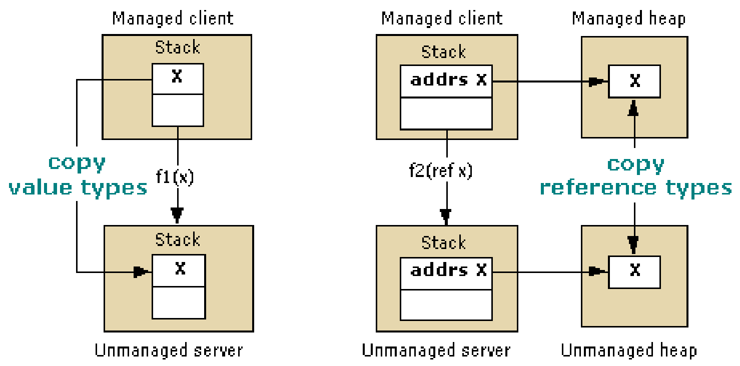
See also

- Default Marshaling Behavior

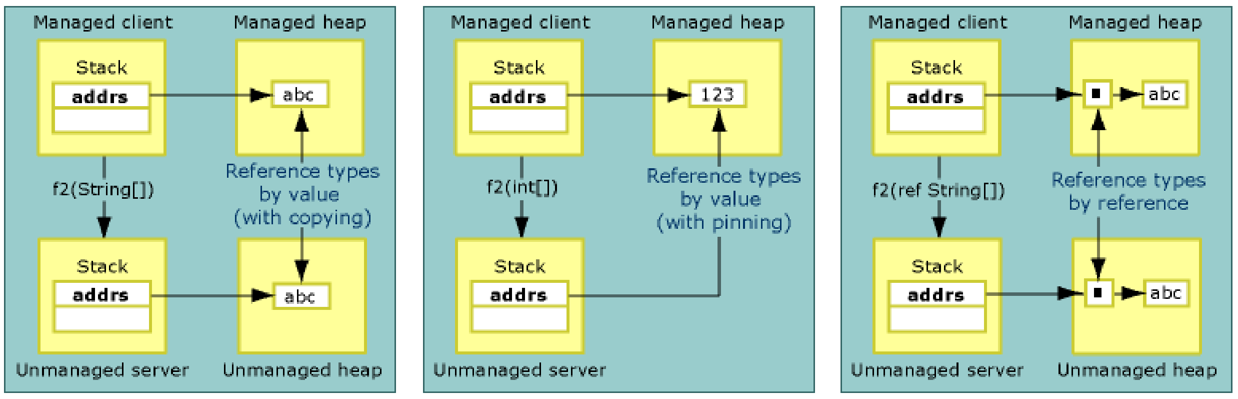
Copying and Pinning

4/26/2019 • 4 minutes to read • Edit Online

When marshaling data, the interop marshaler can copy or pin the data being marshaled. Copying the data places a copy of data from one memory location in another memory location. The following illustration shows the differences between copying a value type and copying a type passed by reference from managed to unmanaged memory.



Method arguments passed by value are marshaled to unmanaged code as values on the stack. The copying process is direct. Arguments passed by reference are passed as pointers on the stack. Reference types are also passed by value and by reference. As the following illustration shows, reference types passed by value are either copied or pinned:



Pinning temporarily locks the data in its current memory location, thus keeping it from being relocated by the common language runtime's garbage collector. The marshaler pins data to reduce the overhead of copying and enhance performance. The type of the data determines whether it is copied or pinned during the marshaling process. Pinning is automatically performed during marshaling for objects such as String, however you can also manually pin memory using the GCHandle class.

Formatted Blittable Classes

Formatted blittable classes have fixed layout (formatted) and common data representation in both managed and unmanaged memory. When these types require marshaling, a pointer to the object in the heap is passed to the callee directly. The callee can change the contents of the memory location being referenced by the pointer.

NOTE

The callee can change the memory contents if the parameter is marked Out or In/Out. In contrast, the callee should avoid changing the contents when the parameter is set to marshal as In, which is the default for formatted blittable types. Modifying an In object generates problems when the same class is exported to a type library and used to make cross-apartment calls.

Formatted Non-Blittable Classes

Formatted non-blittable classes have fixed layout (formatted) but the data representation is different in managed and unmanaged memory. The data can require transformation under the following conditions:

- If a non-blittable class is marshaled by value, the callee receives a pointer to a copy of the data structure.
- If a non-blittable class is marshaled by reference, the callee receives a pointer to a pointer to a copy of the data structure.
- If the InAttribute attribute is set, this copy is always initialized with the instance's state, marshaling as necessary.
- If the OutAttribute attribute is set, the state is always copied back to the instance on return, marshaling as necessary.
- If both **InAttribute** and **OutAttribute** are set, both copies are required. If either attribute is omitted, the marshaler can optimize by eliminating either copy.

Reference Types

Reference types can be passed by value or by reference. When they are passed by value, a pointer to the type is passed on the stack. When passed by reference, a pointer to a pointer to the type is passed on the stack.

Reference types have the following conditional behavior:

- If a reference type is passed by value and it has members of non-blittable types, the types are converted twice:
  - When an argument is passed to the unmanaged side.
  - On return from the call.

  To avoid unnecessarily copying and conversion, these types are marshaled as In parameters. You must explicitly apply the **InAttribute** and **OutAttribute** attributes to an argument for the caller to see changes made by the callee.
- If a reference type is passed by value and it has only members of blittable types, it can be pinned during marshaling and any changes made to the members of the type by the callee are seen by the caller. Apply **InAttribute** and **OutAttribute** explicitly if you want this behavior. Without these directional attributes, the interop marshaler does not export directional information to the type library (it exports as In, which is the default) and this can cause problems with COM cross-apartment marshaling.
- If a reference type is passed by reference, it will be marshaled as In/Out by default.

System.String and System.Text.StringBuilder

When data is marshaled to unmanaged code by value or by reference, the marshaler typically copies the data to a secondary buffer (possibly converting character sets during the copy) and passes a reference to the buffer to the callee. Unless the reference is a **BSTR** allocated with **SysAllocString**, the reference is always allocated with

CoTaskMemAlloc.

As an optimization when either string type is marshaled by value (such as a Unicode character string), the marshaler passes the callee a direct pointer to managed strings in the internal Unicode buffer instead of copying it to a new buffer.

Caution

When a string is passed by value, the callee must never alter the reference passed by the marshaler. Doing so can corrupt the managed heap.

When a System.String is passed by reference, the marshaler copies the contents the string to a secondary buffer before making the call. It then copies the contents of the buffer into a new string on return from the call. This technique ensures that the immutable managed string remains unaltered.

When a System.Text.StringBuilder is passed by value, the marshaler passes a reference to the internal buffer of the **StringBuilder** directly to the caller. The caller and callee must agree on the size of the buffer. The caller is responsible for creating a **StringBuilder** of adequate length. The callee must take the necessary precautions to ensure that the buffer is not overrun. **StringBuilder** is an exception to the rule that reference types passed by value are passed as In parameters by default. It is always passed as In/Out.

See also

- Default Marshaling Behavior
- Directional Attributes
- Interop Marshaling

Default Marshaling for Arrays

4/28/2019 • 10 minutes to read • Edit Online

In an application consisting entirely of managed code, the common language runtime passes array types as In/Out parameters. In contrast, the interop marshaler passes an array as In parameters by default.

With pinning optimization, a blittable array can appear to operate as an In/Out parameter when interacting with objects in the same apartment. However, if you later export the code to a type library used to generate the cross-machine proxy, and that library is used to marshal your calls across apartments, the calls can revert to true In parameter behavior.

Arrays are complex by nature, and the distinctions between managed and unmanaged arrays warrant more information than other non-blittable types.

Managed Arrays

Managed array types can vary; however, the System.Array class is the base class of all array types. The **System.Array** class has properties for determining the rank, length, and lower and upper bounds of an array, as well as methods for accessing, sorting, searching, copying, and creating arrays.

These array types are dynamic and do not have a corresponding static type defined in the base class library. It is convenient to think of each combination of element type and rank as a distinct type of array. Therefore, a one-dimensional array of integers is of a different type than a one-dimensional array of double types. Similarly a two-dimensional array of integers is different from a one-dimensional array of integers. The bounds of the array are not considered when comparing types.

As the following table shows, any instance of a managed array must be of a specific element type, rank, and lower bound.

MANAGED ARRAY TYPE	ELEMENT TYPE	RANK	LOWER BOUND	SIGNATURE NOTATION
ELEMENT_TYPE_ARRAY	Specified by type.	Specified by rank.	Optionally specified by bounds.	*type* **[** *n,m* **]**
ELEMENT_TYPE_CLASS	Unknown	Unknown	Unknown	**System.Array**
ELEMENT_TYPE_SZARRAY	Specified by type.	1	0	*type* **[** *n* **]**

Unmanaged Arrays

Unmanaged arrays are either COM-style safe arrays or C-style arrays with fixed or variable length. Safe arrays are self-describing arrays that carry the type, rank, and bounds of the associated array data. C-style arrays are one-dimensional typed arrays with a fixed lower bound of 0. The marshaling service has limited support for both types of arrays.

Passing Array Parameters to .NET Code

Both C-style arrays and safe arrays can be passed to .NET code from unmanaged code as either a safe array or a C-style array. The following table shows the unmanaged type value and the imported type.

UNMANAGED TYPE	IMPORTED TYPE
SafeArray(*Type* **)**	**ELEMENT_TYPE_SZARRAY <** *ConvertedType* **>** Rank = 1, lower bound = 0. Size is known only if provided in the managed signature. Safe arrays that are not rank = 1 or lower bound = 0 cannot be marshaled as **SZARRAY**.
Type **[]**	**ELEMENT_TYPE_SZARRAY <** *ConvertedType* **>** Rank = 1, lower bound = 0. Size is known only if provided in the managed signature.

Safe Arrays

When a safe array is imported from a type library to a .NET assembly, the array is converted to a one-dimensional array of a known type (such as **int**). The same type conversion rules that apply to parameters also apply to array elements. For example, a safe array of **BSTR** types becomes a managed array of strings and a safe array of variants becomes a managed array of objects. The **SAFEARRAY** element type is captured from the type library and saved in the **SAFEARRAY** value of the UnmanagedType enumeration.

Because the rank and bounds of the safe array cannot be determined from the type library, the rank is assumed to equal 1 and the lower bound is assumed to equal 0. The rank and bounds must be defined in the managed signature produced by the Type Library Importer (Tlbimp.exe). If the rank passed to the method at run time differs, a SafeArrayRankMismatchException is thrown. If the type of the array passed at run time differs, a SafeArrayTypeMismatchException is thrown. The following example shows safe arrays in managed and unmanaged code.

Unmanaged signature

```
HRESULT New1([in] SAFEARRAY( int ) ar);
HRESULT New2([in] SAFEARRAY( DATE ) ar);
HRESULT New3([in, out] SAFEARRAY( BSTR ) *ar);
```

Managed signature

```
Sub New1(<MarshalAs(UnmanagedType.SafeArray, SafeArraySubType:=VT_I4)> _
   ar() As Integer)
Sub New2(<MarshalAs(UnmanagedType.SafeArray, SafeArraySubType:=VT_DATE)> _
   ar() As DateTime)
Sub New3(ByRef <MarshalAs(UnmanagedType.SafeArray, SafeArraySubType:=VT_BSTR)> _
   ar() As String)
```

```
void New1([MarshalAs(UnmanagedType.SafeArray, SafeArraySubType=VT_I4)] int[] ar) ;
void New2([MarshalAs(UnmanagedType.SafeArray, SafeArraySubType=VT_DATE)]
   DateTime[] ar);
void New3([MarshalAs(UnmanagedType.SafeArray, SafeArraySubType=VT_BSTR)]
   ref String[] ar);
```

Multidimensional, or nonzero-bound safe arrays, can be marshaled into managed code if the method signature produced by Tlbimp.exe is modified to indicate an element type of **ELEMENT_TYPE_ARRAY** instead of **ELEMENT_TYPE_SZARRAY**. Alternatively, you can use the **/sysarray** switch with Tlbimp.exe to import all arrays as System.Array objects. In cases where the array being passed is known to be multidimensional, you can edit the Microsoft intermediate language (MSIL) code produced by Tlbimp.exe and then recompile it. For details about how to modify MSIL code, see Customizing Runtime Callable Wrappers.

C-Style Arrays

When a C-style array is imported from a type library to a .NET assembly, the array is converted to **ELEMENT_TYPE_SZARRAY**.

The array element type is determined from the type library and preserved during the import. The same conversion rules that apply to parameters also apply to array elements. For example, an array of **LPStr** types becomes an array of **String** types. Tlbimp.exe captures the array element type and applies the MarshalAsAttribute attribute to the parameter.

The array rank is assumed to equal 1. If the rank is greater than 1, the array is marshaled as a one-dimensional array in column-major order. The lower bound always equals 0.

Type libraries can contain arrays of fixed or variable length. Tlbimp.exe can import only fixed-length arrays from type libraries because type libraries lack the information needed to marshal variable-length arrays. With fixed-length arrays, the size is imported from the type library and captured in the **MarshalAsAttribute** that is applied to the parameter.

You must manually define type libraries containing variable-length arrays, as shown in the following example.

Unmanaged signature

```
HRESULT New1(int ar[10]);
HRESULT New2(double ar[10][20]);
HRESULT New3(LPWStr ar[10]);
```

Managed signature

```
Sub New1(<MarshalAs(UnmanagedType.LPArray, SizeConst=10)> _
   ar() As Integer)
Sub New2(<MarshalAs(UnmanagedType.LPArray, SizeConst=200)> _
   ar() As Double)
Sub New2(<MarshalAs(UnmanagedType.LPArray, _
   ArraySubType=UnmanagedType.LPWStr, SizeConst=10)> _
   ar() As String)
```

```
void New1([MarshalAs(UnmanagedType.LPArray, SizeConst=10)] int[] ar);
void New2([MarshalAs(UnmanagedType.LPArray, SizeConst=200)] double[] ar);
void New2([MarshalAs(UnmanagedType.LPArray,
   ArraySubType=UnmanagedType.LPWStr, SizeConst=10)] String[] ar);
```

Although you can apply the **size_is** or **length_is** attributes to an array in Interface Definition Language (IDL) source to convey the size to a client, the Microsoft Interface Definition Language (MIDL) compiler does not propagate that information to the type library. Without knowing the size, the interop marshaling service cannot marshal the array elements. Consequently, variable-length arrays are imported as reference arguments. For example:

Unmanaged signature

```
HRESULT New1(int ar[]);
HRESULT New2(int ArSize, [size_is(ArSize)] double ar[]);
HRESULT New3(int ElemCnt, [length_is(ElemCnt)] LPStr ar[]);
```

Managed signature

```
Sub New1(ByRef ar As Integer)
Sub New2(ByRef ar As Double)
Sub New3(ByRef ar As String)
```

```
void New1(ref int ar);
void New2(ref double ar);
void New3(ref String ar);
```

You can provide the marshaler with the array size by editing the Microsoft intermediate language (MSIL) code produced by Tlbimp.exe and then recompiling it. For details about how to modify MSIL code, see Customizing Runtime Callable Wrappers. To indicate the number of elements in the array, apply the MarshalAsAttribute type to the array parameter of the managed method definition in one of the following ways:

- Identify another parameter that contains the number of elements in the array. The parameters are identified by position, starting with the first parameter as number 0.

  ```
  Sub [New](ElemCnt As Integer, _
     \<MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=1)> _
     ar() As Integer)
  ```

  ```
  void New(
     int ElemCnt,
     [MarshalAs(UnmanagedType.LPArray, SizeParamIndex=0)] int[] ar );
  ```

- Define the size of the array as a constant. For example:

  ```
  Sub [New](\<MarshalAs(UnmanagedType.LPArray, SizeConst:=128)> _
     ar() As Integer)
  ```

  ```
  void New(
     [MarshalAs(UnmanagedType.LPArray, SizeConst=128)] int[] ar );
  ```

When marshaling arrays from unmanaged code to managed code, the marshaler checks the **MarshalAsAttribute** associated with the parameter to determine the array size. If the array size is not specified, only one element is marshaled.

NOTE
The **MarshalAsAttribute** has no effect on marshaling managed arrays to unmanaged code. In that direction, the array size is determined by examination. There is no way to marshal a subset of a managed array.

The interop marshaler uses the **CoTaskMemAlloc** and **CoTaskMemFree** methods to allocate and retrieve memory. Memory allocation performed by unmanaged code must also use these methods.

Passing Arrays to COM

All managed array types can be passed to unmanaged code from managed code. Depending on the managed type and the attributes applied to it, the array can be accessed as a safe array or a C-style array, as shown in the following table.

MANAGED ARRAY TYPE	EXPORTED AS
ELEMENT_TYPE_SZARRAY **<** *type* **>**	UnmanagedType **.SafeArray(** *type* **)** **UnmanagedType.LPArray** Type is provided in the signature. Rank is always 1, lower bound is always 0. Size is always known at run time.
ELEMENT_TYPE_ARRAY **<** *type* **>** **<** *rank* **>**[**<** *bounds* **>**]	**UnmanagedType.SafeArray(** *type* **)** **UnmanagedType.LPArray** Type, rank, bounds are provided in the signature. Size is always known at run time.
ELEMENT_TYPE_CLASS **<**System.Array**>**	**UT_Interface** **UnmanagedType.SafeArray(** *type* **)** Type, rank, bounds, and size are always known at run time.

There is a limitation in OLE Automation relating to arrays of structures that contain LPSTR or LPWSTR. Therefore, **String** fields have to be marshaled as **UnmanagedType.BSTR**. Otherwise, an exception will be thrown.

ELEMENT_TYPE_SZARRAY

When a method containing an **ELEMENT_TYPE_SZARRAY** parameter (one-dimensional array) is exported from a .NET assembly to a type library, the array parameter is converted to a **SAFEARRAY** of a given type. The same conversion rules apply to the array element types. The contents of the managed array are automatically copied from managed memory into the **SAFEARRAY**. For example:

Managed signature

```
Sub [New](ar() As Long)
Sub [New](ar() As String)
```

```
void New(long[] ar );
void New(String[] ar );
```

Unmanaged signature

```
HRESULT New([in] SAFEARRAY( long ) ar);
HRESULT New([in] SAFEARRAY( BSTR ) ar);
```

The rank of the safe arrays is always 1 and the lower bound is always 0. The size is determined at run time by the size of the managed array being passed.

The array can also be marshaled as a C-style array by using the MarshalAsAttribute attribute. For example:

Managed signature

```
Sub [New](<MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=1)> _
   ar() As Long, size as Integer)
Sub [New](<MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=1)> _
   ar() As String, size as Integer)
Sub [New](<MarshalAs(UnmanagedType.LPArray, _
   ArraySubType= UnmanagedType.LPStr, SizeParamIndex:=1)> _
   ar() As String, size as Integer)
```

```
void New([MarshalAs(UnmanagedType.LPArray, SizeParamIndex=1)]
   long [] ar, int size );
void New([MarshalAs(UnmanagedType.LPArray, SizeParamIndex=1)]
   String [] ar, int size );
void New([MarshalAs(UnmanagedType.LPArray, ArraySubType=
   UnmanagedType.LPStr, SizeParamIndex=1)]
   String [] ar, int size );
```

Unmanaged signature

```
HRESULT New(long ar[]);
HRESULT New(BSTR ar[]);
HRESULT New(LPStr ar[]);
```

Although the marshaler has the length information needed to marshal the array, the array length is usually passed as a separate argument to convey the length to the callee.

ELEMENT_TYPE_ARRAY

When a method containing an **ELEMENT_TYPE_ARRAY** parameter is exported from a .NET assembly to a type library, the array parameter is converted to a **SAFEARRAY** of a given type. The contents of the managed array are automatically copied from managed memory into the **SAFEARRAY**. For example:

Managed signature

```
Sub [New](ar(,) As Long)
Sub [New](ar(,) As String)
```

```
void New( long [,] ar );
void New( String [,] ar );
```

Unmanaged signature

```
HRESULT New([in] SAFEARRAY( long ) ar);
HRESULT New([in] SAFEARRAY( BSTR ) ar);
```

The rank, size, and bounds of the safe arrays are determined at run time by the characteristics of the managed array.

The array can also be marshaled as a C-style array by applying the MarshalAsAttribute attribute. For example:

Managed signature

```
Sub [New](<MarshalAs(UnmanagedType.LPARRAY, SizeParamIndex:=1)> _
   ar(,) As Long, size As Integer)
Sub [New](<MarshalAs(UnmanagedType.LPARRAY, _
   ArraySubType:=UnmanagedType.LPStr, SizeParamIndex:=1)> _
   ar(,) As String, size As Integer)
```

```
void New([MarshalAs(UnmanagedType.LPARRAY, SizeParamIndex=1)]
   long [,] ar, int size );
void New([MarshalAs(UnmanagedType.LPARRAY,
   ArraySubType= UnmanagedType.LPStr, SizeParamIndex=1)]
   String [,] ar, int size );
```

Unmanaged signature

```
HRESULT New(long ar[]);
HRESULT New(LPStr ar[]);
```

Nested arrays cannot be marshaled. For example, the following signature generates an error when exported with the Type Library Exporter (Tlbexp.exe).

Managed signature

```
Sub [New](ar()()() As Long)
```

```
void New(long [][][] ar );
```

ELEMENT_TYPE_CLASS <System.Array>

When a method containing a System.Array parameter is exported from a .NET assembly to a type library, the array parameter is converted to an **_Array** interface. The contents of the managed array are accessible only through the methods and properties of the **_Array** interface. **System.Array** can also be marshaled as a **SAFEARRAY** by using the MarshalAsAttribute attribute. When marshaled as a safe array, the array elements are marshaled as variants. For example:

Managed signature

```
Sub New1( ar As System.Array )
Sub New2( <MarshalAs(UnmanagedType.Safe array)> ar As System.Array )
```

```
void New1( System.Array ar );
void New2( [MarshalAs(UnmanagedType.Safe array)] System.Array ar );
```

Unmanaged signature

```
HRESULT New([in] _Array *ar);
HRESULT New([in] SAFEARRAY(VARIANT) ar);
```

Arrays within Structures

Unmanaged structures can contain embedded arrays. By default, these embedded array fields are marshaled as a SAFEARRAY. In the following example, `s1` is an embedded array that is allocated directly within the structure itself.

Unmanaged representation

```
struct MyStruct {
    short s1[128];
}
```

Arrays can be marshaled as UnmanagedType, which requires you to set the MarshalAsAttribute field. The size can be set only as a constant. The following code shows the corresponding managed definition of `MyStruct`.

```vb
Public Structure <StructLayout(LayoutKind.Sequential)> MyStruct
   Public <MarshalAs(UnmanagedType.ByValArray, SizeConst := 128)> _
     s1() As Short
End Structure
```

```csharp
[StructLayout(LayoutKind.Sequential)]
public struct MyStruct {
   [MarshalAs(UnmanagedType.ByValArray, SizeConst=128)] public short[] s1;
}
```

See also

- Default Marshaling Behavior
- Blittable and Non-Blittable Types
- Directional Attributes
- Copying and Pinning

Default Marshaling for Objects

4/28/2019 • 10 minutes to read • Edit Online

Parameters and fields typed as System.Object can be exposed to unmanaged code as one of the following types:

- A variant when the object is a parameter.
- An interface when the object is a structure field.

Only COM interop supports marshaling for object types. The default behavior is to marshal objects to COM variants. These rules apply only to the type **Object** and do not apply to strongly typed objects that derive from the **Object** class.

Marshaling Options

The following table shows the marshaling options for the **Object** data type. The MarshalAsAttribute attribute provides several UnmanagedType enumeration values to marshal objects.

ENUMERATION TYPE	DESCRIPTION OF UNMANAGED FORMAT
UnmanagedType.Struct (default for parameters)	A COM-style variant.
UnmanagedType.Interface	An **IDispatch** interface, if possible; otherwise, an **IUnknown** interface.
UnmanagedType.IUnknown (default for fields)	An **IUnknown** interface.
UnmanagedType.IDispatch	An **IDispatch** interface.

The following example shows the managed interface definition for `MarshalObject`.

```
Interface MarshalObject
   Sub SetVariant(o As Object)
   Sub SetVariantRef(ByRef o As Object)
   Function GetVariant() As Object

   Sub SetIDispatch( <MarshalAs(UnmanagedType.IDispatch)> o As Object)
   Sub SetIDispatchRef(ByRef <MarshalAs(UnmanagedType.IDispatch)> o _
      As Object)
   Function GetIDispatch() As <MarshalAs(UnmanagedType.IDispatch)> Object
   Sub SetIUnknown( <MarshalAs(UnmanagedType.IUnknown)> o As Object)
   Sub SetIUnknownRef(ByRef <MarshalAs(UnmanagedType.IUnknown)> o _
      As Object)
   Function GetIUnknown() As <MarshalAs(UnmanagedType.IUnknown)> Object
End Interface
```

```
interface MarshalObject {
   void SetVariant(Object o);
   void SetVariantRef(ref Object o);
   Object GetVariant();

   void SetIDispatch ([MarshalAs(UnmanagedType.IDispatch)]Object o);
   void SetIDispatchRef([MarshalAs(UnmanagedType.IDispatch)]ref Object o);
   [MarshalAs(UnmanagedType.IDispatch)] Object GetIDispatch();
   void SetIUnknown ([MarshalAs(UnmanagedType.IUnknown)]Object o);
   void SetIUnknownRef([MarshalAs(UnmanagedType.IUnknown)]ref Object o);
   [MarshalAs(UnmanagedType.IUnknown)] Object GetIUnknown();
}
```

The following code exports the `MarshalObject` interface to a type library.

```
interface MarshalObject {
   HRESULT SetVariant([in] VARIANT o);
   HRESULT SetVariantRef([in,out] VARIANT *o);
   HRESULT GetVariant([out,retval] VARIANT *o)
   HRESULT SetIDispatch([in] IDispatch *o);
   HRESULT SetIDispatchRef([in,out] IDispatch **o);
   HRESULT GetIDispatch([out,retval] IDispatch **o)
   HRESULT SetIUnknown([in] IUnknown *o);
   HRESULT SetIUnknownRef([in,out] IUnknown **o);
   HRESULT GetIUnknown([out,retval] IUnknown **o)
}
```

NOTE

The interop marshaler automatically frees any allocated object inside the variant after the call.

The following example shows a formatted value type.

```
Public Structure ObjectHolder
   Dim o1 As Object
   <MarshalAs(UnmanagedType.IDispatch)> Public o2 As Object
End Structure
```

```
public struct ObjectHolder {
   Object o1;
   [MarshalAs(UnmanagedType.IDispatch)]public Object o2;
}
```

The following code exports the formatted type to a type library.

```
struct ObjectHolder {
   VARIANT o1;
   IDispatch *o2;
}
```

Marshaling Object to Interface

When an object is exposed to COM as an interface, that interface is the class interface for the managed type Object (the _**Object** interface). This interface is typed as an **IDispatch** (UnmanagedType) or an **IUnknown** (**UnmanagedType.IUnknown**) in the resulting type library. COM clients can dynamically invoke the members of the managed class or any members implemented by its derived classes through the **_Object** interface. The client can also call **QueryInterface** to obtain any other interface explicitly implemented by the managed type.

Marshaling Object to Variant

When an object is marshaled to a variant, the internal variant type is determined at run time, based on the following rules:

- If the object reference is null (**Nothing** in Visual Basic), the object is marshaled to a variant of type **VT_EMPTY**.
- If the object is an instance of any type listed in the following table, the resulting variant type is determined by the rules built into the marshaler and shown in the table.
- Other objects that need to explicitly control the marshaling behavior can implement the IConvertible interface. In that case, the variant type is determined by the type code returned from the IConvertible.GetTypeCode method. Otherwise, the object is marshaled as a variant of type **VT_UNKNOWN**.

Marshaling System Types to Variant

The following table shows managed object types and their corresponding COM variant types. These types are converted only when the signature of the method being called is of type System.Object.

OBJECT TYPE	COM VARIANT TYPE
Null object reference (**Nothing** in Visual Basic).	**VT_EMPTY**
System.DBNull	**VT_NULL**
System.Runtime.InteropServices.ErrorWrapper	**VT_ERROR**
System.Reflection.Missing	**VT_ERROR** with **E_PARAMNOTFOUND**
System.Runtime.InteropServices.DispatchWrapper	**VT_DISPATCH**
System.Runtime.InteropServices.UnknownWrapper	**VT_UNKNOWN**
System.Runtime.InteropServices.CurrencyWrapper	**VT_CY**
System.Boolean	**VT_BOOL**
System.SByte	**VT_I1**
System.Byte	**VT_UI1**
System.Int16	**VT_I2**
System.UInt16	**VT_UI2**
System.Int32	**VT_I4**
System.UInt32	**VT_UI4**
System.Int64	**VT_I8**

OBJECT TYPE	COM VARIANT TYPE
System.UInt64	**VT_UI8**
System.Single	**VT_R4**
System.Double	**VT_R8**
System.Decimal	**VT_DECIMAL**
System.DateTime	**VT_DATE**
System.String	**VT_BSTR**
System.IntPtr	**VT_INT**
System.UIntPtr	**VT_UINT**
System.Array	**VT_ARRAY**

Using the `MarshalObject` interface defined in the previous example, the following code example demonstrates how to pass various types of variants to a COM server.

```
Dim mo As New MarshalObject()
mo.SetVariant(Nothing)         ' Marshal as variant of type VT_EMPTY.
mo.SetVariant(System.DBNull.Value) ' Marshal as variant of type VT_NULL.
mo.SetVariant(CInt(27))        ' Marshal as variant of type VT_I2.
mo.SetVariant(CLng(27))        ' Marshal as variant of type VT_I4.
mo.SetVariant(CSng(27.0))      ' Marshal as variant of type VT_R4.
mo.SetVariant(CDbl(27.0))      ' Marshal as variant of type VT_R8.
```

```
MarshalObject mo = new MarshalObject();
mo.SetVariant(null);            // Marshal as variant of type VT_EMPTY.
mo.SetVariant(System.DBNull.Value); // Marshal as variant of type VT_NULL.
mo.SetVariant((int)27);          // Marshal as variant of type VT_I2.
mo.SetVariant((long)27);          // Marshal as variant of type VT_I4.
mo.SetVariant((single)27.0);   // Marshal as variant of type VT_R4.
mo.SetVariant((double)27.0);   // Marshal as variant of type VT_R8.
```

COM types that do not have corresponding managed types can be marshaled using wrapper classes such as ErrorWrapper, DispatchWrapper, UnknownWrapper, and CurrencyWrapper. The following code example demonstrates how to use these wrappers to pass various types of variants to a COM server.

```
Imports System.Runtime.InteropServices
' Pass inew as a variant of type VT_UNKNOWN interface.
mo.SetVariant(New UnknownWrapper(inew))
' Pass inew as a variant of type VT_DISPATCH interface.
mo.SetVariant(New DispatchWrapper(inew))
' Pass a value as a variant of type VT_ERROR interface.
mo.SetVariant(New ErrorWrapper(&H80054002))
' Pass a value as a variant of type VT_CURRENCY interface.
mo.SetVariant(New CurrencyWrapper(New Decimal(5.25)))
```

```
using System.Runtime.InteropServices;
// Pass inew as a variant of type VT_UNKNOWN interface.
mo.SetVariant(new UnknownWrapper(inew));
// Pass inew as a variant of type VT_DISPATCH interface.
mo.SetVariant(new DispatchWrapper(inew));
// Pass a value as a variant of type VT_ERROR interface.
mo.SetVariant(new ErrorWrapper(0x80054002));
// Pass a value as a variant of type VT_CURRENCY interface.
mo.SetVariant(new CurrencyWrapper(new Decimal(5.25)));
```

The wrapper classes are defined in the System.Runtime.InteropServices namespace.

Marshaling the IConvertible Interface to Variant

Types other than those listed in the previous section can control how they are marshaled by implementing the IConvertible interface. If the object implements the **IConvertible** interface, the COM variant type is determined at run time by the value of the TypeCode enumeration returned from the IConvertible.GetTypeCode method.

The following table shows the possible values for the **TypeCode** enumeration and the corresponding COM variant type for each value.

TYPECODE	COM VARIANT TYPE
TypeCode.Empty	**VT_EMPTY**
TypeCode.Object	**VT_UNKNOWN**
TypeCode.DBNull	**VT_NULL**
TypeCode.Boolean	**VT_BOOL**
TypeCode.Char	**VT_UI2**
TypeCode.Sbyte	**VT_I1**
TypeCode.Byte	**VT_UI1**
TypeCode.Int16	**VT_I2**
TypeCode.UInt16	**VT_UI2**
TypeCode.Int32	**VT_I4**
TypeCode.UInt32	**VT_UI4**
TypeCode.Int64	**VT_I8**
TypeCode.UInt64	**VT_UI8**
TypeCode.Single	**VT_R4**
TypeCode.Double	**VT_R8**
TypeCode.Decimal	**VT_DECIMAL**

TYPECODE	COM VARIANT TYPE
TypeCode.DateTime	**VT_DATE**
TypeCode.String	**VT_BSTR**
Not supported.	**VT_INT**
Not supported.	**VT_UINT**
Not supported.	**VT_ARRAY**
Not supported.	**VT_RECORD**
Not supported.	**VT_CY**
Not supported.	**VT_VARIANT**

The value of the COM variant is determined by calling the **IConvertible.To** *Type* interface, where **To** *Type* is the conversion routine that corresponds to the type that was returned from **IConvertible.GetTypeCode**. For example, an object that returns **TypeCode.Double** from **IConvertible.GetTypeCode** is marshaled as a COM variant of type **VT_R8**. You can obtain the value of the variant (stored in the **dblVal** field of the COM variant) by casting to the **IConvertible** interface and calling the ToDouble method.

Marshaling Variant to Object

When marshaling a variant to an object, the type, and sometimes the value, of the marshaled variant determines the type of object produced. The following table identifies each variant type and the corresponding object type that the marshaler creates when a variant is passed from COM to the .NET Framework.

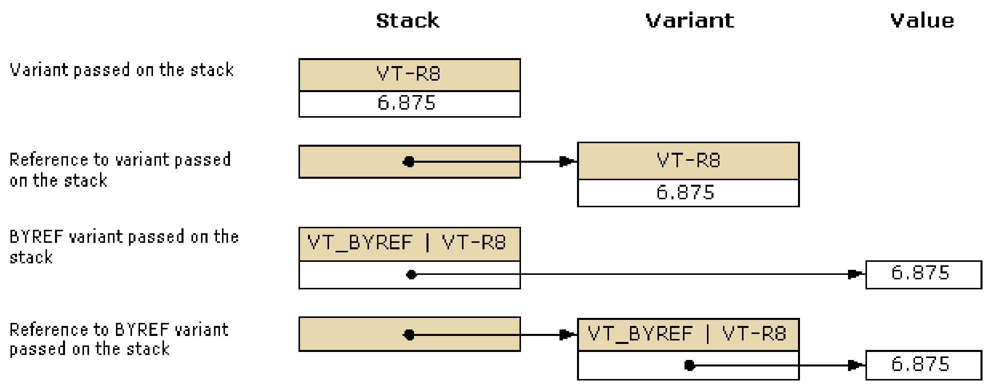
COM VARIANT TYPE	OBJECT TYPE
VT_EMPTY	Null object reference (**Nothing** in Visual Basic).
VT_NULL	System.DBNull
VT_DISPATCH	**System.__ComObject** or null if (pdispVal == null)
VT_UNKNOWN	**System.__ComObject** or null if (punkVal == null)
VT_ERROR	System.UInt32
VT_BOOL	System.Boolean
VT_I1	System.SByte
VT_UI1	System.Byte
VT_I2	System.Int16
VT_UI2	System.UInt16

COM VARIANT TYPE	OBJECT TYPE
VT_I4	System.Int32
VT_UI4	System.UInt32
VT_I8	System.Int64
VT_UI8	System.UInt64
VT_R4	System.Single
VT_R8	System.Double
VT_DECIMAL	System.Decimal
VT_DATE	System.DateTime
VT_BSTR	System.String
VT_INT	System.Int32
VT_UINT	System.UInt32
VT_ARRAY | VT_*	System.Array
VT_CY	System.Decimal
VT_RECORD	Corresponding boxed value type.
VT_VARIANT	Not supported.

Variant types passed from COM to managed code and then back to COM might not retain the same variant type for the duration of the call. Consider what happens when a variant of type **VT_DISPATCH** is passed from COM to the .NET Framework. During marshaling, the variant is converted to a System.Object. If the **Object** is then passed back to COM, it is marshaled back to a variant of type **VT_UNKNOWN**. There is no guarantee that the variant produced when an object is marshaled from managed code to COM will be the same type as the variant initially used to produce the object.

Marshaling ByRef Variants

Although variants themselves can be passed by value or by reference, the **VT_BYREF** flag can also be used with any variant type to indicate that the contents of the variant are being passed by reference instead of by value. The difference between marshaling variants by reference and marshaling a variant with the **VT_BYREF** flag set can be confusing. The following illustration clarifies the differences:


Stack
Variant
Value
Variant passed on the stack
VT-R8
6.875
Reference to variant passed on the stack
VT-R8
6.875
BYREF variant passed on the stack
VT_BYREF | VT-R8
6.875
Reference to BYREF variant passed on the stack
VT_BYREF | VT-R8
6.875

Variants passed by value and by reference

Default behavior for marshaling objects and variants by value

- When passing objects from managed code to COM, the contents of the object are copied into a new variant created by the marshaler, using the rules defined in Marshaling Object to Variant. Changes made to the variant on the unmanaged side are not propagated back to the original object on return from the call.
- When passing variants from COM to managed code, the contents of the variant are copied to a newly created object, using the rules defined in Marshaling Variant to Object. Changes made to the object on the managed side are not propagated back to the original variant on return from the call.

Default behavior for marshaling objects and variants by reference

To propagate changes back to the caller, the parameters must be passed by reference. For example, you can use the **ref** keyword in C# (or **ByRef** in Visual Basic managed code) to pass parameters by reference. In COM, reference parameters are passed using a pointer such as a **variant ***.

- When passing an object to COM by reference, the marshaler creates a new variant and copies the contents of the object reference into the variant before the call is made. The variant is passed to the unmanaged function where the user is free to change the contents of the variant. On return from the call, any changes made to the variant on the unmanaged side are propagated back to the original object. If the type of the variant differs from the type of the variant passed to the call, then the changes are propagated back to an object of a different type. That is, the type of the object passed into the call can differ from the type of the object returned from the call.
- When passing a variant to managed code by reference, the marshaler creates a new object and copies the contents of the variant into the object before making the call. A reference to the object is passed to the managed function, where the user is free to change the object. On return from the call, any changes made to the referenced object are propagated back to the original variant. If the type of the object differs from the type of the object passed in to the call, the type of the original variant is changed and the value is propagated back into the variant. Again, the type of the variant passed into the call can differ from the type of the variant returned from the call.

Default behavior for marshaling a variant with the VT_BYREF flag set

- A variant being passed to managed code by value can have the **VT_BYREF** flag set to indicate that the variant contains a reference instead of a value. In this case, the variant is still marshaled to an object because the variant is being passed by value. The marshaler automatically dereferences the contents of the variant and copies it into a newly created object before making the call. The object is then passed into the managed function; however, on return from the call, the object is not propagated back into the original variant. Changes made to the managed object are lost.

  Caution

  There is no way to change the value of a variant passed by value, even if the variant has the **VT_BYREF** flag set.

- A variant being passed to managed code by reference can also have the **VT_BYREF** flag set to indicate that the variant contains another reference. If it does, the variant is marshaled to a **ref** object because the variant is being passed by reference. The marshaler automatically dereferences the contents of the variant and copies it into a newly created object before making the call. On return from the call, the value of the object is propagated back to the reference within the original variant only if the object is the same type as the object passed in. That is, propagation does not change the type of a variant with the **VT_BYREF** flag set. If the type of the object is changed during the call, an InvalidCastException occurs on return from the call.

The following table summarizes the propagation rules for variants and objects.

FROM	TO	CHANGES PROPAGATED BACK
Variant *v*	**Object** *o*	Never
Object *o*	**Variant** *v*	Never
Variant * *pv*	**Ref Object** *o*	Always
Ref object *o*	**Variant *** *pv*	Always
Variant *v* **(VT_BYREF | VT_*)**	**Object** *o*	Never
Variant *v* **(VT_BYREF | VT_)**	**Ref Object** *o*	Only if the type has not changed.

See also

- Default Marshaling Behavior
- Blittable and Non-Blittable Types
- Directional Attributes
- Copying and Pinning

Default Marshaling for Strings

5/8/2019 • 7 minutes to read • Edit Online

Both the System.String and System.Text.StringBuilder classes have similar marshaling behavior.

Strings are marshaled as a COM-style `BSTR` type or as a null-terminated string (a character array that ends with a null character). The characters within the string can be marshaled as Unicode (the default on Windows systems) or ANSI.

Strings Used in Interfaces

The following table shows the marshaling options for the string data type when marshaled as a method argument to unmanaged code. The MarshalAsAttribute attribute provides several UnmanagedType enumeration values to marshal strings to COM interfaces.

ENUMERATION TYPE	DESCRIPTION OF UNMANAGED FORMAT
`UnmanagedType.BStr` (default)	A COM-style `BSTR` with a prefixed length and Unicode characters.
`UnmanagedType.LPStr`	A pointer to a null-terminated array of ANSI characters.
`UnmanagedType.LPWStr`	A pointer to a null-terminated array of Unicode characters.

This table applies to String. For StringBuilder, the only options allowed are `UnmanagedType.LPStr` and `UnmanagedType.LPWStr`.

The following example shows strings declared in the `IStringWorker` interface.

```
public interface IStringWorker
{
    void PassString1(string s);
    void PassString2([MarshalAs(UnmanagedType.BStr)] string s);
    void PassString3([MarshalAs(UnmanagedType.LPStr)] string s);
    void PassString4([MarshalAs(UnmanagedType.LPWStr)] string s);
    void PassStringRef1(ref string s);
    void PassStringRef2([MarshalAs(UnmanagedType.BStr)] ref string s);
    void PassStringRef3([MarshalAs(UnmanagedType.LPStr)] ref string s);
    void PassStringRef4([MarshalAs(UnmanagedType.LPWStr)] ref string s);
}
```

```
Public Interface IStringWorker
    Sub PassString1(s As String)
    Sub PassString2(<MarshalAs(UnmanagedType.BStr)> s As String)
    Sub PassString3(<MarshalAs(UnmanagedType.LPStr)> s As String)
    Sub PassString4(<MarshalAs(UnmanagedType.LPWStr)> s As String)
    Sub PassStringRef1(ByRef s As String)
    Sub PassStringRef2(<MarshalAs(UnmanagedType.BStr)> ByRef s As String)
    Sub PassStringRef3(<MarshalAs(UnmanagedType.LPStr)> ByRef s As String)
    Sub PassStringRef4(<MarshalAs(UnmanagedType.LPWStr)> ByRef s As String)
End Interface
```

The following example shows the corresponding interface described in a type library.

```
interface IStringWorker : IDispatch
{
    HRESULT PassString1([in] BSTR s);
    HRESULT PassString2([in] BSTR s);
    HRESULT PassString3([in] LPStr s);
    HRESULT PassString4([in] LPWStr s);
    HRESULT PassStringRef1([in, out] BSTR *s);
    HRESULT PassStringRef2([in, out] BSTR *s);
    HRESULT PassStringRef3([in, out] LPStr *s);
    HRESULT PassStringRef4([in, out] LPWStr *s);
};
```

Strings Used in Platform Invoke

Platform invoke copies string arguments, converting from the .NET Framework format (Unicode) to the platform unmanaged format. Strings are immutable and are not copied back from unmanaged memory to managed memory when the call returns.

The following table lists the marshaling options for strings when marshaled as a method argument of a platform invoke call. The MarshalAsAttribute attribute provides several UnmanagedType enumeration values to marshal strings.

ENUMERATION TYPE	DESCRIPTION OF UNMANAGED FORMAT
`UnmanagedType.AnsiBStr`	A COM-style `BSTR` with a prefixed length and ANSI characters.
`UnmanagedType.BStr`	A COM-style `BSTR` with a prefixed length and Unicode characters.
`UnmanagedType.LPStr` (default)	A pointer to a null-terminated array of ANSI characters.
`UnmanagedType.LPTStr`	A pointer to a null-terminated array of platform-dependent characters.
`UnmanagedType.LPUTF8Str`	A pointer to a null-terminated array of UTF-8 encoded characters.
`UnmanagedType.LPWStr`	A pointer to a null-terminated array of Unicode characters.
`UnmanagedType.TBStr`	A COM-style `BSTR` with a prefixed length and platform-dependent characters.
`VBByRefStr`	A value that enables Visual Basic .NET to change a string in unmanaged code and have the results reflected in managed code. This value is supported only for platform invoke. This is the default value in Visual Basic for `ByVal` strings.

This table applies to String. For StringBuilder, the only options allowed are `LPStr`, `LPTStr`, and `LPWStr`.

The following type definition shows the correct use of `MarshalAsAttribute` for platform invoke calls.

```
class StringLibAPI
{
    [DllImport("StringLib.dll")]
    public static extern void PassLPStr([MarshalAs(UnmanagedType.LPStr)] string s);
    [DllImport("StringLib.dll")]
    public static extern void PassLPWStr([MarshalAs(UnmanagedType.LPWStr)] string s);
    [DllImport("StringLib.dll")]
    public static extern void PassLPTStr([MarshalAs(UnmanagedType.LPTStr)] string s);
    [DllImport("StringLib.dll")]
    public static extern void PassLPUTF8Str([MarshalAs(UnmanagedType.LPUTF8Str)] string s);
    [DllImport("StringLib.dll")]
    public static extern void PassBStr([MarshalAs(UnmanagedType.BStr)] string s);
    [DllImport("StringLib.dll")]
    public static extern void PassAnsiBStr([MarshalAs(UnmanagedType.AnsiBStr)] string s);
    [DllImport("StringLib.dll")]
    public static extern void PassTBStr([MarshalAs(UnmanagedType.TBStr)] string s);
}
```

```
Class StringLibAPI
    Public Declare Auto Sub PassLPStr Lib "StringLib.dll" (
        <MarshalAs(UnmanagedType.LPStr)> s As String)
    Public Declare Auto Sub PassLPWStr Lib "StringLib.dll" (
        <MarshalAs(UnmanagedType.LPWStr)> s As String)
    Public Declare Auto Sub PassLPTStr Lib "StringLib.dll" (
        <MarshalAs(UnmanagedType.LPTStr)> s As String)
    Public Declare Auto Sub PassLPUTF8Str Lib "StringLib.dll" (
        <MarshalAs(UnmanagedType.LPUTF8Str)> s As String)
    Public Declare Auto Sub PassBStr Lib "StringLib.dll" (
        <MarshalAs(UnmanagedType.BStr)> s As String)
    Public Declare Auto Sub PassAnsiBStr Lib "StringLib.dll" (
        <MarshalAs(UnmanagedType.AnsiBStr)> s As String)
    Public Declare Auto Sub PassTBStr Lib "StringLib.dll" (
        <MarshalAs(UnmanagedType.TBStr)> s As String)
End Class
```

Strings Used in Structures

Strings are valid members of structures; however, StringBuilder buffers are invalid in structures. The following table shows the marshaling options for the String data type when the type is marshaled as a field. The MarshalAsAttribute attribute provides several UnmanagedType enumeration values to marshal strings to a field.

ENUMERATION TYPE	DESCRIPTION OF UNMANAGED FORMAT
`UnmanagedType.BStr`	A COM-style `BSTR` with a prefixed length and Unicode characters.
`UnmanagedType.LPStr` (default)	A pointer to a null-terminated array of ANSI characters.
`UnmanagedType.LPTStr`	A pointer to a null-terminated array of platform-dependent characters.
`UnmanagedType.LPUTF8Str`	A pointer to a null-terminated array of UTF-8 encoded characters.
`UnmanagedType.LPWStr`	A pointer to a null-terminated array of Unicode characters.
`UnmanagedType.ByValTStr`	A fixed-length array of characters; the array's type is determined by the character set of the containing structure.

The `ByValTStr` type is used for inline, fixed-length character arrays that appear within a structure. Other types apply to string references contained within structures that contain pointers to strings.

The `CharSet` argument of the StructLayoutAttribute that is applied to the containing structure determines the character format of strings in structures. The following example structures contain string references and inline strings, as well as ANSI, Unicode, and platform-dependent characters. The representation of these structures in a type library is shown in the following C++ code:

```
struct StringInfoA
{
    char *  f1;
    char    f2[256];
};

struct StringInfoW
{
    WCHAR * f1;
    WCHAR   f2[256];
    BSTR    f3;
};

struct StringInfoT
{
    TCHAR * f1;
    TCHAR   f2[256];
};
```

The following example shows how to use the MarshalAsAttribute to define the same structure in different formats.

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
struct StringInfoA
{
    [MarshalAs(UnmanagedType.LPStr)] public string f1;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 256)] public string f2;
}

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
struct StringInfoW
{
    [MarshalAs(UnmanagedType.LPWStr)] public string f1;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 256)] public string f2;
    [MarshalAs(UnmanagedType.BStr)] public string f3;
}

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
struct StringInfoT
{
    [MarshalAs(UnmanagedType.LPTStr)] public string f1;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 256)] public string f2;
}
```

```
<StructLayout(LayoutKind.Sequential, CharSet := CharSet.Ansi)> _
Structure StringInfoA
    <MarshalAs(UnmanagedType.LPStr)> Public f1 As String
    <MarshalAs(UnmanagedType.ByValTStr, SizeConst := 256)> _
    Public f2 As String
End Structure

<StructLayout(LayoutKind.Sequential, CharSet := CharSet.Unicode)> _
Structure StringInfoW
    <MarshalAs(UnmanagedType.LPWStr)> Public f1 As String
    <MarshalAs(UnmanagedType.ByValTStr, SizeConst := 256)> _
    Public f2 As String
<MarshalAs(UnmanagedType.BStr)> Public f3 As String
End Structure

<StructLayout(LayoutKind.Sequential, CharSet := CharSet.Auto)> _
Structure StringInfoT
    <MarshalAs(UnmanagedType.LPTStr)> Public f1 As String
    <MarshalAs(UnmanagedType.ByValTStr, SizeConst := 256)> _
    Public f2 As String
End Structure
```

Fixed-Length String Buffers

In some circumstances, a fixed-length character buffer must be passed into unmanaged code to be manipulated. Simply passing a string does not work in this case because the callee cannot modify the contents of the passed buffer. Even if the string is passed by reference, there is no way to initialize the buffer to a given size.

The solution is to pass a StringBuilder buffer as the argument instead of a String. A `StringBuilder` can be dereferenced and modified by the callee, provided it does not exceed the capacity of the `StringBuilder`. It can also be initialized to a fixed length. For example, if you initialize a `StringBuilder` buffer to a capacity of `N`, the marshaler provides a buffer of size (`N`+1) characters. The +1 accounts for the fact that the unmanaged string has a null terminator while `StringBuilder` does not.

For example, the Windows `GetWindowText` API function (defined in *winuser.h*) requires that the caller pass a fixed-length character buffer to which the function writes the window's text. `LpString` points to a caller-allocated buffer of size `nMaxCount`. The caller is expected to allocate the buffer and set the `nMaxCount` argument to the size of the allocated buffer. The following example shows the `GetWindowText` function declaration as defined in *winuser.h*.

```
int GetWindowText(
    HWND hWnd,        // Handle to window or control.
    LPTStr lpString,  // Text buffer.
    int nMaxCount     // Maximum number of characters to copy.
);
```

A `StringBuilder` can be dereferenced and modified by the callee, provided it does not exceed the capacity of the `StringBuilder`. The following code example demonstrates how `StringBuilder` can be initialized to a fixed length.

```
using System;
using System.Runtime.InteropServices;
using System.Text;

internal static class NativeMethods
{
    [DllImport("User32.dll")]
    internal static extern void GetWindowText(IntPtr hWnd, StringBuilder lpString, int nMaxCount);
}

public class Window
{
    internal IntPtr h;        // Internal handle to Window.
    public String GetText()
    {
        StringBuilder sb = new StringBuilder(256);
        NativeMethods.GetWindowText(h, sb, sb.Capacity + 1);
        return sb.ToString();
    }
}
```

```
Imports System.Text

Friend Class NativeMethods
    Friend Declare Auto Sub GetWindowText Lib "User32.dll" _
        (hWnd As IntPtr, lpString As StringBuilder, nMaxCount As Integer)
End Class

Public Class Window
    Friend h As IntPtr ' Friend handle to Window.
    Public Function GetText() As String
        Dim sb As New StringBuilder(256)
        NativeMethods.GetWindowText(h, sb, sb.Capacity + 1)
        Return sb.ToString()
   End Function
End Class
```

See also

- Default Marshaling Behavior
- Marshaling Strings
- Blittable and Non-Blittable Types
- Directional Attributes
- Copying and Pinning

Marshaling Data with Platform Invoke

3/25/2019 • 8 minutes to read • Edit Online

To call functions exported from an unmanaged library, a .NET Framework application requires a function prototype in managed code that represents the unmanaged function. To create a prototype that enables platform invoke to marshal data correctly, you must do the following:

- Apply the DllImportAttribute attribute to the static function or method in managed code.
- Substitute managed data types for unmanaged data types.

You can use the documentation supplied with an unmanaged function to construct an equivalent managed prototype by applying the attribute with its optional fields and substituting managed data types for unmanaged types. For instructions about how to apply the DllImportAttribute, see Consuming Unmanaged DLL Functions.

This section provides samples that demonstrate how to create managed function prototypes for passing arguments to and receiving return values from functions exported by unmanaged libraries. The samples also demonstrate when to use the MarshalAsAttribute attribute and the Marshal class to explicitly marshal data.

Platform invoke data types

The following table lists data types used in the Windows APIs and C-style functions. Many unmanaged libraries contain functions that pass these data types as parameters and return values. The third column lists the corresponding .NET Framework built-in value type or class that you use in managed code. In some cases, you can substitute a type of the same size for the type listed in the table.

UNMANAGED TYPE IN WINDOWS APIS	UNMANAGED C LANGUAGE TYPE	MANAGED TYPE	DESCRIPTION
`VOID`	`void`	System.Void	Applied to a function that does not return a value.
`HANDLE`	`void *`	System.IntPtr or System.UIntPtr	32 bits on 32-bit Windows operating systems, 64 bits on 64-bit Windows operating systems.
`BYTE`	`unsigned char`	System.Byte	8 bits
`SHORT`	`short`	System.Int16	16 bits
`WORD`	`unsigned short`	System.UInt16	16 bits
`INT`	`int`	System.Int32	32 bits
`UINT`	`unsigned int`	System.UInt32	32 bits
`LONG`	`long`	System.Int32	32 bits
`BOOL`	`long`	System.Boolean or System.Int32	32 bits

UNMANAGED TYPE IN WINDOWS APIS	UNMANAGED C LANGUAGE TYPE	MANAGED TYPE	DESCRIPTION
`DWORD`	`unsigned long`	System.UInt32	32 bits
`ULONG`	`unsigned long`	System.UInt32	32 bits
`CHAR`	`char`	System.Char	Decorate with ANSI.
`WCHAR`	`wchar_t`	System.Char	Decorate with Unicode.
`LPSTR`	`char *`	System.String or System.Text.StringBuilder	Decorate with ANSI.
`LPCSTR`	`const char *`	System.String or System.Text.StringBuilder	Decorate with ANSI.
`LPWSTR`	`wchar_t *`	System.String or System.Text.StringBuilder	Decorate with Unicode.
`LPCWSTR`	`const wchar_t *`	System.String or System.Text.StringBuilder	Decorate with Unicode.
`FLOAT`	`float`	System.Single	32 bits
`DOUBLE`	`double`	System.Double	64 bits

For corresponding types in Visual Basic, C#, and C++, see the Introduction to the .NET Framework Class Library.

PinvokeLib.dll

The following code defines the library functions provided by Pinvoke.dll. Many samples described in this section call this library.

Example

```
// PInvokeLib.cpp : Defines the entry point for the DLL application.
//

#define PINVOKELIB_EXPORTS
#include "PInvokeLib.h"

#include <strsafe.h>
#include <objbase.h>
#include <stdio.h>

#pragma comment(lib,"ole32.lib")

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved )
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }

    return TRUE;
}

//******************************************************************
// This is the constructor of a class that has been exported.
CTestClass::CTestClass()
{
    m_member = 1;
}

int CTestClass::DoSomething( int i )
{
    return i*i + m_member;
}

PINVOKELIB_API CTestClass* CreateTestClass()
{
    return new CTestClass();
}

PINVOKELIB_API void DeleteTestClass( CTestClass* instance )
{
    delete instance;
}

//******************************************************************
PINVOKELIB_API int TestArrayOfInts( int* pArray, int size )
{
    int result = 0;

    for ( int i = 0; i < size; i++ )
    {
        result += pArray[ i ];
        pArray[i] += 100;
    }
    return result;
}

//******************************************************************
PINVOKELIB_API int TestRefArrayOfInts( int** ppArray, int* pSize )
{
    int result = 0;

    // CoTaskMemAlloc must be used instead of the new operator
    // because code on the managed side will call Marshal.FreeCoTaskMem
    // to free this memory.

    int* newArray = (int*)CoTaskMemAlloc( sizeof(int) * 5 );

    for ( int i = 0; i < *pSize; i++ )
    {
        result += (*ppArray)[i];
    }

    for ( int j = 0; j < 5; j++ )
    {
        newArray[j] = (*ppArray)[j] + 100;
    }

    CoTaskMemFree( *ppArray );
    *ppArray = newArray;
    *pSize = 5;

    return result;
}
```

```
//******************************************************************
PINVOKELIB_API int TestMatrixOfInts( int pMatrix[][COL_DIM], int row )
{
    int result = 0;

    for ( int i = 0; i < row; i++ )
    {
        for ( int j = 0; j < COL_DIM; j++ )
        {
            result += pMatrix[i][j];
            pMatrix[i][j] += 100;
        }
    }
    return result;
}

//******************************************************************
PINVOKELIB_API int TestArrayOfStrings( char* ppStrArray[], int count )
{
    int result = 0;
    STRSAFE_LPSTR temp;
    size_t len;
    const size_t alloc_size = sizeof(char) * 10;

    for ( int i = 0; i < count; i++ )
    {
        len = 0;
        StringCchLengthA( ppStrArray[i], STRSAFE_MAX_CCH, &len );
        result += len;

        temp = (STRSAFE_LPSTR)CoTaskMemAlloc( alloc_size );
        StringCchCopyA( temp, alloc_size, (STRSAFE_LPCSTR)"123456789" );

       // CoTaskMemFree must be used instead of delete to free memory.

       CoTaskMemFree( ppStrArray[i] );
       ppStrArray[i] = (char *) temp;
   }

   return result;
}

//******************************************************************
PINVOKELIB_API int TestArrayOfStructs( MYPOINT* pPointArray, int size )
{
    int result = 0;
    MYPOINT* pCur = pPointArray;

    for ( int i = 0; i < size; i++ )
    {
        result += pCur->x + pCur->y;
        pCur->y = 0;
        pCur++;
    }

    return result;
}

//******************************************************************
PINVOKELIB_API int TestStructInStruct( MYPERSON2* pPerson2 )
{
    size_t len = 0;

    StringCchLengthA( pPerson2->person->last, STRSAFE_MAX_CCH, &len );
    len = sizeof(char) * ( len + 2 ) + 1;

    STRSAFE_LPSTR temp = (STRSAFE_LPSTR)CoTaskMemAlloc( len );
    StringCchCopyA( temp, len, (STRSAFE_LPSTR)"Mc" );
    StringCbCatA( temp, len, (STRSAFE_LPSTR)pPerson2->person->last );
    StringCbCatA( temp, len, (STRSAFE_LPSTR)pPerson2->person->last );

    CoTaskMemFree( pPerson2->person->last );
    pPerson2->person->last = (char *)temp;

    return pPerson2->age;
}

//******************************************************************
PINVOKELIB_API int TestArrayOfStructs2( MYPERSON* pPersonArray, int size )
{
    int result = 0;
    MYPERSON* pCur = pPersonArray;
    STRSAFE_LPSTR temp;
    size_t len;

    for ( int i = 0; i < size; i++ )
    {
        len = 0;
        StringCchLengthA( pCur->first, STRSAFE_MAX_CCH, &len );
        len++;
        result += len;
        len = 0;
        StringCchLengthA( pCur->last, STRSAFE_MAX_CCH, &len );
        len++;
        result += len;

        len = sizeof(char) * ( len + 2 );
        temp = (STRSAFE_LPSTR)CoTaskMemAlloc( len );
        StringCchCopyA( temp, len, (STRSAFE_LPCSTR)"Mc" );
        StringCbCatA( temp, len, (STRSAFE_LPCSTR)pCur->last );
        result += 2;

        // CoTaskMemFree must be used instead of delete to free memory.
        CoTaskMemFree( pCur->last );
        pCur->last = (char *)temp;
        pCur++;
   }

   return result;
}

//******************************************************************
PINVOKELIB_API void TestStructInStruct3( MYPERSON3 person3 )
{
    printf( "\n\nperson passed by value:\n" );
    printf( "first = %s last = %s age = %i\n\n",
            person3.person.first,
            person3.person.last,
            person3.age );
}

//*********************************************************************
PINVOKELIB_API void TestUnion( MYUNION u, int type )
{
    if ( ( type != 1 ) && ( type != 2 ) )
    {
        return;
    }
    if ( type == 1 )
    {
        printf( "\n\ninteger passed: %i", u.i );
    }
    else if ( type == 2 )
    {
        printf( "\n\ndouble passed: %f", u.d );
    }
}

//******************************************************************
```

```
//******************************************************************
PINVOKELIB_API void TestUnion2( MYUNION2 u, int type )
{
    if ( ( type != 1 ) && ( type != 2 ) )
    {
        return;
    }
    if ( type == 1 )
    {
        printf( "\n\ninteger passed: %i", u.i );
    }
    else if ( type == 2 )
    {
        printf( "\n\nstring passed: %s", u.str );
    }
}

//******************************************************************
PINVOKELIB_API void TestCallBack( FPTR pf, int value )
{
    printf( "\nReceived value: %i", value );
    printf( "\nPassing to callback..." );
    bool res = (*pf)(value);

    if ( res )
    {
        printf( "Callback returned true.\n" );
    }
    else
    {
        printf( "Callback returned false.\n" );
    }
}

//******************************************************************
PINVOKELIB_API void TestCallBack2( FPTR2 pf2, char* value )
{
    printf( "\nReceived value: %s", value );
    printf( "\nPassing to callback..." );
    bool res = (*pf2)(value);

    if ( res )
    {
        printf( "Callback2 returned true.\n" );
    }
    else
    {
        printf( "Callback2 returned false.\n" );
    }
}

//******************************************************************
PINVOKELIB_API void TestStringInStruct( MYSTRSTRUCT* pStruct )
{
    wprintf( L"\nUnicode buffer content: %s\n", pStruct->buffer );

    // Assuming that the buffer is big enough.
    StringCbCatW( pStruct->buffer, pStruct->size, (STRSAFE_LPWSTR)L"++" );
}

//******************************************************************
PINVOKELIB_API void TestStringInStructAnsi( MYSTRSTRUCT2* pStruct )
{
    printf( "\nAnsi buffer content: %s\n", pStruct->buffer );

    // Assuming that the buffer is big enough.
    StringCbCatA( (STRSAFE_LPSTR) pStruct->buffer, pStruct->size, (STRSAFE_LPSTR)"++" );
}
```

```
//******************************************************************
PINVOKELIB_API void TestOutArrayOfStructs( int* pSize, MYSTRSTRUCT2** ppStruct )
{
    const int cArraySize = 5;
    *pSize = 0;
    *ppStruct = (MYSTRSTRUCT2*)CoTaskMemAlloc( cArraySize * sizeof( MYSTRSTRUCT2 ));

    if ( ppStruct != NULL )
    {
        MYSTRSTRUCT2* pCurStruct = *ppStruct;
        LPSTR buffer;
        *pSize = cArraySize;

        STRSAFE_LPCSTR teststr = "***";
        size_t len = 0;
        StringCchLengthA(teststr, STRSAFE_MAX_CCH, &len);
        len++;

        for ( int i = 0; i < cArraySize; i++, pCurStruct++ )
        {
            pCurStruct->size = len;
            buffer = (LPSTR)CoTaskMemAlloc( len );
            StringCchCopyA( buffer, len, teststr );
            pCurStruct->buffer = (char *)buffer;
        }
    }
}

//************************************************************************
PINVOKELIB_API char * TestStringAsResult()
{

    const size_t alloc_size = 64;
    STRSAFE_LPSTR result = (STRSAFE_LPSTR)CoTaskMemAlloc( alloc_size );
    STRSAFE_LPCSTR teststr = "This is return value";
    StringCchCopyA( result, alloc_size, teststr );

    return (char *) result;
}

//************************************************************************
PINVOKELIB_API void SetData( DataType typ, void* object )
{
    switch ( typ )
    {
        case DT_I2: printf( "Short %i\n", *((short*)object) ); break;
        case DT_I4: printf( "Long %i\n", *((long*)object) ); break;
        case DT_R4: printf( "Float %f\n", *((float*)object) ); break;
        case DT_R8: printf( "Double %f\n", *((double*)object) ); break;
        case DT_STR: printf( "String %s\n", (char*)object ); break;
        default: printf( "Unknown type" ); break;
    }
}

//************************************************************************
PINVOKELIB_API void TestArrayInStruct( MYARRAYSTRUCT* pStruct )
{
    pStruct->flag = true;
    pStruct->vals[0] += 100;
    pStruct->vals[1] += 100;
    pStruct->vals[2] += 100;
}
```

```
// PInvokeLib.h : The header file for the DLL application.
//

#pragma once
```

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>

// The following ifdef block is the standard way of creating macros which make exporting
// from a DLL simpler. All files within this DLL are compiled with the PINVOKELIB_EXPORTS
// symbol defined on the command line. this symbol should not be defined on any project
// that uses this DLL. This way any other project whose source files include this file see
// PINVOKELIB_API functions as being imported from a DLL, wheras this DLL sees symbols
// defined with this macro as being exported.
#ifdef PINVOKELIB_EXPORTS
#define PINVOKELIB_API __declspec(dllexport)
#else
#define PINVOKELIB_API __declspec(dllimport)
#endif

// Define the test structures

typedef struct _MYPOINT
{
    int x;
    int y;
} MYPOINT;

typedef struct _MYPERSON
{
    char* first;
    char* last;
} MYPERSON;

typedef struct _MYPERSON2
{
    MYPERSON* person;
    int age;
} MYPERSON2;

typedef struct _MYPERSON3
{
    MYPERSON person;
    int age;
} MYPERSON3;

union MYUNION
{
    int i;
    double d;
};

union MYUNION2
{
    int i;
    char str[128];
};

typedef struct _MYSTRSTRUCT
{
    wchar_t* buffer;
    UINT size;
} MYSTRSTRUCT;

typedef struct _MYSTRSTRUCT2
{
    char* buffer;
    UINT size;
} MYSTRSTRUCT2;

typedef struct _MYARRAYSTRUCT
{
    bool flag;
    int vals[3];
} MYARRAYSTRUCT;

// constants and pointer definitions

const int COL_DIM = 5;

typedef bool (CALLBACK *FPTR)( int i );

typedef bool (CALLBACK *FPTR2)( char* str );

// Data type codes
enum DataType
{
    DT_I2 = 1,
    DT_I4,
    DT_R4,
    DT_R8,
    DT_STR
};

// This is an exported class.
class PINVOKELIB_API CTestClass
{
public:
    CTestClass( void );
    int DoSomething( int i );

private:
    int m_member;
};

// Exports for PInvokeLib.dll

#ifdef __cplusplus
extern "C"
{
#endif

PINVOKELIB_API CTestClass* CreateTestClass();

PINVOKELIB_API void DeleteTestClass( CTestClass* instance );

PINVOKELIB_API int TestArrayOfInts( int* pArray, int size );

PINVOKELIB_API int TestRefArrayOfInts( int** ppArray, int* pSize );

PINVOKELIB_API int TestMatrixOfInts( int pMatrix[][COL_DIM], int row );

PINVOKELIB_API int TestArrayOfStrings( char* ppStrArray[], int size );

PINVOKELIB_API int TestArrayOfStructs( MYPOINT* pPointArray, int size );

PINVOKELIB_API int TestArrayOfStructs2( MYPERSON* pPersonArray, int size );

PINVOKELIB_API int TestStructInStruct( MYPERSON2* pPerson2 );

PINVOKELIB_API void TestStructInStruct3( MYPERSON3 person3 );

PINVOKELIB_API void TestUnion( MYUNION u, int type );

PINVOKELIB_API void TestUnion2( MYUNION2 u, int type );

PINVOKELIB_API void TestCallBack( FPTR pf, int value );

PINVOKELIB_API void TestCallBack2( FPTR2 pf2, char* value );

// buffer is an in/out param
PINVOKELIB_API void TestStringInStruct( MYSTRSTRUCT* pStruct );
```

```
// buffer is in/out param
PINVOKELIB_API void TestStringInStructAnsi( MYSTRSTRUCT2* pStruct );

PINVOKELIB_API void TestOutArrayOfStructs( int* pSize, MYSTRSTRUCT2** ppStruct );

PINVOKELIB_API char* TestStringAsResult();

PINVOKELIB_API void SetData( DataType typ, void* object );

PINVOKELIB_API void TestArrayInStruct( MYARRAYSTRUCT* pStruct );

#ifdef __cplusplus
}
#endif
```

Marshaling Strings

4/19/2019 • 2 minutes to read • Edit Online

Platform invoke copies string parameters, converting them from the .NET Framework format (Unicode) to the unmanaged format (ANSI), if needed. Because managed strings are immutable, platform invoke does not copy them back from unmanaged memory to managed memory when the function returns.

The following table lists marshaling options for strings, describes their usage, and provides a link to the corresponding .NET Framework sample.

STRING	DESCRIPTION	SAMPLE
By value.	Passes strings as In parameters.	MsgBox
As result.	Returns strings from unmanaged code.	Strings
By reference.	Passes strings as In/Out parameters using StringBuilder.	Buffers
In a structure by value.	Passes strings in a structure that is an In parameter.	Structs
In a structure by reference **(char*)**.	Passes strings in a structure that is an In/Out parameter. The unmanaged function expects a pointer to a character buffer and the buffer size is a member of the structure.	Strings
In a structure by reference **(char[])**.	Passes strings in a structure that is an In/Out parameter. The unmanaged function expects an embedded character buffer.	OSInfo
In a class by value **(char*)**.	Passes strings in a class (a class is an In/Out parameter). The unmanaged function expects a pointer to a character buffer.	OpenFileDlg
In a class by value **(char[])**.	Passes strings in a class (a class is an In/Out parameter). The unmanaged function expects an embedded character buffer.	OSInfo
As an array of strings by value.	Creates an array of strings that is passed by value.	Arrays
As an array of structures that contain strings by value.	Creates an array of structures that contain strings and the array is passed by value.	Arrays

See also

- Default Marshaling for Strings

- Marshaling Data with Platform Invoke
- Marshaling Classes, Structures, and Unions
- Marshaling Different Types of Arrays
- Miscellaneous Marshaling Samples

MsgBox Sample

4/28/2019 • 2 minutes to read • Edit Online

This sample demonstrates how to pass string types by value as In parameters and when to use the EntryPoint, CharSet, and ExactSpelling fields.

The MsgBox sample uses the following unmanaged function, shown with its original function declaration:

- **MessageBox** exported from User32.dll.

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption,
   UINT uType);
```

In this sample, the `LibWrap` class contains a managed prototype for each unmanaged function called by the `MsgBoxSample` class. The managed prototype methods `MsgBox` , `MsgBox2` , and `MsgBox3` have different declarations for the same unmanaged function.

The declaration for `MsgBox2` produces incorrect output in the message box because the character type, specified as ANSI, is mismatched with the entry point `MessageBoxW` , which is the name of the Unicode function. The declaration for `MsgBox3` creates a mismatch between the **EntryPoint**, **CharSet**, and **ExactSpelling** fields. When called, `MsgBox3` throws an exception. For detailed information on string naming and name marshaling, see Specifying a Character Set.

Declaring Prototypes

```
public ref class LibWrap
{
public:
    // Declares managed prototypes for unmanaged functions.
    [DllImport( "User32.dll", EntryPoint="MessageBox",
        CharSet=CharSet::Auto)]
    static int MsgBox(int hWnd, String^ text, String^ caption,
      unsigned int type);

    // Causes incorrect output in the message window.
    [DllImport( "User32.dll", EntryPoint="MessageBoxW",
        CharSet=CharSet::Ansi )]
    static int MsgBox2(int hWnd, String^ text,
       String^ caption, unsigned int type);

    // Causes an exception to be thrown. EntryPoint, CharSet, and
    // ExactSpelling fields are mismatched.
    [DllImport( "User32.dll", EntryPoint="MessageBox",
        CharSet=CharSet::Ansi, ExactSpelling=true )]
    static int MsgBox3(int hWnd, String^ text,
        String^ caption, unsigned int type);
};
```

```
public class LibWrap
{
    // Declares managed prototypes for unmanaged functions.
    [DllImport("User32.dll", EntryPoint = "MessageBox",
        CharSet = CharSet.Auto)]
    public static extern int MsgBox(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);

    // Causes incorrect output in the message window.
    [DllImport("User32.dll", EntryPoint = "MessageBoxW",
        CharSet = CharSet.Ansi)]
    public static extern int MsgBox2(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);

    // Causes an exception to be thrown. EntryPoint, CharSet, and
    // ExactSpelling fields are mismatched.
    [DllImport("User32.dll", EntryPoint = "MessageBox",
        CharSet = CharSet.Ansi, ExactSpelling = true)]
    public static extern int MsgBox3(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);
}
```

```
Public Class LibWrap
    ' Declares managed prototypes for unmanaged functions.
    Declare Auto Function MsgBox Lib "User32.dll" Alias "MessageBox" (
        ByVal hWnd As IntPtr, ByVal lpText As String, ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer

    ' Causes incorrect output in the message window.
    Declare Ansi Function MsgBox2 Lib "User32.dll" Alias "MessageBoxW" (
        ByVal hWnd As IntPtr, ByVal lpText As String, ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer

    ' Causes an exception to be thrown.
    ' ExactSpelling is True by default when Ansi or Unicode is used.
    Declare Ansi Function MsgBox3 Lib "User32.dll" Alias "MessageBox" (
        ByVal hWnd As IntPtr, ByVal lpText As String, ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer
End Class
```

Calling Functions

```
public class MsgBoxSample
{
public:
    static void Main()
    {
        LibWrap::MsgBox(0, "Correct text", "MsgBox Sample", 0);
        LibWrap::MsgBox2(0, "Incorrect text", "MsgBox Sample", 0);

        try
        {
            LibWrap::MsgBox3(0, "No such function", "MsgBox Sample", 0);
        }
        catch (EntryPointNotFoundException^)
        {
           Console::WriteLine("EntryPointNotFoundException thrown as expected!");
        }
    }
};
```

```csharp
public class MsgBoxSample
{
    public static void Main()
    {
        LibWrap.MsgBox(0, "Correct text", "MsgBox Sample", 0);
        LibWrap.MsgBox2(0, "Incorrect text", "MsgBox Sample", 0);

        try
        {
            LibWrap.MsgBox3(0, "No such function", "MsgBox Sample", 0);
        }
        catch (EntryPointNotFoundException)
        {
            Console.WriteLine("EntryPointNotFoundException thrown as expected!");
        }
    }
}
```

```vb
Public Class MsgBoxSample
    Public Shared Sub Main()
        LibWrap.MsgBox(0, "Correct text", "MsgBox Sample", 0)
        LibWrap.MsgBox2(0, "Incorrect text", "MsgBox Sample", 0)

        Try
            LibWrap.MsgBox3(0, "No such function", "MsgBox Sample", 0)
        Catch e As EntryPointNotFoundException
            Console.WriteLine("EntryPointNotFoundException thrown as expected!")
        End Try
    End Sub
End Class
```

See also

- Marshaling Strings
- Default Marshaling for Strings
- Creating Prototypes in Managed Code
- Specifying a Character Set

Marshaling Classes, Structures, and Unions

5/9/2019 • 21 minutes to read • Edit Online

Classes and structures are similar in the .NET Framework. Both can have fields, properties, and events. They can also have static and nonstatic methods. One notable difference is that structures are value types and classes are reference types.

The following table lists marshaling options for classes, structures, and unions; describes their usage; and provides a link to the corresponding platform invoke sample.

TYPE	DESCRIPTION	SAMPLE
Class by value.	Passes a class with integer members as an In/Out parameter, like the managed case.	SysTime sample
Structure by value.	Passes structures as In parameters.	Structures sample
Structure by reference.	Passes structures as In/Out parameters.	OSInfo sample
Structure with nested structures (flattened).	Passes a class that represents a structure with nested structures in the unmanaged function. The structure is flattened to one big structure in the managed prototype.	FindFile sample
Structure with a pointer to another structure.	Passes a structure that contains a pointer to a second structure as a member.	Structures Sample
Array of structures with integers by value.	Passes an array of structures that contain only integers as an In/Out parameter. Members of the array can be changed.	Arrays Sample
Array of structures with integers and strings by reference.	Passes an array of structures that contain integers and strings as an Out parameter. The called function allocates memory for the array.	OutArrayOfStructs Sample
Unions with value types.	Passes unions with value types (integer and double).	Unions sample
Unions with mixed types.	Passes unions with mixed types (integer and string).	Unions sample
Null values in structure.	Passes a null reference (**Nothing** in Visual Basic) instead of a reference to a value type.	HandleRef sample

Structures sample

This sample demonstrates how to pass a structure that points to a second structure, pass a structure with an embedded structure, and pass a structure with an embedded array.

The Structs sample uses the following unmanaged functions, shown with their original function declaration:

- **TestStructInStruct** exported from PinvokeLib.dll.

  ```
  int TestStructInStruct(MYPERSON2* pPerson2);
  ```

- **TestStructInStruct3** exported from PinvokeLib.dll.

  ```
  void TestStructInStruct3(MYPERSON3 person3);
  ```

- **TestArrayInStruct** exported from PinvokeLib.dll.

  ```
  void TestArrayInStruct( MYARRAYSTRUCT* pStruct );
  ```

PinvokeLib.dll is a custom unmanaged library that contains implementations for the previously listed functions and four structures: **MYPERSON**, **MYPERSON2**, **MYPERSON3**, and **MYARRAYSTRUCT**. These structures contain the following elements:

```
typedef struct _MYPERSON
{
   char* first;
   char* last;
} MYPERSON, *LP_MYPERSON;

typedef struct _MYPERSON2
{
   MYPERSON* person;
   int age;
} MYPERSON2, *LP_MYPERSON2;

typedef struct _MYPERSON3
{
   MYPERSON person;
   int age;
} MYPERSON3;

typedef struct _MYARRAYSTRUCT
{
   bool flag;
   int vals[ 3 ];
} MYARRAYSTRUCT;
```

The managed `MyPerson`, `MyPerson2`, `MyPerson3`, and `MyArrayStruct` structures have the following characteristic:

- `MyPerson` contains only string members. The CharSet field sets the strings to ANSI format when passed to the unmanaged function.

- `MyPerson2` contains an **IntPtr** to the `MyPerson` structure. The **IntPtr** type replaces the original pointer to the unmanaged structure because .NET Framework applications do not use pointers unless the code is marked **unsafe**.

- `MyPerson3` contains `MyPerson` as an embedded structure. A structure embedded within another structure can be flattened by placing the elements of the embedded structure directly into the main structure, or it can be left as an embedded structure, as is done in this sample.

- `MyArrayStruct` contains an array of integers. The MarshalAsAttribute attribute sets the UnmanagedType enumeration value to **ByValArray**, which is used to indicate the number of elements in the array.

For all structures in this sample, the StructLayoutAttribute attribute is applied to ensure that the members are arranged in memory sequentially, in the order in which they appear.

The `LibWrap` class contains managed prototypes for the `TestStructInStruct`, `TestStructInStruct3`, and `TestArrayInStruct` methods called by the `App` class. Each prototype declares a single parameter, as follows:

- `TestStructInStruct` declares a reference to type `MyPerson2` as its parameter.
- `TestStructInStruct3` declares type `MyPerson3` as its parameter and passes the parameter by value.
- `TestArrayInStruct` declares a reference to type `MyArrayStruct` as its parameter.

Structures as arguments to methods are passed by value unless the parameter contains the **ref** (**ByRef** in Visual Basic) keyword. For example, the `TestStructInStruct` method passes a reference (the value of an address) to an object of type `MyPerson2` to unmanaged code. To manipulate the structure that `MyPerson2` points to, the sample creates a buffer of a specified size and returns its address by combining the Marshal.AllocCoTaskMem and Marshal.SizeOf methods. Next, the sample copies the content of the managed structure to the unmanaged buffer. Finally, the sample uses the Marshal.PtrToStructure method to marshal data from the unmanaged buffer to a managed object and the Marshal.FreeCoTaskMem method to free the unmanaged block of memory.

Declaring Prototypes

```
// Declares a managed structure for each unmanaged structure.
[StructLayout(LayoutKind::Sequential, CharSet=CharSet::Ansi)]
public value struct MyPerson
{
public:
    String^ first;
    String^ last;
};

[StructLayout(LayoutKind::Sequential)]
public value struct MyPerson2
{
public:
    IntPtr person;
    int age;
};

[StructLayout(LayoutKind::Sequential)]
public value struct MyPerson3
{
public:
    MyPerson person;
    int age;
};

[StructLayout(LayoutKind::Sequential)]
public value struct MyArrayStruct
{
public:
    bool flag;
    [MarshalAs(UnmanagedType::ByValArray, SizeConst=3)]
    array<int>^ vals;
};

public ref class LibWrap
{
public:
    // Declares a managed prototype for unmanaged function.
    [DllImport("..\\LIB\\PinvokeLib.dll")]
    static int TestStructInStruct(MyPerson2% person2);

    [DllImport("..\\LIB\\PinvokeLib.dll")]
    static int TestStructInStruct3(MyPerson3 person3);

    [DllImport("..\\LIB\\PinvokeLib.dll")]
    static int TestArrayInStruct(MyArrayStruct% myStruct);
};
```

```
// Declares a managed structure for each unmanaged structure.
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct MyPerson
{
    public string first;
    public string last;
}

[StructLayout(LayoutKind.Sequential)]
public struct MyPerson2
{
    public IntPtr person;
    public int age;
}

[StructLayout(LayoutKind.Sequential)]
public struct MyPerson3
{
    public MyPerson person;
    public int age;
}

[StructLayout(LayoutKind.Sequential)]
public struct MyArrayStruct
{
    public bool flag;

    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 3)]
    public int[] vals;
}

public class LibWrap
{
    // Declares a managed prototype for unmanaged function.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    public static extern int TestStructInStruct(ref MyPerson2 person2);

    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    public static extern int TestStructInStruct3(MyPerson3 person3);

    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    public static extern int TestArrayInStruct(ref MyArrayStruct myStruct);
}
```

```vb
' Declares a managed structure for each unmanaged structure.
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Ansi)>
Public Structure MyPerson
    Public first As String
    Public last As String
End Structure 'MyPerson

<StructLayout(LayoutKind.Sequential)>
Public Structure MyPerson2
    Public person As IntPtr
    Public age As Integer
End Structure 'MyPerson2

<StructLayout(LayoutKind.Sequential)>
Public Structure MyPerson3
    Public person As MyPerson
    Public age As Integer
End Structure 'MyPerson3

<StructLayout(LayoutKind.Sequential)>
Public Structure MyArrayStruct
    Public flag As Boolean
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=3)>
    Public vals As Integer()
End Structure 'MyArrayStruct

Public Class LibWrap
    ' Declares managed prototypes for unmanaged functions.
    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Function TestStructInStruct(
        ByRef person2 As MyPerson2) As Integer
    End Function

    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Function TestStructInStruct3(
        ByVal person3 As MyPerson3) As Integer
    End Function

    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Function TestArrayInStruct(
        ByRef myStruct As MyArrayStruct) As Integer
    End Function
End Class 'LibWrap
```

Calling Functions

```
public ref class App
{
public:
    static void Main()
    {
        // Structure with a pointer to another structure.
        MyPerson personName;
        personName.first = "Mark";
        personName.last = "Lee";

        MyPerson2 personAll;
        personAll.age = 30;

        IntPtr buffer = Marshal::AllocCoTaskMem(Marshal::SizeOf(personName));
        Marshal::StructureToPtr(personName, buffer, false);

        personAll.person = buffer;

        Console::WriteLine("\nPerson before call:");
        Console::WriteLine("first = {0}, last = {1}, age = {2}",
            personName.first, personName.last, personAll.age);

        int res = LibWrap::TestStructInStruct(personAll);

        MyPerson personRes =
            (MyPerson)Marshal::PtrToStructure(personAll.person,
            MyPerson::typeid);

        Marshal::FreeCoTaskMem(buffer);

        Console::WriteLine("Person after call:");
        Console::WriteLine("first = {0}, last = {1}, age = {2}",
            personRes.first, personRes.last, personAll.age);

        // Structure with an embedded structure.
        MyPerson3 person3;// = gcnew MyPerson3();
        person3.person.first = "John";
        person3.person.last = "Evans";
        person3.age = 27;
        LibWrap::TestStructInStruct3(person3);

        // Structure with an embedded array.
        MyArrayStruct myStruct;// = new MyArrayStruct();

        myStruct.flag = false;
        myStruct.vals = gcnew array<int>(3);
        myStruct.vals[0] = 1;
        myStruct.vals[1] = 4;
        myStruct.vals[2] = 9;

        Console::WriteLine("\nStructure with array before call:");
        Console::WriteLine(myStruct.flag);
        Console::WriteLine("{0} {1} {2}", myStruct.vals[0],
            myStruct.vals[1], myStruct.vals[2]);

        LibWrap::TestArrayInStruct(myStruct);
        Console::WriteLine("\nStructure with array after call:");
        Console::WriteLine(myStruct.flag);
        Console::WriteLine("{0} {1} {2}", myStruct.vals[0],
            myStruct.vals[1], myStruct.vals[2]);
    }
};
```

```
public class App
{
    public static void Main()
    {
        // Structure with a pointer to another structure.
        MyPerson personName;
        personName.first = "Mark";
        personName.last = "Lee";

        MyPerson2 personAll;
        personAll.age = 30;

        IntPtr buffer = Marshal.AllocCoTaskMem(Marshal.SizeOf(personName));
        Marshal.StructureToPtr(personName, buffer, false);

        personAll.person = buffer;

        Console.WriteLine("\nPerson before call:");
        Console.WriteLine("first = {0}, last = {1}, age = {2}",
            personName.first, personName.last, personAll.age);

        int res = LibWrap.TestStructInStruct(ref personAll);

        MyPerson personRes =
            (MyPerson)Marshal.PtrToStructure(personAll.person,
            typeof(MyPerson));

        Marshal.FreeCoTaskMem(buffer);

        Console.WriteLine("Person after call:");
        Console.WriteLine("first = {0}, last = {1}, age = {2}",
            personRes.first, personRes.last, personAll.age);

        // Structure with an embedded structure.
        MyPerson3 person3 = new MyPerson3();
        person3.person.first = "John";
        person3.person.last = "Evans";
        person3.age = 27;
        LibWrap.TestStructInStruct3(person3);

        // Structure with an embedded array.
        MyArrayStruct myStruct = new MyArrayStruct();

        myStruct.flag = false;
        myStruct.vals = new int[3];
        myStruct.vals[0] = 1;
        myStruct.vals[1] = 4;
        myStruct.vals[2] = 9;

        Console.WriteLine("\nStructure with array before call:");
        Console.WriteLine(myStruct.flag);
        Console.WriteLine("{0} {1} {2}", myStruct.vals[0],
            myStruct.vals[1], myStruct.vals[2]);

        LibWrap.TestArrayInStruct(ref myStruct);
        Console.WriteLine("\nStructure with array after call:");
        Console.WriteLine(myStruct.flag);
        Console.WriteLine("{0} {1} {2}", myStruct.vals[0],
            myStruct.vals[1], myStruct.vals[2]);
    }
}
```

```
Public Class App
    Public Shared Sub Main()
        ' Structure with a pointer to another structure.
        Dim personName As MyPerson
        personName.first = "Mark"
        personName.last = "Lee"

        Dim personAll As MyPerson2
        personAll.age = 30

        Dim buffer As IntPtr = Marshal.AllocCoTaskMem(Marshal.SizeOf(
            personName))
        Marshal.StructureToPtr(personName, buffer, False)

        personAll.person = buffer

        Console.WriteLine(ControlChars.CrLf & "Person before call:")
        Console.WriteLine("first = {0}, last = {1}, age = {2}",
            personName.first, personName.last, personAll.age)

        Dim res As Integer = LibWrap.TestStructInStruct(personAll)

        Dim personRes As MyPerson =
            CType(Marshal.PtrToStructure(personAll.person,
            GetType(MyPerson)), MyPerson)

        Marshal.FreeCoTaskMem(buffer)

        Console.WriteLine("Person after call:")
        Console.WriteLine("first = {0}, last = {1}, age = {2}",
        personRes.first,
            personRes.last, personAll.age)

        ' Structure with an embedded structure.
        Dim person3 As New MyPerson3()
        person3.person.first = "John"
        person3.person.last = "Evans"
        person3.age = 27
        LibWrap.TestStructInStruct3(person3)

        ' Structure with an embedded array.
        Dim myStruct As New MyArrayStruct()

        myStruct.flag = False
        Dim array(2) As Integer
        myStruct.vals = array
        myStruct.vals(0) = 1
        myStruct.vals(1) = 4
        myStruct.vals(2) = 9

        Console.WriteLine(vbNewLine + "Structure with array before call:")
        Console.WriteLine(myStruct.flag)
        Console.WriteLine("{0} {1} {2}", myStruct.vals(0),
            myStruct.vals(1), myStruct.vals(2))

        LibWrap.TestArrayInStruct(myStruct)
        Console.WriteLine(vbNewLine + "Structure with array after call:")
        Console.WriteLine(myStruct.flag)
        Console.WriteLine("{0} {1} {2}", myStruct.vals(0),
            myStruct.vals(1), myStruct.vals(2))
    End Sub 'Main
End Class 'App
```

FindFile sample

This sample demonstrates how to pass a structure that contains a second, embedded structure to an unmanaged function. It also demonstrates how to use the MarshalAsAttribute attribute to declare a fixed-length array within the structure. In this sample, the embedded structure elements are added to the parent structure. For a sample of an embedded structure that is not flattened, see Structures Sample.

The FindFile sample uses the following unmanaged function, shown with its original function declaration:

- **FindFirstFile** exported from Kernel32.dll.

  ```
  HANDLE FindFirstFile(LPCTSTR lpFileName, LPWIN32_FIND_DATA lpFindFileData);
  ```

The original structure passed to the function contains the following elements:

```
typedef struct _WIN32_FIND_DATA
{
  DWORD    dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
  DWORD    dwReserved0;
  DWORD    dwReserved1;
  TCHAR    cFileName[ MAX_PATH ];
  TCHAR    cAlternateFileName[ 14 ];
} WIN32_FIND_DATA, *PWIN32_FIND_DATA;
```

In this sample, the `FindData` class contains a corresponding data member for each element of the original structure and the embedded structure. In place of two original character buffers, the class substitutes strings. **MarshalAsAttribute** sets the UnmanagedType enumeration to **ByValTStr**, which is used to identify the inline, fixed-length character arrays that appear within the unmanaged structures.

The `LibWrap` class contains a managed prototype of the `FindFirstFile` method, which passes the `FindData` class as a parameter. The parameter must be declared with the InAttribute and OutAttribute attributes because classes, which are reference types, are passed as In parameters by default.

Declaring Prototypes

```
// Declares a class member for each structure element.
[StructLayout(LayoutKind::Sequential, CharSet=CharSet::Auto)]
public ref class FindData
{
public:
    int  fileAttributes;
    // creationTime was an embedded FILETIME structure.
    int  creationTime_lowDateTime;
    int  creationTime_highDateTime;
    // lastAccessTime was an embedded FILETIME structure.
    int  lastAccessTime_lowDateTime;
    int  lastAccessTime_highDateTime;
    // lastWriteTime was an embedded FILETIME structure.
    int  lastWriteTime_lowDateTime;
    int  lastWriteTime_highDateTime;
    int  nFileSizeHigh;
    int  nFileSizeLow;
    int  dwReserved0;
    int  dwReserved1;
    [MarshalAs(UnmanagedType::ByValTStr, SizeConst=260)]
    String^  fileName;
    [MarshalAs(UnmanagedType::ByValTStr, SizeConst=14)]
    String^  alternateFileName;
};

public ref class LibWrap
{
public:
    // Declares a managed prototype for the unmanaged function.
    [DllImport("Kernel32.dll", CharSet=CharSet::Auto)]
    static IntPtr FindFirstFile(String^ fileName, [In, Out]
        FindData^ findFileData);
};
```

```
// Declares a class member for each structure element.
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
public class FindData
{
    public int fileAttributes = 0;
    // creationTime was an embedded FILETIME structure.
    public int creationTime_lowDateTime = 0;
    public int creationTime_highDateTime = 0;
    // lastAccessTime was an embedded FILETIME structure.
    public int lastAccessTime_lowDateTime = 0;
    public int lastAccessTime_highDateTime = 0;
    // lastWriteTime was an embedded FILETIME structure.
    public int lastWriteTime_lowDateTime = 0;
    public int lastWriteTime_highDateTime = 0;
    public int nFileSizeHigh = 0;
    public int nFileSizeLow = 0;
    public int dwReserved0 = 0;
    public int dwReserved1 = 0;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)]
    public string fileName = null;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 14)]
    public string alternateFileName = null;
}

public class LibWrap
{
    // Declares a managed prototype for the unmanaged function.
    [DllImport("Kernel32.dll", CharSet = CharSet.Auto)]
    public static extern IntPtr FindFirstFile(
        string fileName, [In, Out] FindData findFileData);
}
```

```
' Declares a class member for each structure element.
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Auto)>
Public Class FindData
    Public fileAttributes As Integer = 0
    ' creationTime was a by-value FILETIME structure.
    Public creationTime_lowDateTime As Integer = 0
    Public creationTime_highDateTime As Integer = 0
    ' lastAccessTime was a by-value FILETIME structure.
    Public lastAccessTime_lowDateTime As Integer = 0
    Public lastAccessTime_highDateTime As Integer = 0
    ' lastWriteTime was a by-value FILETIME structure.
    Public lastWriteTime_lowDateTime As Integer = 0
    Public lastWriteTime_highDateTime As Integer = 0
    Public nFileSizeHigh As Integer = 0
    Public nFileSizeLow As Integer = 0
    Public dwReserved0 As Integer = 0
    Public dwReserved1 As Integer = 0
    <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=260)>
    Public fileName As String = Nothing
    <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=14)>
    Public alternateFileName As String = Nothing
End Class 'FindData

Public Class LibWrap
    ' Declares a managed prototype for the unmanaged function.
    Declare Auto Function FindFirstFile Lib "Kernel32.dll" (
        ByVal fileName As String, <[In], Out> ByVal findFileData As _
        FindData) As IntPtr
End Class
```

Calling Functions

```
public ref class App
{
public:
    static void Main()
    {
        FindData^ fd = gcnew FindData();
        IntPtr handle = LibWrap::FindFirstFile("C:\\*.*", fd);
        Console::WriteLine("The first file: {0}", fd->fileName);
    }
};
```

```
public class App
{
    public static void Main()
    {
        FindData fd = new FindData();
        IntPtr handle = LibWrap.FindFirstFile("C:\\*.*", fd);
        Console.WriteLine($"The first file: {fd.fileName}");
    }
}
```

```
Public Class App
    Public Shared Sub Main()
        Dim fd As New FindData()
        Dim handle As IntPtr = LibWrap.FindFirstFile("C:\*.*", fd)
        Console.WriteLine($"The first file: {fd.fileName}")
    End Sub
End Class
```

Unions sample

This sample demonstrates how to pass structures containing only value types, and structures containing a value type and a string as parameters to an unmanaged function expecting a union. A union represents a memory location that can be shared by two or more variables.

The Unions sample uses the following unmanaged function, shown with its original function declaration:

- **TestUnion** exported from PinvokeLib.dll.

  ```
  void TestUnion(MYUNION u, int type);
  ```

PinvokeLib.dll is a custom unmanaged library that contains an implementation for the previously listed function and two unions, **MYUNION** and **MYUNION2**. The unions contain the following elements:

```
union MYUNION
{
    int number;
    double d;
}

union MYUNION2
{
    int i;
    char str[128];
};
```

In managed code, unions are defined as structures. The `MyUnion` structure contains two value types as its members: an integer and a double. The StructLayoutAttribute attribute is set to control the precise position of each data member. The FieldOffsetAttribute attribute provides the physical position of fields within the unmanaged representation of a union. Notice that both members have the same offset values, so the members can define the same piece of memory.

`MyUnion2_1` and `MyUnion2_2` contain a value type (integer) and a string, respectively. In managed code, value types and reference types are not permitted to overlap. This sample uses method overloading to enable the caller to use both types when calling the same unmanaged function. The layout of `MyUnion2_1` is explicit and has a precise offset value. In contrast, `MyUnion2_2` has a sequential layout, because explicit layouts are not permitted with reference types. The MarshalAsAttribute attribute sets the UnmanagedType enumeration to **ByValTStr**, which is used to identify the inline, fixed-length character arrays that appear within the unmanaged representation of the union.

The `LibWrap` class contains the prototypes for the `TestUnion` and `TestUnion2` methods. `TestUnion2` is overloaded to declare `MyUnion2_1` or `MyUnion2_2` as parameters.

Declaring Prototypes

```
// Declares managed structures instead of unions.
[StructLayout(LayoutKind::Explicit)]
public value struct MyUnion
{
public:
    [FieldOffset(0)]
    int i;
    [FieldOffset(0)]
    double d;
};

[StructLayout(LayoutKind::Explicit, Size=128)]
public value struct MyUnion2_1
{
public:
    [FieldOffset(0)]
    int i;
};

[StructLayout(LayoutKind::Sequential)]
public value struct MyUnion2_2
{
public:
    [MarshalAs(UnmanagedType::ByValTStr, SizeConst=128)]
    String^ str;
};

public ref class LibWrap
{
public:
    // Declares managed prototypes for unmanaged function.
    [DllImport("..\\LIB\\PInvokeLib.dll")]
    static void TestUnion(MyUnion u, int type);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
    static void TestUnion2(MyUnion2_1 u, int type);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
    static void TestUnion2(MyUnion2_2 u, int type);
};
```

```
// Declares managed structures instead of unions.
[StructLayout(LayoutKind.Explicit)]
public struct MyUnion
{
    [FieldOffset(0)]
    public int i;
    [FieldOffset(0)]
    public double d;
}

[StructLayout(LayoutKind.Explicit, Size = 128)]
public struct MyUnion2_1
{
    [FieldOffset(0)]
    public int i;
}

[StructLayout(LayoutKind.Sequential)]
public struct MyUnion2_2
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    public string str;
}

public class LibWrap
{
    // Declares managed prototypes for unmanaged function.
    [DllImport("..\\LIB\\PInvokeLib.dll")]
    public static extern void TestUnion(MyUnion u, int type);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
    public static extern void TestUnion2(MyUnion2_1 u, int type);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
    public static extern void TestUnion2(MyUnion2_2 u, int type);
}
```

```
' Declares managed structures instead of unions.
<StructLayout(LayoutKind.Explicit)>
Public Structure MyUnion
    <FieldOffset(0)> Public i As Integer
    <FieldOffset(0)> Public d As Double
End Structure 'MyUnion

<StructLayout(LayoutKind.Explicit, Size:=128)>
Public Structure MyUnion2_1
    <FieldOffset(0)> Public i As Integer
End Structure 'MyUnion2_1

<StructLayout(LayoutKind.Sequential)>
Public Structure MyUnion2_2
    <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=128)>
    Public str As String
End Structure 'MyUnion2_2

Public Class LibWrap
    ' Declares managed prototypes for unmanaged function.
    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Sub TestUnion(
        ByVal u As MyUnion, ByVal type As Integer)
    End Sub

    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Overloads Shared Sub TestUnion2(
        ByVal u As MyUnion2_1, ByVal type As Integer)
    End Sub

    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Overloads Shared Sub TestUnion2(
        ByVal u As MyUnion2_2, ByVal type As Integer)
    End Sub
End Class 'LibWrap
```

Calling Functions

```
public ref class App
{
public:
    static void Main()
    {
        MyUnion mu;// = new MyUnion();
        mu.i = 99;
        LibWrap::TestUnion(mu, 1);

        mu.d = 99.99;
        LibWrap::TestUnion(mu, 2);

        MyUnion2_1 mu2_1;// = new MyUnion2_1();
        mu2_1.i = 99;
        LibWrap::TestUnion2(mu2_1, 1);

        MyUnion2_2 mu2_2;// = new MyUnion2_2();
        mu2_2.str = "*** string ***";
        LibWrap::TestUnion2(mu2_2, 2);
    }
};
```

```
public class App
{
    public static void Main()
    {
        MyUnion mu = new MyUnion();
        mu.i = 99;
        LibWrap.TestUnion(mu, 1);

        mu.d = 99.99;
        LibWrap.TestUnion(mu, 2);

        MyUnion2_1 mu2_1 = new MyUnion2_1();
        mu2_1.i = 99;
        LibWrap.TestUnion2(mu2_1, 1);

        MyUnion2_2 mu2_2 = new MyUnion2_2();
        mu2_2.str = "*** string ***";
        LibWrap.TestUnion2(mu2_2, 2);
    }
}
```

```
Public Class App
    Public Shared Sub Main()
        Dim mu As New MyUnion()
        mu.i = 99
        LibWrap.TestUnion(mu, 1)

        mu.d = 99.99
        LibWrap.TestUnion(mu, 2)

        Dim mu2_1 As New MyUnion2_1()
        mu2_1.i = 99
        LibWrap.TestUnion2(mu2_1, 1)

        Dim mu2_2 As New MyUnion2_2()
        mu2_2.str = "*** string ***"
        LibWrap.TestUnion2(mu2_2, 2)
    End Sub 'Main
End Class 'App
```

SysTime sample

This sample demonstrates how to pass a pointer to a class to an unmanaged function that expects a pointer to a structure.

The SysTime sample uses the following unmanaged function, shown with its original function declaration:

- **GetSystemTime** exported from Kernel32.dll.

  ```
  VOID GetSystemTime(LPSYSTEMTIME lpSystemTime);
  ```

The original structure passed to the function contains the following elements:

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

In this sample, the `SystemTime` class contains the elements of the original structure represented as class members. The StructLayoutAttribute attribute is set to ensure that the members are arranged in memory sequentially, in the order in which they appear.

The `LibWrap` class contains a managed prototype of the `GetSystemTime` method, which passes the `SystemTime` class as an In/Out parameter by default. The parameter must be declared with the InAttribute and OutAttribute attributes because classes, which are reference types, are passed as In parameters by default. For the caller to receive the results, these directional attributes must be applied explicitly. The `App` class creates a new instance of the `SystemTime` class and accesses its data fields.

Code Samples

```
using namespace System;
using namespace System::Runtime::InteropServices;     // For StructLayout, DllImport

[StructLayout(LayoutKind::Sequential)]
public ref class SystemTime
{
public:
    unsigned short year;
    unsigned short month;
    unsigned short weekday;
    unsigned short day;
    unsigned short hour;
    unsigned short minute;
    unsigned short second;
    unsigned short millisecond;
};

public class LibWrap
{
public:
    // Declares a managed prototype for the unmanaged function using Platform Invoke.
    [DllImport("Kernel32.dll")]
    static void GetSystemTime([In,Out] SystemTime^ st);
};

public class App
{
public:
    static void Main()
    {
        Console::WriteLine("C++/CLI SysTime Sample using Platform Invoke");
        SystemTime^ st = gcnew SystemTime();
        LibWrap::GetSystemTime(st);
        Console::Write("The Date is: ");
        Console::Write("{0} {1} {2}",  st->month, st->day, st->year);
    }
};

int main()
{
    App::Main();
}
// The program produces output similar to the following:
//
// C++/CLI SysTime Sample using Platform Invoke
// The Date is: 3 21 2010
```

```
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public class SystemTime
{
    public ushort year;
    public ushort month;
    public ushort weekday;
    public ushort day;
    public ushort hour;
    public ushort minute;
    public ushort second;
    public ushort millisecond;
}

public class LibWrap
{
    // Declares a managed prototype for the unmanaged function using Platform Invoke.
    [DllImport("Kernel32.dll")]
    public static extern void GetSystemTime([In, Out] SystemTime st);
}

public class App
{
    public static void Main()
    {
        Console.WriteLine("C# SysTime Sample using Platform Invoke");
        SystemTime st = new SystemTime();
        LibWrap.GetSystemTime(st);
        Console.Write("The Date is: ");
        Console.Write("{0} {1} {2}", st.month, st.day, st.year);
    }
}

// The program produces output similar to the following:
//
// C# SysTime Sample using Platform Invoke
// The Date is: 3 21 2010
```

```
Imports System
Imports System.Runtime.InteropServices

' Declares a class member for each structure element.
<StructLayout(LayoutKind.Sequential)>
Public Class SystemTime
    Public year As Short
    Public month As Short
    Public weekday As Short
    Public day As Short
    Public hour As Short
    Public minute As Short
    Public second As Short
    Public millisecond As Short
End Class 'SystemTime

Public Class LibWrap
    ' Declares a managed prototype for the unmanaged function.
    Declare Sub GetSystemTime Lib "Kernel32.dll" (
        <[In](), Out()> ByVal st As SystemTime)
End Class 'LibWrap

Public Class App
    Public Shared Sub Main()
        Console.WriteLine("VB .NET SysTime Sample using Platform Invoke")
        Dim st As New SystemTime()
        LibWrap.GetSystemTime(st)
        Console.Write("The Date is: {0} {1} {2}", st.month, st.day, st.year)
    End Sub 'Main
End Class 'App

' The program produces output similar to the following:
'
' VB .NET SysTime Sample using Platform Invoke
' The Date is: 3 21 2010
```

OutArrayOfStructs sample

This sample shows how to pass an array of structures that contains integers and strings as Out parameters to an unmanaged function.

This sample demonstrates how to call a native function by using the Marshal class and by using unsafe code.

This sample uses a wrapper functions and platform invokes defined in PinvokeLib.dll, also provided in the source files. It uses the `TestOutArrayOfStructs` function and the `MYSTRSTRUCT2` structure. The structure contains the following elements:

```
typedef struct _MYSTRSTRUCT2
{
   char* buffer;
   UINT size;
} MYSTRSTRUCT2;
```

The `MyStruct` class contains a string object of ANSI characters. The CharSet field specifies ANSI format. `MyUnsafeStruct`, is a structure containing an IntPtr type instead of a string.

The `LibWrap` class contains the overloaded `TestOutArrayOfStructs` prototype method. If a method declares a pointer as a parameter, the class should be marked with the `unsafe` keyword. Because Visual Basic cannot use unsafe code, the overloaded method, unsafe modifier, and the `MyUnsafeStruct` structure are unnecessary.

The `App` class implements the `UsingMarshaling` method, which performs all the tasks necessary to pass the array.

The array is marked with the `out` (`ByRef` in Visual Basic) keyword to indicate that data passes from callee to caller. The implementation uses the following Marshal class methods:

- PtrToStructure to marshal data from the unmanaged buffer to a managed object.
- DestroyStructure to release the memory reserved for strings in the structure.
- FreeCoTaskMem to release the memory reserved for the array.

As previously mentioned, C# allows unsafe code and Visual Basic does not. In the C# sample, `UsingUnsafePointer` is an alternative method implementation that uses pointers instead of the Marshal class to pass back the array containing the `MyUnsafeStruct` structure.

Declaring Prototypes

```
// Declares a class member for each structure element.
[StructLayout(LayoutKind::Sequential, CharSet=CharSet::Ansi)]
public ref class MyStruct
{
public:
    String^ buffer;
    int size;
};

// Declares a structure with a pointer.
[StructLayout(LayoutKind::Sequential)]
public value struct MyUnsafeStruct
{
public:
    IntPtr buffer;
    int size;
};

public ref class LibWrap
{
public:
    // Declares managed prototypes for the unmanaged function.
    [DllImport("..\\LIB\\PInvokeLib.dll")]
    static void TestOutArrayOfStructs(int% size,
        IntPtr% outArray);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
    static void TestOutArrayOfStructs(int% size,
        MyUnsafeStruct** outArray);
};
```

```
// Declares a class member for each structure element.
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public class MyStruct
{
    public string buffer;
    public int size;
}

// Declares a structure with a pointer.
[StructLayout(LayoutKind.Sequential)]
public struct MyUnsafeStruct
{
    public IntPtr buffer;
    public int size;
}

public unsafe class LibWrap
{
    // Declares managed prototypes for the unmanaged function.
    [DllImport("..\\LIB\\PInvokeLib.dll")]
    public static extern void TestOutArrayOfStructs(
        out int size, out IntPtr outArray);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
    public static extern void TestOutArrayOfStructs(
        out int size, MyUnsafeStruct** outArray);
}
```

```
' Declares a class member for each structure element.
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Ansi)>
Public Class MyStruct
    Public buffer As String
    Public someSize As Integer
End Class 'MyStruct

Public Class LibWrap
    ' Declares a managed prototype for the unmanaged function.
    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Sub TestOutArrayOfStructs(
        ByRef arrSize As Integer, ByRef outArray As IntPtr)
    End Sub
End Class 'LibWrap
```

Calling Functions

```
public ref class App
{
public:
    static void Main()
    {
        Console::WriteLine("\nUsing marshal class\n");
        UsingMarshaling();
        Console::WriteLine("\nUsing unsafe code\n");
        UsingUnsafePointer();
    }

    static void UsingMarshaling()
    {
        int size;
        IntPtr outArray;

        LibWrap::TestOutArrayOfStructs(size, outArray);
        array<MyStruct^>^ manArray = gcnew array<MyStruct^>(size);
        IntPtr current = outArray;
        for (int i = 0; i < size; i++)
        {
            manArray[i] = gcnew MyStruct();
            Marshal::PtrToStructure(current, manArray[i]);

            Marshal::DestroyStructure(current, MyStruct::typeid);
            //current = (IntPtr)((long)current + Marshal::SizeOf(manArray[i]));
            current = current + Marshal::SizeOf(manArray[i]);

            Console::WriteLine("Element {0}: {1} {2}", i, manArray[i]->buffer,
                manArray[i]->size);
        }
        Marshal::FreeCoTaskMem(outArray);
    }

    static void UsingUnsafePointer()
    {
        int size;
        MyUnsafeStruct* pResult;

        LibWrap::TestOutArrayOfStructs(size, &pResult);
        MyUnsafeStruct* pCurrent = pResult;
        for (int i = 0; i < size; i++, pCurrent++)
        {
            Console::WriteLine("Element {0}: {1} {2}", i,
                Marshal::PtrToStringAnsi(pCurrent->buffer), pCurrent->size);
            Marshal::FreeCoTaskMem(pCurrent->buffer);
        }
        Marshal::FreeCoTaskMem((IntPtr)pResult);
    }
};
```

```
public class App
{
    public static void Main()
    {
        Console.WriteLine("\nUsing marshal class\n");
        UsingMarshaling();
        Console.WriteLine("\nUsing unsafe code\n");
        UsingUnsafePointer();
    }

    public static void UsingMarshaling()
    {
        int size;
        IntPtr outArray;

        LibWrap.TestOutArrayOfStructs(out size, out outArray);
        MyStruct[] manArray = new MyStruct[size];
        IntPtr current = outArray;
        for (int i = 0; i < size; i++)
        {
            manArray[i] = new MyStruct();
            Marshal.PtrToStructure(current, manArray[i]);

            //Marshal.FreeCoTaskMem( (IntPtr)Marshal.ReadInt32( current ));
            Marshal.DestroyStructure(current, typeof(MyStruct));
            current = (IntPtr)((long)current + Marshal.SizeOf(manArray[i]));

            Console.WriteLine("Element {0}: {1} {2}", i, manArray[i].buffer,
                manArray[i].size);
        }

        Marshal.FreeCoTaskMem(outArray);
    }

    public static unsafe void UsingUnsafePointer()
    {
        int size;
        MyUnsafeStruct* pResult;

        LibWrap.TestOutArrayOfStructs(out size, &pResult);
        MyUnsafeStruct* pCurrent = pResult;
        for (int i = 0; i < size; i++, pCurrent++)
        {
            Console.WriteLine("Element {0}: {1} {2}", i,
                Marshal.PtrToStringAnsi(pCurrent->buffer), pCurrent->size);
            Marshal.FreeCoTaskMem(pCurrent->buffer);
        }

        Marshal.FreeCoTaskMem((IntPtr)pResult);
    }
}
```

```
Public Class App
    Public Shared Sub Main()
        Console.WriteLine(vbNewLine + "Using marshal class" + vbNewLine)
        UsingMarshaling()
        'Visual Basic 2005 cannot use unsafe code.
    End Sub 'Main

    Public Shared Sub UsingMarshaling()
        Dim arrSize As Integer
        Dim outArray As IntPtr

        LibWrap.TestOutArrayOfStructs(arrSize, outArray)
        Dim manArray(arrSize - 1) As MyStruct
        Dim current As IntPtr = outArray
        Dim i As Integer

        For i = 0 To arrSize - 1
            manArray(i) = New MyStruct()
            Marshal.PtrToStructure(current, manArray(i))

            Marshal.DestroyStructure(current, GetType(MyStruct))
            current = IntPtr.op_Explicit(current.ToInt64() _
                + Marshal.SizeOf(manArray(i)))

            Console.WriteLine("Element {0}: {1} {2}", i, manArray(i).
                buffer, manArray(i).someSize)
        Next i
        Marshal.FreeCoTaskMem(outArray)
    End Sub 'UsingMarshal
End Class 'App
```

See also

- Marshaling Data with Platform Invoke
- Marshaling Strings
- Marshaling Different Types of Arrays

Marshaling Different Types of Arrays

4/28/2019 • 10 minutes to read • Edit Online

An array is a reference type in managed code that contains one or more elements of the same type. Although arrays are reference types, they are passed as In parameters to unmanaged functions. This behavior is inconsistent with way managed arrays are passed to managed objects, which is as In/Out parameters. For additional details, see Copying and Pinning.

The following table lists marshaling options for arrays and describes their usage.

ARRAY	DESCRIPTION
Of integers by value.	Passes an array of integers as an In parameter.
Of integers by reference.	Passes an array of integers as an In/Out parameter.
Of integers by value (two-dimensional).	Passes a matrix of integers as an In parameter.
Of strings by value.	Passes an array of strings as an In parameter.
Of structures with integers.	Passes an array of structures that contain integers as an In parameter.
Of structures with strings.	Passes an array of structures that contain only strings as an In/Out parameter. Members of the array can be changed.

Example

This sample demonstrates how to pass the following types of arrays:

- Array of integers by value.
- Array of integers by reference, which can be resized.
- Multidimensional array (matrix) of integers by value.
- Array of strings by value.
- Array of structures with integers.
- Array of structures with strings.

Unless an array is explicitly marshaled by reference, the default behavior marshals the array as an In parameter. You can change this behavior by applying the InAttribute and OutAttribute attributes explicitly.

The Arrays sample uses the following unmanaged functions, shown with their original function declaration:

- **TestArrayOfInts** exported from PinvokeLib.dll.

  ```
  int TestArrayOfInts(int* pArray, int pSize);
  ```

- **TestRefArrayOfInts** exported from PinvokeLib.dll.

```
int TestRefArrayOfInts(int** ppArray, int* pSize);
```

- **TestMatrixOfInts** exported from PinvokeLib.dll.

```
int TestMatrixOfInts(int pMatrix[][COL_DIM], int row);
```

- **TestArrayOfStrings** exported from PinvokeLib.dll.

```
int TestArrayOfStrings(char** ppStrArray, int size);
```

- **TestArrayOfStructs** exported from PinvokeLib.dll.

```
int TestArrayOfStructs(MYPOINT* pPointArray, int size);
```

- **TestArrayOfStructs2** exported from PinvokeLib.dll.

```
int TestArrayOfStructs2 (MYPERSON* pPersonArray, int size);
```

PinvokeLib.dll is a custom unmanaged library that contains implementations for the previously listed functions and two structure variables, **MYPOINT** and **MYPERSON**. The structures contain the following elements:

```
typedef struct _MYPOINT
{
   int x;
   int y;
} MYPOINT;

typedef struct _MYPERSON
{
   char* first;
   char* last;
} MYPERSON;
```

In this sample, the `MyPoint` and `MyPerson` structures contain embedded types. The StructLayoutAttribute attribute is set to ensure that the members are arranged in memory sequentially, in the order in which they appear.

The `LibWrap` class contains a set of methods called by the `App` class. For specific details about passing arrays, see the comments in the following sample. An array, which is a reference type, is passed as an In parameter by default. For the caller to receive the results, **InAttribute** and **OutAttribute** must be applied explicitly to the argument containing the array.

Declaring Prototypes

```
// Declares a managed structure for each unmanaged structure.
[StructLayout(LayoutKind.Sequential)]
public struct MyPoint
{
    public int X;
    public int Y;

    public MyPoint(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }
}

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct MyPerson
{
    public string First;
    public string Last;

    public MyPerson(string first, string last)
    {
        this.First = first;
        this.Last = last;
    }
}

public class LibWrap
{
    // Declares a managed prototype for an array of integers by value.
    // The array size cannot be changed, but the array is copied back.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    public static extern int TestArrayOfInts(
        [In, Out] int[] array, int size);

    // Declares a managed prototype for an array of integers by reference.
    // The array size can change, but the array is not copied back
    // automatically because the marshaler does not know the resulting size.
    // The copy must be performed manually.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    public static extern int TestRefArrayOfInts(
        ref IntPtr array, ref int size);

    // Declares a managed prototype for a matrix of integers by value.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    public static extern int TestMatrixOfInts(
        [In, Out] int[,] pMatrix, int row);

    // Declares a managed prototype for an array of strings by value.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    public static extern int TestArrayOfstrings(
        [In, Out] string[] stringArray, int size);

    // Declares a managed prototype for an array of structures with integers.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    public static extern int TestArrayOfStructs(
        [In, Out] MyPoint[] pointArray, int size);

    // Declares a managed prototype for an array of structures with strings.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    public static extern int TestArrayOfStructs2(
        [In, Out] MyPerson[] personArray, int size);
}
```

```
' Declares a managed structure for each unmanaged structure.
<StructLayout(LayoutKind.Sequential)>
```

```
Public Structure MyPoint
    Public x As Integer
    Public y As Integer
    Public Sub New(x As Integer, y As Integer)
        Me.x = x
        Me.y = y
    End Sub 'New
End Structure

<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Ansi)>
Public Structure MyPerson
    Public first As String
    Public last As String
    Public Sub New(first As String, last As String)
        Me.first = first
        Me.last = last
    End Sub 'New
End Structure

Public Class LibWrap
    ' Declares a managed prototype for an array of integers by value.
    ' The array size cannot be changed, but the array is copied back.
    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Function TestArrayOfInts(
        <[In], Out> ByVal myArray() As Integer, ByVal size As Integer) _
        As Integer
    End Function

    ' Declares managed prototype for an array of integers by reference.
    ' The array size can change, but the array is not copied back
    ' automatically because the marshaler does not know the resulting size.
    ' The copy must be performed manually.
    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Function TestRefArrayOfInts(
        ByRef myArray As IntPtr, ByRef size As Integer) As Integer
    End Function

    ' Declares a managed prototype for a matrix of integers by value.
    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Function TestMatrixOfInts(
        <[In], Out> ByVal matrix(,) As Integer, ByVal row As Integer) _
        As Integer
    End Function

    ' Declares a managed prototype for an array of strings by value.
    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Function TestArrayOfStrings(
        <[In], Out> ByVal strArray() As String, ByVal size As Integer) _
        As Integer
    End Function

    ' Declares a managed prototype for an array of structures with
    ' integers.
    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Function TestArrayOfStructs(
        <[In], Out> ByVal pointArray() As MyPoint, ByVal size As Integer) _
        As Integer
    End Function

    ' Declares a managed prototype for an array of structures with strings.
    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Function TestArrayOfStructs2(
        <[In], Out> ByVal personArray() As MyPerson, ByVal size As Integer) _
        As Integer
    End Function
End Class
```

Calling Functions

```
public class App
{
    public static void Main()
    {
        // array ByVal
        int[] array1 = new int[10];
        Console.WriteLine("Integer array passed ByVal before call:");
        for (int i = 0; i < array1.Length; i++)
        {
            array1[i] = i;
            Console.Write(" " + array1[i]);
        }

        int sum1 = LibWrap.TestArrayOfInts(array1, array1.Length);
        Console.WriteLine("\nSum of elements:" + sum1);
        Console.WriteLine("\nInteger array passed ByVal after call:");

        foreach (int i in array1)
        {
            Console.Write(" " + i);
        }

        // array ByRef
        int[] array2 = new int[10];
        int size = array2.Length;
        Console.WriteLine("\n\nInteger array passed ByRef before call:");
        for (int i = 0; i < array2.Length; i++)
        {
            array2[i] = i;
            Console.Write(" " + array2[i]);
        }

        IntPtr buffer = Marshal.AllocCoTaskMem(Marshal.SizeOf(size)
           * array2.Length);
        Marshal.Copy(array2, 0, buffer, array2.Length);

        int sum2 = LibWrap.TestRefArrayOfInts(ref buffer, ref size);
        Console.WriteLine("\nSum of elements:" + sum2);
        if (size > 0)
        {
            int[] arrayRes = new int[size];
            Marshal.Copy(buffer, arrayRes, 0, size);
            Marshal.FreeCoTaskMem(buffer);
            Console.WriteLine("\nInteger array passed ByRef after call:");
            foreach (int i in arrayRes)
            {
                Console.Write(" " + i);
            }
        }
        else
        {
            Console.WriteLine("\nArray after call is empty");
        }

        // matrix ByVal
        const int DIM = 5;
        int[,] matrix = new int[DIM, DIM];

        Console.WriteLine("\n\nMatrix before call:");
        for (int i = 0; i < DIM; i++)
        {
            for (int j = 0; j < DIM; j++)
            {
                matrix[i, j] = j;
                Console.Write(" " + matrix[i, j]);
            }
            Console.WriteLine("");
        }

        int sum3 = LibWrap.TestMatrixOfInts(matrix, DIM);
        Console.WriteLine("\nSum of elements:" + sum3);
        Console.WriteLine("\nMatrix after call:");
        for (int i = 0; i < DIM; i++)
        {
            for (int j = 0; j < DIM; j++)
            {
                Console.Write(" " + matrix[i, j]);
            }

            Console.WriteLine("");
        }

        // string array ByVal
        string[] strArray = { "one", "two", "three", "four", "five" };
        Console.WriteLine("\n\nstring array before call:");
        foreach (string s in strArray)
        {
            Console.Write(" " + s);
        }

        int lenSum = LibWrap.TestArrayOfstrings(strArray, strArray.Length);
        Console.WriteLine("\nSum of string lengths:" + lenSum);
        Console.WriteLine("\nstring array after call:");
        foreach (string s in strArray)
        {
            Console.Write(" " + s);
        }

        // struct array ByVal
        MyPoint[] points = { new MyPoint(1, 1), new MyPoint(2, 2), new MyPoint(3, 3) };
        Console.WriteLine("\n\nPoints array before call:");
        foreach (MyPoint p in points)
        {
            Console.WriteLine($"X = {p.X}, Y = {p.Y}");
        }

        int allSum = LibWrap.TestArrayOfStructs(points, points.Length);
        Console.WriteLine("\nSum of points:" + allSum);
        Console.WriteLine("\nPoints array after call:");
        foreach (MyPoint p in points)
        {
            Console.WriteLine($"X = {p.X}, Y = {p.Y}");
        }

        // struct with strings array ByVal
        MyPerson[] persons =
        {
            new MyPerson("Kim", "Akers"),
            new MyPerson("Adam", "Barr"),
            new MyPerson("Jo", "Brown")
        };

        Console.WriteLine("\n\nPersons array before call:");
        foreach (MyPerson pe in persons)
        {
            Console.WriteLine($"First = {pe.First}, Last = {pe.Last}");
        }

        int namesSum = LibWrap.TestArrayOfStructs2(persons, persons.Length);
        Console.WriteLine("\nSum of name lengths:" + namesSum);
        Console.WriteLine("\n\nPersons array after call:");
        foreach (MyPerson pe in persons)
        {
            Console.WriteLine($"First = {pe.First}, Last = {pe.Last}");
        }
    }
}
```

```
Public Class App
    Public Shared Sub Main()
        ' array ByVal
        Dim array1(9) As Integer

        Console.WriteLine("Integer array passed ByVal before call:")
        Dim i As Integer
        For i = 0 To array1.Length - 1
            array1(i) = i
            Console.Write(" " & array1(i))
        Next i

        Dim sum1 As Integer = LibWrap.TestArrayOfInts(array1, array1.Length)
        Console.WriteLine(ControlChars.CrLf & "Sum of elements:" & sum1)
        Console.WriteLine(ControlChars.CrLf & "Integer array passed ByVal after call:")
        For Each i In array1
            Console.Write(" " & i)
        Next i

        ' array ByRef
        Dim array2(9) As Integer
        Dim arraySize As Integer = array2.Length
        Console.WriteLine(ControlChars.CrLf & ControlChars.CrLf &
            "Integer array passed ByRef before call:")
        For i = 0 To array2.Length - 1
            array2(i) = i
            Console.Write(" " & array2(i))
        Next i
        Dim buffer As IntPtr = Marshal.AllocCoTaskMem(Marshal.SizeOf(
            arraySize) * array2.Length)
        Marshal.Copy(array2, 0, buffer, array2.Length)
        Dim sum2 As Integer = LibWrap.TestRefArrayOfInts(buffer,
            arraySize)
        Console.WriteLine(ControlChars.CrLf & "Sum of elements:" & sum2)

        If arraySize > 0 Then
            Dim arrayRes(arraySize - 1) As Integer
            Marshal.Copy(buffer, arrayRes, 0, arraySize)
            Marshal.FreeCoTaskMem(buffer)

            Console.WriteLine(ControlChars.CrLf & "Integer array passed ByRef after call:")
            For Each i In arrayRes
                Console.Write(" " & i)
            Next i
        Else
            Console.WriteLine(ControlChars.CrLf & "Array after call is empty")
        End If

        ' matrix ByVal
        Const [DIM] As Integer = 4
        Dim matrix([DIM], [DIM]) As Integer

        Console.WriteLine(ControlChars.CrLf & ControlChars.CrLf &
            "Matrix before call:")
        For i = 0 To [DIM]
            Dim j As Integer
            For j = 0 To [DIM]
                matrix(i, j) = j
                Console.Write(" " & matrix(i, j))
            Next j
            Console.WriteLine("")
        Next i
        Dim sum3 As Integer = LibWrap.TestMatrixOfInts(matrix, [DIM] + 1)
        Console.WriteLine(ControlChars.CrLf & "Sum of elements:" & sum3)
        Console.WriteLine(ControlChars.CrLf & "Matrix after call:")
        For i = 0 To [DIM]
            Dim j As Integer
            For j = 0 To [DIM]
                Console.Write(" " & matrix(i, j))
            Next j
            Console.WriteLine("")
        Next i

        ' string array ByVal
        Dim strArray As String() = {"one", "two", "three", "four",
            "five"}
        Console.WriteLine(ControlChars.CrLf & ControlChars.CrLf &
            "String array before call:")
        Dim s As String
        For Each s In strArray
            Console.Write(" " & s)
        Next s
        Dim lenSum As Integer = LibWrap.TestArrayOfStrings(
            strArray, strArray.Length)
        Console.WriteLine(ControlChars.CrLf &
            "Sum of string lengths:" & lenSum)
        Console.WriteLine(ControlChars.CrLf & "String array after call:")
        For Each s In strArray
            Console.Write(" " & s)
        Next s

        ' struct array ByVal
        Dim points As MyPoint() = {New MyPoint(1, 1), New MyPoint(2, 2),
            New MyPoint(3, 3)}
        Console.WriteLine(ControlChars.CrLf & ControlChars.CrLf &
            "Points array before call:")
        Dim p As MyPoint
        For Each p In points
            Console.WriteLine($"x = {p.x}, y = {p.y}")
        Next p
        Dim allSum As Integer = LibWrap.TestArrayOfStructs(points,
            points.Length)
        Console.WriteLine(ControlChars.CrLf & "Sum of points:" & allSum)
        Console.WriteLine(ControlChars.CrLf & "Points array after call:")
        For Each p In points
            Console.WriteLine($"x = {p.x}, y = {p.y}")
        Next p

        ' struct with strings array ByVal
        Dim persons As MyPerson() = {New MyPerson("Kim", "Akers"),
            New MyPerson("Adam", "Barr"),
            New MyPerson("Jo", "Brown")}
        Console.WriteLine(ControlChars.CrLf & ControlChars.CrLf &
            "Persons array before call:")
        Dim pe As MyPerson
        For Each pe In persons
            Console.WriteLine($"first = {pe.first}, last = {pe.last}")
        Next pe

        Dim namesSum As Integer = LibWrap.TestArrayOfStructs2(persons,
            persons.Length)
        Console.WriteLine(ControlChars.CrLf & "Sum of name lengths:" &
            namesSum)
        Console.WriteLine(ControlChars.CrLf & ControlChars.CrLf _
            & "Persons array after call:")
        For Each pe In persons
            Console.WriteLine($"first = {pe.first}, last = {pe.last}")
        Next pe
    End Sub
End Class
```

See also

- Platform invoke data types
- Creating Prototypes in Managed Code

Marshaling a Delegate as a Callback Method

4/28/2019 • 2 minutes to read • Edit Online

This sample demonstrates how to pass delegates to an unmanaged function expecting function pointers. A delegate is a class that can hold a reference to a method and is equivalent to a type-safe function pointer or a callback function.

NOTE

When you use a delegate inside a call, the common language runtime protects the delegate from being garbage collected for the duration of that call. However, if the unmanaged function stores the delegate to use after the call completes, you must manually prevent garbage collection until the unmanaged function finishes with the delegate. For more information, see the HandleRef Sample and GCHandle Sample.

The Callback sample uses the following unmanaged functions, shown with their original function declaration:

- `TestCallBack` exported from PinvokeLib.dll.

  ```
  void TestCallBack(FPTR pf, int value);
  ```

- `TestCallBack2` exported from PinvokeLib.dll.

  ```
  void TestCallBack2(FPTR2 pf2, char* value);
  ```

PinvokeLib.dll is a custom unmanaged library that contains an implementation for the previously listed functions.

In this sample, the `LibWrap` class contains managed prototypes for the `TestCallBack` and `TestCallBack2` methods. Both methods pass a delegate to a callback function as a parameter. The signature of the delegate must match the signature of the method it references. For example, the `FPtr` and `FPtr2` delegates have signatures that are identical to the `DoSomething` and `DoSomething2` methods.

Declaring Prototypes

```
public delegate bool FPtr(int value);
public delegate bool FPtr2(String^ value);

public ref class LibWrap
{
public:
    // Declares managed prototypes for unmanaged functions.
    [DllImport("..\\LIB\\PinvokeLib.dll")]
    static void TestCallBack(FPtr^ cb, int value);

    [DllImport("..\\LIB\\PinvokeLib.dll")]
    static void TestCallBack2(FPtr2^ cb2, String^ value);
};
```

```
public delegate bool FPtr(int value);
public delegate bool FPtr2(string value);

public class LibWrap
{
    // Declares managed prototypes for unmanaged functions.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    public static extern void TestCallBack(FPtr cb, int value);

    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    public static extern void TestCallBack2(FPtr2 cb2, string value);
}
```

```
Public Delegate Function FPtr(ByVal value As Integer) As Boolean
Public Delegate Function FPtr2(ByVal value As String) As Boolean

Public Class LibWrap
    ' Declares managed prototypes for unmanaged functions.
    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Sub TestCallBack(
        ByVal cb As FPtr, ByVal value As Integer)
    End Sub

    <DllImport("..\LIB\PinvokeLib.dll", CallingConvention:=CallingConvention.Cdecl)>
    Shared Sub TestCallBack2(
        ByVal cb2 As FPtr2, ByVal value As String)
    End Sub
End Class
```

Calling Functions

```
public ref class App
{
public:
    static void Main()
    {
        FPtr^ cb = gcnew FPtr(&App::DoSomething);
        LibWrap::TestCallBack(cb, 99);
        FPtr2^ cb2 = gcnew FPtr2(&App::DoSomething2);
        LibWrap::TestCallBack2(cb2, "abc");
    }

    static bool DoSomething(int value)
    {
        Console::WriteLine("\nCallback called with param: {0}", value);
        // ...
        return true;
    }

    static bool DoSomething2(String^ value)
    {
        Console::WriteLine("\nCallback called with param: {0}", value);
        // ...
        return true;
    }
};
```

```csharp
public class App
{
    public static void Main()
    {
        FPtr cb = new FPtr(App.DoSomething);
        LibWrap.TestCallBack(cb, 99);
        FPtr2 cb2 = new FPtr2(App.DoSomething2);
        LibWrap.TestCallBack2(cb2, "abc");
    }

    public static bool DoSomething(int value)
    {
        Console.WriteLine($"\nCallback called with param: {value}");
        // ...
        return true;
    }

    public static bool DoSomething2(string value)
    {
        Console.WriteLine($"\nCallback called with param: {value}");
        // ...
        return true;
    }
}
```

```vb
Public Class App
    Public Shared Sub Main()
        Dim cb As FPtr = AddressOf App.DoSomething
        Dim cb2 As FPtr2 = AddressOf App.DoSomething2
        LibWrap.TestCallBack(cb, 99)
        LibWrap.TestCallBack2(cb2, "abc")
    End Sub 'Main

    Public Shared Function DoSomething(ByVal value As Integer) As Boolean
        Console.WriteLine(ControlChars.CrLf + $"Callback called with param: {value}")
        ' ...
        Return True
    End Function

    Public Shared Function DoSomething2(ByVal value As String) As Boolean
        Console.WriteLine(ControlChars.CrLf + $"Callback called with param: {value}")
        ' ...
        Return True
    End Function
End Class
```

See also

- Miscellaneous Marshaling Samples
- Platform Invoke Data Types
- Creating Prototypes in Managed Code

Marshaling Data with COM Interop

4/28/2019 • 2 minutes to read • Edit Online

COM interop provides support for both using COM objects from managed code and exposing managed objects to COM. Support for marshaling data to and from COM is extensive and almost always provides the correct marshaling behavior.

The Windows Software Development Kit (SDK) includes the following COM interop tools:

- Type Library Importer (Tlbimp.exe), which converts a COM type library to an interop assembly. From this assembly, the interop marshaling service generates wrappers that perform data marshaling between managed and unmanaged memory.
- Type Library Exporter (Tlbexp.exe), which produces a COM type library from an assembly and generates a wrapper that performs marshaling during method calls.

The following sections link to topics that describe the processes for customizing interop wrappers when you can (or must) supply the marshaler with additional type information.

In This Section

How to: Create Wrappers Manually
Describes how to create a COM wrapper manually in managed source code.

How to: Migrate Managed-Code DCOM to WCF
Describes how to migrate managed DCOM code to WCF for the most secure solution.

Related Sections

COM Data Types
Provides corresponding managed and unmanaged data types.

Customizing COM Callable Wrappers
Describes how to explicitly marshal data types using the MarshalAsAttribute attribute at design time.

Customizing Runtime Callable Wrappers
Describes how to adjust the marshaling behavior of types in an interop assembly and how to define COM types manually.

Advanced COM Interoperability
Provides links to more information about incorporating COM components into your .NET Framework application.

Assembly to Type Library Conversion Summary
Describes the assembly to type library export conversion process.

Type Library to Assembly Conversion Summary
Describes the type library to assembly import conversion process.

Interoperating Using Generic Types
Describes which actions are supported when using generic types for COM interoperability.

How to: Create Wrappers Manually

4/9/2019 • 2 minutes to read • Edit Online

If you decide to declare COM types manually in managed source code, the best place to start is with an existing Interface Definition Language (IDL) file or type library. When you do not have the IDL file or cannot generate a type library file, you can simulate the COM types by creating managed declarations and exporting the resulting assembly to a type library.

To simulate COM types from managed source

1. Declare the types in a language that is compliant with the Common Language Specification (CLS) and compile the file.
2. Export the assembly containing the types with the Type Library Exporter (Tlbexp.exe).
3. Use the exported COM type library as a basis for declaring COM-oriented managed types.

To create a runtime callable wrapper (RCW)

1. Assuming that you have an IDL file or type library file, decide which classes and interfaces you want to include in the custom RCW. You can exclude any types that you do not intend to use directly or indirectly in your application.
2. Create a source file in a CLS-compliant language and declare the types. See Type Library to Assembly Conversion Summary for a complete description of the import conversion process. Effectively, when you create a custom RCW, you are manually performing the type conversion activity provided by the Type Library Importer (Tlbimp.exe). The example in the next section shows types in an IDL or type library file and their corresponding types in C# code.
3. When the declarations are complete, compile the file as you compile any other managed source code.
4. As with the types imported with Tlbimp.exe, some require additional information, which you can add directly to your code. For details, see How to: Edit Interop Assemblies.

Example

The following code shows an example of the `ISATest` interface and `SATest` class in IDL and the corresponding types in C# source code.

IDL or type library file

```
 [
object,
uuid(40A8C65D-2448-447A-B786-64682CBEF133),
dual,
helpstring("ISATest Interface"),
pointer_default(unique)
 ]
interface ISATest : IDispatch
 {
[id(1), helpstring("method InSArray")]
HRESULT InSArray([in] SAFEARRAY(int) *ppsa, [out,retval] int *pSum);
 };
 [
uuid(116CCA1E-7E39-4515-9849-90790DA6431E),
helpstring("SATest Class")
 ]
coclass SATest
 {
  [default] interface ISATest;
 };
```

Wrapper in managed source code

```
using System;
using System.Runtime.InteropServices;
using System.Runtime.CompilerServices;

[assembly:Guid("E4A992B8-6F5C-442C-96E7-C4778924C753")]
[assembly:ImportedFromTypeLib("SAServerLib")]
namespace SAServer
{
 [ComImport]
 [Guid("40A8C65D-2448-447A-B786-64682CBEF133")]
 [TypeLibType(TypeLibTypeFlags.FLicensed)]
 public interface ISATest
 {
  [DispId(1)]
  //[MethodImpl(MethodImplOptions.InternalCall,
  // MethodCodeType=MethodCodeType.Runtime)]
  int InSArray( [MarshalAs(UnmanagedType.SafeArray,
      SafeArraySubType=VarEnum.VT_I4)] ref int[] param );
 }
 [ComImport]
 [Guid("116CCA1E-7E39-4515-9849-90790DA6431E")]
 [ClassInterface(ClassInterfaceType.None)]
 [TypeLibType(TypeLibTypeFlags.FCanCreate)]
 public class SATest : ISATest
 {
  [DispId(1)]
  [MethodImpl(MethodImplOptions.InternalCall,
  MethodCodeType=MethodCodeType.Runtime)]
  extern int ISATest.InSArray( [MarshalAs(UnmanagedType.SafeArray,
  SafeArraySubType=VarEnum.VT_I4)] ref int[] param );
 }
}
```

See also

- Customizing Runtime Callable Wrappers
- COM Data Types
- How to: Edit Interop Assemblies
- Type Library to Assembly Conversion Summary

- Tlbimp.exe (Type Library Importer)
- Tlbexp.exe (Type Library Exporter)

How to: Migrate Managed-Code DCOM to WCF

4/28/2019 • 10 minutes to read • Edit Online

Windows Communication Foundation (WCF) is the recommended and secure choice over Distributed Component Object Model (DCOM) for managed code calls between servers and clients in a distributed environment. This article shows how you to migrate code from DCOM to WCF for the following scenarios.

- The remote service returns an object by-value to the client
- The client sends an object by-value to the remote service
- The remote service returns an object by-reference to the client

For security reasons, sending an object by-reference from the client to the service is not allowed in WCF. A scenario that requires a conversation back and forth between client and server can be achieved in WCF using a duplex service. For more information about duplex services, see Duplex Services.

For more details about creating WCF services and clients for those services, see Basic WCF Programming, Designing and Implementing Services, and Building Clients.

DCOM example code

For these scenarios, the DCOM interfaces that are illustrated using WCF have the following structure:

```
[ComVisible(true)]
[Guid("AA9C4CDB-55EA-4413-90D2-843F1A49E6E6")]
public interface IRemoteService
{
   Customer GetObjectByValue();
   IRemoteObject GetObjectByReference();
   void SendObjectByValue(Customer customer);
}

[ComVisible(true)]
[Guid("A12C98DE-B6A1-463D-8C24-81E4BBC4351B")]
public interface IRemoteObject
{
}

public class Customer
{
}
```

The service returns an object by-value

For this scenario, you make a call to a service and it method returns an object, which is passed by-value from the server to the client. This scenario represents the following COM call:

```
public interface IRemoteService
{
    Customer GetObjectByValue();
}
```

In this scenario, the client receives a deserialized copy of an object from the remote service. The client can interact with this local copy without calling back to the service. In other words, the client is guaranteed the service will not be involved in any way when methods on the local copy are called. WCF always returns objects from the service by value, so the following steps describe creating a regular WCF service.

Step 1: Define the WCF service interface

Define a public interface for the WCF service and mark it with the [ServiceContractAttribute] attribute. Mark the methods you want to expose to clients with the [OperationContractAttribute] attribute. The following example shows using these attributes to identify the server-side interface and interface methods a client can call. The method used for this scenario is shown in bold.

```
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Web;
. . .
[ServiceContract]
public interface ICustomerManager
{
    [OperationContract]
    void StoreCustomer(Customer customer);

    [OperationContract]     Customer GetCustomer(string firstName, string lastName);

}
```

Step 2: Define the data contract

Next you should create a data contract for the service, which will describe how the data will be exchanged between the service and its clients. Classes described in the data contract should be marked with the [DataContractAttribute] attribute. The individual properties or fields you want visible to both client and server should be marked with the [DataMemberAttribute] attribute. If you want types derived from a class in the data contract to be allowed, you must identify them with the [KnownTypeAttribute] attribute. WCF will only serialize or deserialize types in the service interface and types identified as known types. If you attempt to use a type that is not a known type, an exception will occur.

For more information about data contracts, see Data Contracts.

```
[DataContract]
[KnownType(typeof(PremiumCustomer))]
public class Customer
{
    [DataMember]
    public string Firstname;
    [DataMember]
    public string Lastname;
    [DataMember]
    public Address DefaultDeliveryAddress;
    [DataMember]
    public Address DefaultBillingAddress;
}
 [DataContract]
public class PremiumCustomer : Customer
{
    [DataMember]
    public int AccountID;
}

 [DataContract]
public class Address
{
    [DataMember]
    public string Street;
    [DataMember]
    public string Zipcode;
    [DataMember]
    public string City;
    [DataMember]
    public string State;
    [DataMember]
    public string Country;
}
```

Step 3: Implement the WCF service

Next, you should implement the WCF service class that implements the interface you defined in the previous step.

```
public class CustomerService: ICustomerManager
{
    public void StoreCustomer(Customer customer)
    {
        // write to a database
    }
    public Customer GetCustomer(string firstName, string lastName)
    {
        // read from a database
    }
}
```

Step 4: Configure the service and the client

To run a WCF service, you need to declare an endpoint that exposes that service interface at a specific URL using a specific WCF binding. A binding specifies the transport, encoding and protocol details for the clients and server to communicate. You typically add bindings to the service project's configuration file (web.config). The following shows a binding entry for the example service:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Server.CustomerService">
        <endpoint address="http://localhost:8083/CustomerManager"
                  binding="basicHttpBinding"
                  contract="Shared.ICustomerManager" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Next, you need to configure the client to match the binding information specified by the service. To do so, add the following to the client's application configuration (app.config) file.

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="customermanager"
                address="http://localhost:8083/CustomerManager"
                binding="basicHttpBinding"
                contract="Shared.ICustomerManager"/>
  </system.serviceModel>
</configuration>
```

Step 5: Run the service

Finally, you can self-host it in a console application by adding the following lines to the service app, and starting the app. For more information about other ways to host a WCF service application, Hosting Services.

```
ServiceHost customerServiceHost = new ServiceHost(typeof(CustomerService));
customerServiceHost.Open();
```

Step 6: Call the service from the client

To call the service from the client, you need to create a channel factory for the service, and request a channel, which will enable you to directly call the `GetCustomer` method directly from the client. The channel implements the service's interface and handles the underlying request/reply logic for you. The return value from that method call is the deserialized copy of the service response.

```
ChannelFactory<ICustomerManager> factory =
     new ChannelFactory<ICustomerManager>("customermanager");
ICustomerManager service = factory.CreateChannel();
Customer customer = service.GetCustomer("Mary", "Smith");
```

The client sends a by-value object to the server

In this scenario, the client sends an object to the server, by-value. This means that the server will receive a deserialized copy of the object. The server can call methods on that copy and be guaranteed there is no callback into client code. As mentioned previously, normal WCF exchanges of data are by-value. This guarantees that calling methods on one of these objects executes locally only – it will not invoke code on the client.

This scenario represents the following COM method call:

```
public interface IRemoteService
{
    void SendObjectByValue(Customer customer);
}
```

This scenario uses the same service interface and data contract as shown in the first example. In addition, the client and service will be configured in the same way. In this example, a channel is created to send the object and run the same way. However, for this example, you will create a client that calls the service, passing an object by-value. The service method the client will call in the service contract is shown in bold:

```
[ServiceContract]
public interface ICustomerManager
{
    [OperationContract]     void StoreCustomer(Customer customer);

    [OperationContract]
    Customer GetCustomer(string firstName, string lastName);
}
```

Add code to the client that sends a by-value object

The following code shows how the client creates a new by-value customer object, creates a channel to communicate with the `ICustomerManager` service, and sends the customer object to it.

The customer object will be serialized, and sent to the service, where it is deserialized by the service into a new copy of that object. Any methods the service calls on this object will execute only locally on the server. It's important to note that this code illustrates sending a derived type (`PremiumCustomer`). The service contract expects a `Customer` object, but the service data contract uses the [KnownTypeAttribute] attribute to indicate that `PremiumCustomer` is also allowed. WCF will fail attempts to serialize or deserialize any other type via this service interface.

```
PremiumCustomer customer = new PremiumCustomer();
customer.Firstname = "John";
customer.Lastname = "Doe";
customer.DefaultBillingAddress = new Address();
customer.DefaultBillingAddress.Street = "One Microsoft Way";
customer.DefaultDeliveryAddress = customer.DefaultBillingAddress;
customer.AccountID = 42;

ChannelFactory<ICustomerManager> factory =
   new ChannelFactory<ICustomerManager>("customermanager");
ICustomerManager customerManager = factory.CreateChannel();
customerManager.StoreCustomer(customer);
```

The service returns an object by reference

For this scenario, the client app makes a call to the remote service and the method returns an object, which is passed by reference from the service to the client.

As mentioned previously, WCF services always return object by value. However, you can achieve a similar result by using the EndpointAddress10 class. The EndpointAddress10 is a serializable by-value object that can be used by the client to obtain a sessionful by-reference object on the server.

The behavior of the by-reference object in WCF shown in this scenario is different than DCOM. In DCOM, the server can return a by-reference object to the client directly, and the client can call that object's methods, which execute on the server. In WCF, however, the object returned is always by-value. The client must take that by-value object, represented by EndpointAddress10 and use it to create its own sessionful by-reference object. The client method calls on the sessionful object execute on the server.In other words, this by-reference object in WCF is a normal WCF service that is configured to be sessionful.

In WCF, a session is a way of correlating multiple messages sent between two endpoints. This means that once a client obtains a connection to this service, a session will be established between the client and the server. The client will use a single unique instance of the server-side object for all its interactions within this single session. Sessionful WCF contracts are similar to connection-oriented network request/response patterns.

This scenario is represented by the following DCOM method.

```
public interface IRemoteService
{
    IRemoteObject GetObjectByReference();
}
```

Step 1: Define the Sessionful WCF service interface and implementation

First, define a WCF service interface that contains the sessionful object.

In this code, the sessionful object is marked with the `ServiceContract` attribute, which identifies it as a regular WCF service interface. In addition, the SessionMode property is set to indicate it will be a sessionful service.

```
[ServiceContract(SessionMode = SessionMode.Allowed)]
public interface ISessionBoundObject
{
    [OperationContract]
    string GetCurrentValue();

    [OperationContract]
    void SetCurrentValue(string value);
}
```

The following code shows the service implementation.

The service is marked with the [ServiceBehavior] attribute, and its InstanceContextMode property set to InstanceContextMode.PerSessions to indicate that a unique instance of this type should be created for each session.

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
    public class MySessionBoundObject : ISessionBoundObject
    {
        private string _value;

        public string GetCurrentValue()
        {
            return _value;
        }

        public void SetCurrentValue(string val)
        {
            _value = val;
        }

    }
```

Step 2: Define the WCF factory service for the sessionful object

The service that creates the sessionful object must be defined and implemented. The following code shows how to do this. This code creates another WCF service that returns an EndpointAddress10 object. This is a serializable form of an endpoint the can use to create the session-full object.

```
[ServiceContract]
    public interface ISessionBoundFactory
    {
        [OperationContract]
        EndpointAddress10 GetInstanceAddress();
    }
```

Following is the implementation of this service. This implementation maintains a singleton channel factory to create sessionful objects. When `GetInstanceAddress` is called, it creates a channel and creates an EndpointAddress10 object that points to the remote address associated with this channel. EndpointAddress10 is a data type that can be returned to the client by-value.

```
public class SessionBoundFactory : ISessionBoundFactory
    {
        public static ChannelFactory<ISessionBoundObject> _factory =
            new ChannelFactory<ISessionBoundObject>("sessionbound");

        public SessionBoundFactory()
        {
        }

        public EndpointAddress10 GetInstanceAddress()
        {
            IClientChannel channel = (IClientChannel)_factory.CreateChannel();
            return EndpointAddress10.FromEndpointAddress(channel.RemoteAddress);
        }
    }
```

Step 3: Configure and start the WCF services

To host these services, you will need to make the following additions to the server's configuration file (web.config).

1. Add a `<client>` section that describes the endpoint for the sessionful object. In this scenario, the server also acts as a client and must be configured to enable this.

2. In the `<services>` section, declare service endpoints for the factory and sessionful object. This enables the client to communicate with the service endpoints, acquire the EndpointAddress10 and create the sessionful channel.

Following is an example configuration file with these settings:

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="sessionbound"
                address="net.tcp://localhost:8081/SessionBoundObject"
                binding="netTcpBinding"
                contract="Shared.ISessionBoundObject"/>
    </client>

    <services>
      <service name="Server.MySessionBoundObject">
        <endpoint address="net.tcp://localhost:8081/SessionBoundObject"
                  binding="netTcpBinding"
                  contract="Shared.ISessionBoundObject" />
      </service>
      <service name="Server.SessionBoundFactory">
        <endpoint address="net.tcp://localhost:8081/SessionBoundFactory"
                  binding="netTcpBinding"
                  contract="Shared.ISessionBoundFactory" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Add the following lines to a console application, to self-host the service, and start the app.

```
ServiceHost factoryHost = new ServiceHost(typeof(SessionBoundFactory));
factoryHost.Open();

ServiceHost sessionBoundServiceHost = new ServiceHost(
typeof(MySessionBoundObject));
sessionBoundServiceHost.Open();
```

Step 4: Configure the client and call the service

Configure the client to communicate with the WCF services by making the following entries in the project's application configuration file (app.config).

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="sessionbound"
                address="net.tcp://localhost:8081/SessionBoundObject"
                binding="netTcpBinding"
                contract="Shared.ISessionBoundObject"/>
      <endpoint name="factory"
                address="net.tcp://localhost:8081/SessionBoundFactory"
                binding="netTcpBinding"
                contract="Shared.ISessionBoundFactory"/>
    </client>
  </system.serviceModel>
</configuration>
```

To call the service, add the code to the client to do the following:

1. Create a channel to the `ISessionBoundFactory` service.
2. Use the channel to invoke the `ISessionBoundFactory` service an obtain an EndpointAddress10 object.
3. Use the EndpointAddress10 to create a channel to obtain a sessionful object.
4. Call the `SetCurrentValue` and `GetCurrentValue` methods to demonstrate it remains the same object

instance is used across multiple calls.

```
ChannelFactory<ISessionBoundFactory> factory =
        new ChannelFactory<ISessionBoundFactory>("factory");

ISessionBoundFactory sessionBoundFactory = factory.CreateChannel();

EndpointAddress10 address = sessionBoundFactory.GetInstanceAddress();

ChannelFactory<ISessionBoundObject> sessionBoundObjectFactory =
    new ChannelFactory<ISessionBoundObject>(
        new NetTcpBinding(),
        address.ToEndpointAddress());

ISessionBoundObject sessionBoundObject =
        sessionBoundObjectFactory.CreateChannel();

sessionBoundObject.SetCurrentValue("Hello");
if (sessionBoundObject.GetCurrentValue() == "Hello")
{
    Console.WriteLine("Session-full instance management works as expected");
}
```

See also

- Basic WCF Programming
- Designing and Implementing Services
- Building Clients
- Duplex Services

How to: Map HRESULTs and Exceptions

4/9/2019 • 3 minutes to read • Edit Online

COM methods report errors by returning HRESULTs; .NET methods report them by throwing exceptions. The runtime handles the transition between the two. Each exception class in the .NET Framework maps to an HRESULT.

User-defined exception classes can specify whatever HRESULT is appropriate. These exception classes can dynamically change the HRESULT to be returned when the exception is generated by setting the **HResult** field on the exception object. Additional information about the exception is provided to the client through the **IErrorInfo** interface, which is implemented on the .NET object in the unmanaged process.

If you create a class that extends **System.Exception**, you must set the HRESULT field during construction. Otherwise, the base class assigns the HRESULT value. You can map new exception classes to an existing HRESULT by supplying the value in the exception's constructor.

Note that the runtime will sometimes ignore an `HRESULT` in cases where there is an `IErrorInfo` present on the thread. This behavior can occur in cases where the `HRESULT` and the `IErrorInfo` do not represent the same error.

To create a new exception class and map it to an HRESULT

1. Use the following code to create a new exception class called `NoAccessException` and map it to the HRESULT `E_ACCESSDENIED`.

```
Class NoAccessException : public ApplicationException
{
    NoAccessException () {
    HResult = E_ACCESSDENIED;
}
}
CMyClass::MethodThatThrows
{
throw new NoAccessException();
}
```

You might encounter a program (in any programming language) that uses both managed and unmanaged code at the same time. For example, the custom marshaler in the following code example uses the **Marshal.ThrowExceptionForHR(int HResult)** method to throw an exception with a specific HRESULT value. The method looks up the HRESULT and generates the appropriate exception type. For example, the HRESULT in the following code fragment generates **ArgumentException**.

```
CMyClass::MethodThatThrows
{
    Marshal.ThrowExceptionForHR(COR_E_ARGUMENT);
}
```

The following table provides the complete mapping from each HRESULT to its comparable exception class in the .NET Framework.

HRESULT	.NET EXCEPTION
MSEE_E_APPDOMAINUNLOADED	**AppDomainUnloadedException**

HRESULT	.NET EXCEPTION
COR_E_APPLICATION	ApplicationException
COR_E_ARGUMENT or E_INVALIDARG	ArgumentException
COR_E_ARGUMENTOUTOFRANGE	ArgumentOutOfRangeException
COR_E_ARITHMETIC or ERROR_ARITHMETIC_OVERFLOW	ArithmeticException
COR_E_ARRAYTYPEMISMATCH	ArrayTypeMismatchException
COR_E_BADIMAGEFORMAT or ERROR_BAD_FORMAT	BadImageFormatException
COR_E_COMEMULATE_ERROR	COMEmulateException
COR_E_CONTEXTMARSHAL	ContextMarshalException
COR_E_CORE	CoreException
NTE_FAIL	CryptographicException
COR_E_DIRECTORYNOTFOUND or ERROR_PATH_NOT_FOUND	DirectoryNotFoundException
COR_E_DIVIDEBYZERO	DivideByZeroException
COR_E_DUPLICATEWAITOBJECT	DuplicateWaitObjectException
COR_E_ENDOFSTREAM	EndOfStreamException
COR_E_TYPELOAD	EntryPointNotFoundException
COR_E_EXCEPTION	Exception
COR_E_EXECUTIONENGINE	ExecutionEngineException
COR_E_FIELDACCESS	FieldAccessException
COR_E_FILENOTFOUND or ERROR_FILE_NOT_FOUND	FileNotFoundException
COR_E_FORMAT	FormatException
COR_E_INDEXOUTOFRANGE	IndexOutOfRangeException
COR_E_INVALIDCAST or E_NOINTERFACE	InvalidCastException
COR_E_INVALIDCOMOBJECT	InvalidComObjectException
COR_E_INVALIDFILTERCRITERIA	InvalidFilterCriteriaException
COR_E_INVALIDOLEVARIANTTYPE	InvalidOleVariantTypeException

HRESULT	.NET EXCEPTION
COR_E_INVALIDOPERATION	**InvalidOperationException**
COR_E_IO	**IOException**
COR_E_MEMBERACCESS	**AccessException**
COR_E_METHODACCESS	**MethodAccessException**
COR_E_MISSINGFIELD	**MissingFieldException**
COR_E_MISSINGMANIFESTRESOURCE	**MissingManifestResourceException**
COR_E_MISSINGMEMBER	**MissingMemberException**
COR_E_MISSINGMETHOD	**MissingMethodException**
COR_E_MULTICASTNOTSUPPORTED	**MulticastNotSupportedException**
COR_E_NOTFINITENUMBER	**NotFiniteNumberException**
E_NOTIMPL	**NotImplementedException**
COR_E_NOTSUPPORTED	**NotSupportedException**
COR_E_NULLREFERENCE orE_POINTER	**NullReferenceException**
COR_E_OUTOFMEMORY or E_OUTOFMEMORY	**OutOfMemoryException**
COR_E_OVERFLOW	**OverflowException**
COR_E_PATHTOOLONG or ERROR_FILENAME_EXCED_RANGE	**PathTooLongException**
COR_E_RANK	**RankException**
COR_E_REFLECTIONTYPELOAD	**ReflectionTypeLoadException**
COR_E_REMOTING	**RemotingException**
COR_E_SAFEARRAYTYPEMISMATCH	**SafeArrayTypeMismatchException**
COR_E_SECURITY	**SecurityException**
COR_E_SERIALIZATION	**SerializationException**
COR_E_STACKOVERFLOW orERROR_STACK_OVERFLOW	**StackOverflowException**
COR_E_SYNCHRONIZATIONLOCK	**SynchronizationLockException**

HRESULT	.NET EXCEPTION
COR_E_SYSTEM	**SystemException**
COR_E_TARGET	**TargetException**
COR_E_TARGETINVOCATION	**TargetInvocationException**
COR_E_TARGETPARAMCOUNT	**TargetParameterCountException**
COR_E_THREADABORTED	**ThreadAbortException**
COR_E_THREADINTERRUPTED	**ThreadInterruptedException**
COR_E_THREADSTATE	**ThreadStateException**
COR_E_THREADSTOP	**ThreadStopException**
COR_E_TYPELOAD	**TypeLoadException**
COR_E_TYPEINITIALIZATION	**TypeInitializationException**
COR_E_VERIFICATION	**VerificationException**
COR_E_WEAKREFERENCE	**WeakReferenceException**
COR_E_VTABLECALLSNOTSUPPORTED	**VTableCallsNotSupportedException**
All other HRESULTs	**COMException**

To retrieve extended error information, the managed client must examine the fields of the exception object that was generated. For the exception object to provide useful information about an error, the COM object must implement the **IErrorInfo** interface. The runtime uses the information provided by **IErrorInfo** to initialize the exception object.

If the COM object does not support **IErrorInfo**, the runtime initializes an exception object with default values. The following table lists each field associated with an exception object and identifies the source of default information when the COM object supports **IErrorInfo**.

Note that the runtime will sometimes ignore an `HRESULT` in cases where there is an `IErrorInfo` present on the thread. This behavior can occur in cases where the `HRESULT` and the `IErrorInfo` do not represent the same error.

EXCEPTION FIELD	SOURCE OF INFORMATION FROM COM
ErrorCode	HRESULT returned from call.
HelpLink	If **IErrorInfo->HelpContext** is nonzero, the string is formed by concatenating **IErrorInfo->GetHelpFile** and "#" and **IErrorInfo->GetHelpContext**. Otherwise the string is returned from **IErrorInfo->GetHelpFile**.
InnerException	Always a null reference (**Nothing** in Visual Basic).

EXCEPTION FIELD	SOURCE OF INFORMATION FROM COM
Message	String returned from **IErrorInfo->GetDescription**.
Source	String returned from **IErrorInfo->GetSource**.
StackTrace	The stack trace.
TargetSite	The name of the method that returned the failing HRESULT.

Exception fields, such as **Message**, **Source**, and **StackTrace** are not available for the **StackOverflowException**.

See also

- Advanced COM Interoperability
- Exceptions

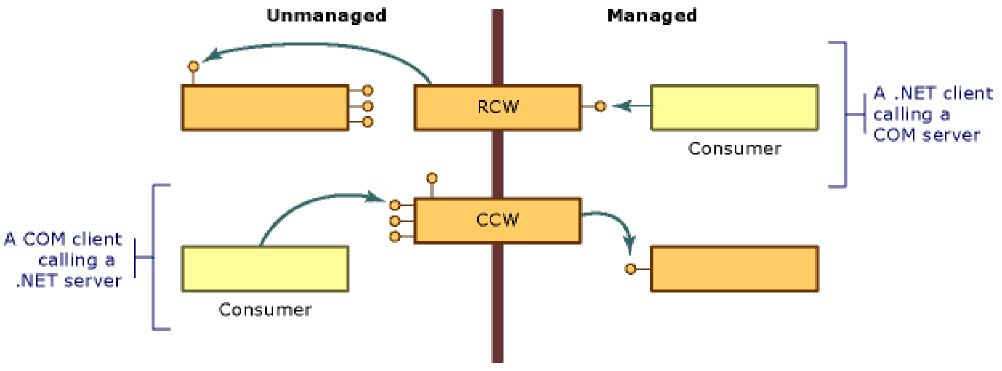
COM Wrappers

4/28/2019 • 2 minutes to read • Edit Online

COM differs from the .NET Framework object model in several important ways:

- Clients of COM objects must manage the lifetime of those objects; the common language runtime manages the lifetime of objects in its environment.
- Clients of COM objects discover whether a service is available by requesting an interface that provides that service and getting back an interface pointer, or not. Clients of .NET objects can obtain a description of an object's functionality using reflection.
- NET objects reside in memory managed by the .NET Framework execution environment. The execution environment can move objects around in memory for performance reasons and update all references to the objects it moves. Unmanaged clients, having obtained a pointer to an object, rely on the object to remain at the same location. These clients have no mechanism for dealing with an object whose location is not fixed.

To overcome these differences, the runtime provides wrapper classes to make both managed and unmanaged clients think they are calling objects within their respective environment. Whenever your managed client calls a method on a COM object, the runtime creates a runtime callable wrapper (RCW). RCWs abstract the differences between managed and unmanaged reference mechanisms, among other things. The runtime also creates a COM callable wrapper (CCW) to reverse the process, enabling a COM client to seamlessly call a method on a .NET object. As the following illustration shows, the perspective of the calling code determines which wrapper class the runtime creates.


Unmanaged
Managed
RCW
Consumer
A .NET client calling a COM server
CCW
A COM client calling a .NET server
Consumer

In most cases, the standard RCW or CCW generated by the runtime provides adequate marshaling for calls that cross the boundary between COM and the .NET Framework. Using custom attributes, you can optionally adjust the way the runtime represents managed and unmanaged code.

See also

- Advanced COM Interoperability
- Runtime Callable Wrapper
- COM Callable Wrapper
- Customizing Standard Wrappers
- How to: Customize Runtime Callable Wrappers

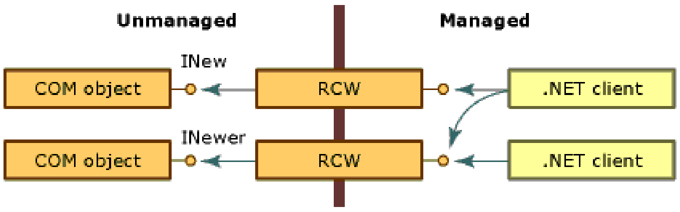
Runtime Callable Wrapper

4/8/2019 • 3 minutes to read • Edit Online

The common language runtime exposes COM objects through a proxy called the runtime callable wrapper (RCW). Although the RCW appears to be an ordinary object to .NET clients, its primary function is to marshal calls between a .NET client and a COM object.

The runtime creates exactly one RCW for each COM object, regardless of the number of references that exist on that object. The runtime maintains a single RCW per process for each object. If you create an RCW in one application domain or apartment, and then pass a reference to another application domain or apartment, a proxy to the first object will be used. As the following illustration shows, any number of managed clients can hold a reference to the COM objects that expose INew and INewer interfaces.

The following image shows the process for accessing COM objects through the runtime callable wrapper:



Using metadata derived from a type library, the runtime creates both the COM object being called and a wrapper for that object. Each RCW maintains a cache of interface pointers on the COM object it wraps and releases its reference on the COM object when the RCW is no longer needed. The runtime performs garbage collection on the RCW.

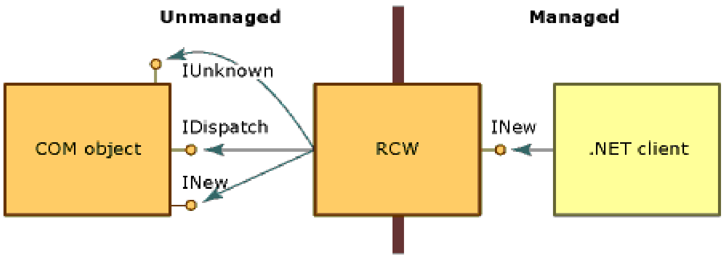
Among other activities, the RCW marshals data between managed and unmanaged code, on behalf of the wrapped object. Specifically, the RCW provides marshaling for method arguments and method return values whenever the client and server have different representations of the data passed between them.

The standard wrapper enforces built-in marshaling rules. For example, when a .NET client passes a String type as part of an argument to an unmanaged object, the wrapper converts the string to a BSTR type. Should the COM object return a BSTR to its managed caller, the caller receives a String. Both the client and the server send and receive data that is familiar to them. Other types require no conversion. For instance, a standard wrapper will always pass a 4-byte integer between managed and unmanaged code without converting the type.

Marshaling selected interfaces

The primary goal of the runtime callable wrapper (RCW) is to hide the differences between the managed and unmanaged programming models. To create a seamless transition, the RCW consumes selected COM interfaces without exposing them to the .NET client, as shown in the following illustration.

The following image shows COM interfaces and the runtime callable wrapper:



When created as an early-bound object, the RCW is a specific type. It implements the interfaces that the COM object implements and exposes the methods, properties, and events from the object's interfaces. In the illustration, the RCW exposes the INew interface but consumes the **IUnknown** and **IDispatch** interfaces. Further, the RCW exposes all members of the INew interface to the .NET client.

The RCW consumes the interfaces listed in the following table, which are exposed by the object it wraps.

INTERFACE	DESCRIPTION
IDispatch	For late binding to COM objects through reflection.
IErrorInfo	Provides a textual description of the error, its source, a Help file, Help context, and the GUID of the interface that defined the error (always **GUID_NULL** for .NET classes).
IProvideClassInfo	If the COM object being wrapped implements **IProvideClassInfo**, the RCW extracts the type information from this interface to provide better type identity.
IUnknown	For object identity, type coercion, and lifetime management: - Object identity The runtime distinguishes between COM objects by comparing the value of the **IUnknown** interface for each object. - Type coercion The RCW recognizes the dynamic type discovery performed by the **QueryInterface** method. - Lifetime management Using the **QueryInterface** method, the RCW gets and holds a reference to an unmanaged object until the runtime performs garbage collection on the wrapper, which releases the unmanaged object.

The RCW optionally consumes the interfaces listed in the following table, which are exposed by the object it wraps.

INTERFACE	DESCRIPTION
IConnectionPoint and **IConnectionPointContainer**	The RCW converts objects that expose the connection-point event style to delegate-based events.
IDispatchEx	If the class implements **IDispatchEx**, the RCW implements **IExpando**. The **IDispatchEx** interface is an extension of the **IDispatch** interface that, unlike **IDispatch**, enables enumeration, addition, deletion, and case-sensitive calling of members.
IEnumVARIANT	Enables COM types that support enumerations to be treated as collections.

See also

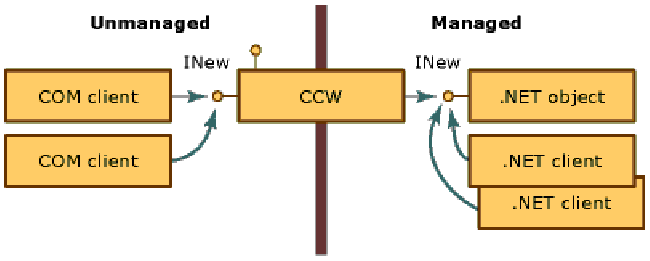
- COM Wrappers
- COM Callable Wrapper
- Type Library to Assembly Conversion Summary
- Importing a Type Library as an Assembly

COM Callable Wrapper

4/16/2019 • 8 minutes to read • Edit Online

When a COM client calls a .NET object, the common language runtime creates the managed object and a COM callable wrapper (CCW) for the object. Unable to reference a .NET object directly, COM clients use the CCW as a proxy for the managed object.

The runtime creates exactly one CCW for a managed object, regardless of the number of COM clients requesting its services. As the following illustration shows, multiple COM clients can hold a reference to the CCW that exposes the INew interface. The CCW, in turn, holds a single reference to the managed object that implements the interface and is garbage collected. Both COM and .NET clients can make requests on the same managed object simultaneously.



COM callable wrappers are invisible to other classes running within the .NET Framework. Their primary purpose is to marshal calls between managed and unmanaged code; however, CCWs also manage the object identity and object lifetime of the managed objects they wrap.

Object Identity

The runtime allocates memory for the .NET object from its garbage-collected heap, which enables the runtime to move the object around in memory as necessary. In contrast, the runtime allocates memory for the CCW from a noncollected heap, making it possible for COM clients to reference the wrapper directly.

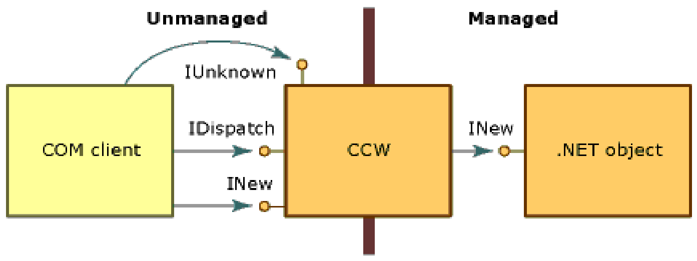
Object Lifetime

Unlike the .NET client it wraps, the CCW is reference-counted in traditional COM fashion. When the reference count on the CCW reaches zero, the wrapper releases its reference on the managed object. A managed object with no remaining references is collected during the next garbage-collection cycle.

Simulating COM interfaces

CCW exposes all public, COM-visible interfaces, data types, and return values to COM clients in a manner that is consistent with COM's enforcement of interface-based interaction. For a COM client, invoking methods on a .NET Framework object is identical to invoking methods on a COM object.

To create this seamless approach, the CCW manufactures traditional COM interfaces, such as **IUnknown** and **IDispatch**. As the following illustration shows, the CCW maintains a single reference on the .NET object that it wraps. Both the COM client and the .NET object interact with each other through the proxy and stub construction of the CCW.



In addition to exposing the interfaces that are explicitly implemented by a class in the managed environment, the .NET Framework supplies implementations of the COM interfaces listed in the following table on behalf of the object. A .NET class can override the default behavior by providing its own implementation of these interfaces. However, the runtime always provides the implementation for the **IUnknown** and **IDispatch** interfaces.

INTERFACE	DESCRIPTION
IDispatch	Provides a mechanism for late binding to type.
IErrorInfo	Provides a textual description of the error, its source, a Help file, Help context, and the GUID of the interface that defined the error (always **GUID_NULL** for .NET classes).
IProvideClassInfo	Enables COM clients to gain access to the **ITypeInfo** interface implemented by a managed class.
ISupportErrorInfo	Enables a COM client to determine whether the managed object supports the **IErrorInfo** interface. If so, enables the client to obtain a pointer to the latest exception object. All managed types support the **IErrorInfo** interface.
ITypeInfo	Provides type information for a class that is exactly the same as the type information produced by Tlbexp.exe.
IUnknown	Provides the standard implementation of the **IUnknown** interface with which the COM client manages the lifetime of the CCW and provides type coercion.

A managed class can also provide the COM interfaces described in the following table.

INTERFACE	DESCRIPTION
The (_*classname*) class interface	Interface, exposed by the runtime and not explicitly defined, that exposes all public interfaces, methods, properties, and fields that are explicitly exposed on a managed object.
IConnectionPoint and **IConnectionPointContainer**	Interface for objects that source delegate-based events (an interface for registering event subscribers).
IDispatchEx	Interface supplied by the runtime if the class implements **IExpando**. The **IDispatchEx** interface is an extension of the **IDispatch** interface that, unlike **IDispatch**, enables enumeration, addition, deletion, and case-sensitive calling of members.
IEnumVARIANT	Interface for collection-type classes, which enumerates the objects in the collection if the class implements **IEnumerable**.

Introducing the class interface

The class interface, which is not explicitly defined in managed code, is an interface that exposes all public methods, properties, fields, and events that are explicitly exposed on the .NET object. This interface can be a dual or dispatch-only interface. The class interface receives the name of the .NET class itself, preceded by an underscore. For example, for class Mammal, the class interface is _Mammal.

For derived classes, the class interface also exposes all public methods, properties, and fields of the base class. The derived class also exposes a class interface for each base class. For example, if class Mammal extends class MammalSuperclass, which itself extends System.Object, the .NET object exposes to COM clients three class interfaces named _Mammal, _MammalSuperclass, and _Object.

For example, consider the following .NET class:

```
' Applies the ClassInterfaceAttribute to set the interface to dual.
<ClassInterface(ClassInterfaceType.AutoDual)> _
' Implicitly extends System.Object.
Public Class Mammal
    Sub Eat()
    Sub Breathe()
    Sub Sleep()
End Class
```

```
// Applies the ClassInterfaceAttribute to set the interface to dual.
[ClassInterface(ClassInterfaceType.AutoDual)]
// Implicitly extends System.Object.
public class Mammal
{
    public void Eat() {}
    public void Breathe() {}
    public void Sleep() {}
}
```

The COM client can obtain a pointer to a class interface named `_Mammal`, which is described in the type library that the Type Library Exporter (Tlbexp.exe) tool generates. If the `Mammal` class implemented one or more interfaces, the interfaces would appear under the coclass.

```
[odl, uuid(…), hidden, dual, nonextensible, oleautomation]
interface _Mammal : IDispatch
{
    [id(0x00000000), propget] HRESULT ToString([out, retval] BSTR*
        pRetVal);
    [id(0x60020001)] HRESULT Equals([in] VARIANT obj, [out, retval]
        VARIANT_BOOL* pRetVal);
    [id(0x60020002)] HRESULT GetHashCode([out, retval] short* pRetVal);
    [id(0x60020003)] HRESULT GetType([out, retval] _Type** pRetVal);
    [id(0x6002000d)] HRESULT Eat();
    [id(0x6002000e)] HRESULT Breathe();
    [id(0x6002000f)] HRESULT Sleep();
}
[uuid(…)]
coclass Mammal
{
    [default] interface _Mammal;
}
```

Generating the class interface is optional. By default, COM interop generates a dispatch-only interface for each class you export to a type library. You can prevent or modify the automatic creation of this interface by applying the ClassInterfaceAttribute to your class. Although the class interface can ease the task of exposing managed classes to COM, its uses are limited.

Caution

Using the class interface, instead of explicitly defining your own, can complicate the future versioning of your managed class. Please read the following guidelines before using the class interface.

Define an explicit interface for COM clients to use rather than generating the class interface.

Because COM interop generates a class interface automatically, post-version changes to your class can alter the layout of the class interface exposed by the common language runtime. Since COM clients are typically unprepared to handle changes in the layout of an interface, they break if you change the member layout of the class.

This guideline reinforces the notion that interfaces exposed to COM clients must remain unchangeable. To reduce the risk of breaking COM clients by inadvertently reordering the interface layout, isolate all changes to the class from the interface layout by explicitly defining interfaces.

Use the **ClassInterfaceAttribute** to disengage the automatic generation of the class interface and implement an explicit interface for the class, as the following code fragment shows:

```
<ClassInterface(ClassInterfaceType.None)>Public Class LoanApp
    Implements IExplicit
    Sub M() Implements IExplicit.M
…
End Class
```

```
[ClassInterface(ClassInterfaceType.None)]
public class LoanApp : IExplicit
{
    int IExplicit.M() { return 0; }
}
```

The **ClassInterfaceType.None** value prevents the class interface from being generated when the class metadata is exported to a type library. In the preceding example, COM clients can access the `LoanApp` class only through the `IExplicit` interface.

Avoid caching dispatch identifiers (DispIds)

Using the class interface is an acceptable option for scripted clients, Microsoft Visual Basic 6.0 clients, or any late-bound client that does not cache the DispIds of interface members. DispIds identify interface members to enable late binding.

For the class interface, generation of DispIds is based on the position of the member in the interface. If you change the order of the member and export the class to a type library, you will alter the DispIds generated in the class interface.

To avoid breaking late-bound COM clients when using the class interface, apply the **ClassInterfaceAttribute** with the **ClassInterfaceType.AutoDispatch** value. This value implements a dispatch-only class interface, but omits the interface description from the type library. Without an interface description, clients are unable to cache DispIds at compile time. Although this is the default interface type for the class interface, you can apply the attribute value explicitly.

```
<ClassInterface(ClassInterfaceType.AutoDispatch)> Public Class LoanApp
    Implements IAnother
    Sub M() Implements IAnother.M
…
End Class
```

```
[ClassInterface(ClassInterfaceType.AutoDispatch)]
public class LoanApp
{
    public int M() { return 0; }
}
```

To get the DispId of an interface member at run time, COM clients can call **IDispatch.GetIdsOfNames**. To invoke a method on the interface, pass the returned DispId as an argument to **IDispatch.Invoke**.

Restrict using the dual interface option for the class interface.

Dual interfaces enable early and late binding to interface members by COM clients. At design time and during testing, you might find it useful to set the class interface to dual. For a managed class (and its base classes) that will never be modified, this option is also acceptable. In all other cases, avoid setting the class interface to dual.

An automatically generated dual interface might be appropriate in rare cases; however, more often it creates version-related complexity. For example, COM clients using the class interface of a derived class can easily break with changes to the base class. When a third party provides the base class, the layout of the class interface is out of your control. Further, unlike a dispatch-only interface, a dual interface (**ClassInterfaceType.AutoDual**) provides a description of the class interface in the exported type library. Such a description encourages late-bound clients to cache DispIds at compile time.

Ensure that all COM event notifications are late-bound.

By default, COM type information is embedded directly into managed assemblies, which eliminates the need for primary interop assemblies (PIAs). However, one of the limitations of embedded type information is that it does not supported delivery of COM event notifications by early-bound vtable calls, but only supports late-bound `IDispatch::Invoke` calls.

If your application requires early-bound calls to COM event interface methods, you can set the **Embed Interop Types** property in Visual Studio to `true`, or include the following element in your project file:

```
<EmbedInteropTypes>True</EmbedInteropTypes>
```

See also

- ClassInterfaceAttribute
- COM Wrappers
- Exposing .NET Framework Components to COM
- Qualifying .NET Types for Interoperation
- Runtime Callable Wrapper

How to: Create COM Wrappers

5/9/2019 • 2 minutes to read • Edit Online

You can create Component Object Model (COM) wrappers by using Visual Studio 2005 features or the .NET Framework tools Tlbimp.exe and Regasm.exe. Both methods generate two types of COM wrappers:

- A Runtime Callable Wrapper from a type library to run a COM object in managed code.
- A COM Callable Wrapper with the required registry settings to run a managed object in a native application.

In Visual Studio 2005, you can add the COM wrapper as a reference to your project.

Wrap COM Objects in a Managed Application

To create a runtime callable wrapper using Visual Studio

1. Open the project for your managed application.
2. On the **Project** menu, click **Show All Files**.
3. On the **Project** menu, click **Add Reference**.
4. In the Add Reference dialog box, click the **COM** tab, select the component you want to use, and click **OK**.

   In **Solution Explorer**, note that the COM component is added to the References folder in your project.

You can now write code to access the COM object. You can begin by declaring the object, such as with an `Imports` statement for Visual Basic or a `Using` statement for C#.

NOTE

If you want to program Microsoft Office components, first install the Microsoft Office Primary Interop Assemblies (PIAs) from the Microsoft Download Center. In step 4, select the latest version of the object library available for the Office product you want, such as the **Microsoft Word 11.0 Object Library**.

To create a runtime callable wrapper using .NET Framework tools

- Run the Tlbimp.exe (Type Library Importer) tool.

This tool creates an assembly that contains run-time metadata for the types defined in the original type library.

Wrap Managed Objects in a Native Application

To create a COM callable wrapper using Visual Studio

1. Create a Class Library project for the managed class that you want to run in native code. The class must have a default constructor.

   Verify that you have a complete four-part version number for your assembly in the AssemblyInfo file. This number is required for maintaining versioning in the Windows registry. For more information about version numbers, see Assembly Versioning.
2. On the **Project** menu, click **Properties**.
3. Click the **Compile** tab.
4. Select the **Register for COM interop** check box.

When you build the project, the assembly is automatically registered for COM interop. If you are building a native application in Visual Studio 2005, you can use the assembly by clicking **Add Reference** on the **Project** menu.

To create a COM callable wrapper using .NET Framework tools

Run the Regasm.exe (Assembly Registration Tool) tool.

This tool reads the assembly metadata and adds the necessary entries to the registry. As a result, COM clients can create .NET Framework classes transparently. You can use the assembly as if it were a native COM class.

You can run Regasm.exe on an assembly located in any directory, and then run the Gacutil.exe (Global Assembly Cache Tool) to move it to the global assembly cache. Moving the assembly does not invalidate location registry entries, because the global assembly cache is always examined if the assembly is not found elsewhere.

See also

- Runtime Callable Wrapper
- COM Callable Wrapper

Type equivalence and embedded interop types

2/13/2019 • 2 minutes to read • Edit Online

Beginning with the .NET Framework 4, the common language runtime supports embedding type information for COM types directly into managed assemblies, instead of requiring the managed assemblies to obtain type information for COM types from interop assemblies. Because the embedded type information includes only the types and members that are actually used by a managed assembly, two managed assemblies might have very different views of the same COM type. Each managed assembly has a different Type object to represent its view of the COM type. The common language runtime supports type equivalence between these different views for interfaces, structures, enumerations, and delegates.

Type equivalence means that a COM object that is passed from one managed assembly to another can be cast to the appropriate managed type in the receiving assembly.

NOTE

Type equivalence and embedded interop types simplify the deployment of applications and add-ins that use COM components, because it is not necessary to deploy interop assemblies with the applications. Developers of shared COM components still have to create primary interop assemblies (PIAs) if they want their components to be used by earlier versions of the .NET Framework.

Type equivalence

Equivalence of COM types is supported for interfaces, structures, enumerations, and delegates. COM types qualify as equivalent if all of the following are true:

- The types are both interfaces, or both structures, or both enumerations, or both delegates.
- The types have the same identity, as described in the next section.
- Both types are eligible for type equivalence, as described in the Marking COM types for type equivalence section.

Type identity

Two types are determined to have the same identity when their scopes and identities match, in other words, if they each have the TypeIdentifierAttribute attribute, and the two attributes have matching Scope and Identifier properties. The comparison for Scope is case-insensitive.

If a type does not have the TypeIdentifierAttribute attribute, or if it has a TypeIdentifierAttribute attribute that does not specify scope and identifier, the type can still be considered for equivalence as follows:

- For interfaces, the value of the GuidAttribute is used instead of the TypeIdentifierAttribute.Scope property, and the Type.FullName property (that is, the type name, including the namespace) is used instead of the TypeIdentifierAttribute.Identifier property.
- For structures, enumerations, and delegates, the GuidAttribute of the containing assembly is used instead of the Scope property, and the Type.FullName property is used instead of the Identifier property.

Marking COM types for type equivalence

You can mark a type as eligible for type equivalence in two ways:

- Apply the TypeIdentifierAttribute attribute to the type.

- Make the type a COM import type. An interface is a COM import type if it has the ComImportAttribute attribute. An interface, structure, enumeration, or delegate is a COM import type if the assembly in which it is defined has the ImportedFromTypeLibAttribute attribute.

See also

- IsEquivalentTo
- Using COM Types in Managed Code
- Importing a Type Library as an Assembly

How to: Generate Primary Interop Assemblies Using Tlbimp.exe

4/28/2019 • 2 minutes to read • Edit Online

There are two ways to generate a primary interop assembly:

- Using the Type Library Importer (Tlbimp.exe) provided by the Windows Software Development Kit (SDK).

  The most straightforward way to produce primary interop assemblies is to use the Tlbimp.exe (Type Library Importer). Tlbimp.exe provides the following safeguards:

  - Checks for other registered primary interop assemblies before creating new interop assemblies for any nested type library references.
  - Fails to emit the primary interop assembly if you do not specify either the container or file name to give the primary interop assembly a strong name.
  - Fails to emit a primary interop assembly if you omit references to dependent assemblies.
  - Fails to emit a primary interop assembly if you add references to dependent assemblies that are not primary interop assemblies.
- Creating primary interop assemblies manually in source code by using a language that is compliant with the Common Language Specification (CLS), such as C#. This approach is useful when a type library is unavailable.

You must have a cryptographic key pair to sign the assembly with a strong name. For details, see Creating A Key Pair.

To generate a primary interop assembly using Tlbimp.exe

1. At the command prompt, type:

   tlbimp *tlbfile* **/primary /keyfile:** *filename* **/out:** *assemblyname*

   In this command, *tlbfile* is the file containing the COM type library, *filename* is the name of the container or file that contains the key pair, and *assemblyname* is the name of the assembly to sign with a strong name.

Primary interop assemblies can reference only other primary interop assemblies. If your assembly references types from a third-party COM type library, you must obtain a primary interop assembly from the publisher before you can generate your primary interop assembly. If you are the publisher, you must generate a primary interop assembly for the dependent type library before generating the referencing primary interop assembly.

A dependent primary interop assembly with a version number that differs from that of the original type library is not discoverable when installed in the current directory. You must either register the dependent primary interop assembly in the Windows registry or use the **/reference** option to be sure that Tlbimp.exe finds the dependent DLL.

You can also wrap multiple versions of a type library. For instructions, see How to: Wrap Multiple Versions of Type Libraries.

Example

The following example imports the COM type library `LibUtil.tlb` and signs the assembly `LibUtil.dll` with a strong name using the key file `CompanyA.snk`. By omitting a specific namespace name, this example produces the default namespace, `LibUtil`.

```
tlbimp LibUtil.tlb /primary /keyfile:CompanyA.snk /out:LibUtil.dll
```

For a more descriptive name (using the *VendorName.LibraryName* naming guideline), the following example overrides the default assembly file name and namespace name.

```
tlbimp LibUtil.tlb /primary /keyfile:CompanyA.snk /namespace:CompanyA.LibUtil /out:CompanyA.LibUtil.dll
```

The following example imports `MyLib.tlb`, which references `CompanyA.LibUtil.dll`, and signs the assembly `CompanyB.MyLib.dll` with a strong name using the key file `CompanyB.snk`. The namespace, `CompanyB.MyLib`, overrides the default namespace name.

```
tlbimp MyLib.tlb /primary /keyfile:CompanyB.snk /namespace:CompanyB.MyLib /reference:CompanyA.LibUtil.dll
/out:CompanyB.MyLib.dll
```

See also

- How to: Register Primary Interop Assemblies

How to: Register Primary Interop Assemblies

5/15/2019 • 2 minutes to read • Edit Online

Classes can be marshaled only by COM interop and are always marshaled as interfaces. In some cases the interface used to marshal the class is known as the class interface. For information about overriding the class interface with an interface of your choice, see COM Callable Wrapper.

Although any developer who wants to use COM types from a .NET Framework application can generate an interop assembly, doing so creates a problem. Each time a developer imports and signs a COM type library, that developer creates a set of unique types that are incompatible with those imported and signed by another developer. The solution to this type incompatibility problem is for each developer to obtain the vendor-supplied and signed primary interop assembly.

If you plan to expose third-party COM types to other applications, always use the primary interop assembly provided by the same publisher as the type library it defines. In addition to providing guaranteed type compatibility, primary interop assemblies are often customized by the vendor to enhance interoperability.

Even if you do not plan to expose third-party COM types, using the primary interop assembly can ease the task of interoperating with COM components. However, this strategy provides no insulation from changes a vendor might make to types defined in a primary interop assembly. When your application requires such insulation, generate your own interop assembly instead of using the primary interop assembly.

You must register all acquired primary interop assemblies on your development computer before you can reference them with Visual Studio. Visual Studio looks for and uses a primary interop assembly the first time that you reference a type from a COM type library. If Visual Studio cannot locate the primary interop assembly associated with the type library, it prompts you to acquire it or offers to create an interop assembly instead. Likewise, the Type Library Importer (Tlbimp.exe) also uses the registry to locate primary interop assemblies.

Although it is not necessary to register primary interop assemblies unless you plan to use Visual Studio, registration provides two advantages:

- A registered primary interop assembly is clearly marked under the registry key of the original type library. Registration is the best way for you to locate a primary interop assembly on your computer.
- You can avoid accidentally generating and using a new interop assembly if, at some time in the future, you do use Visual Studio to reference a type for which you have an unregistered primary interop assembly.

Use the Assembly Registration Tool (Regasm.exe) to register a primary interop assembly.

To register a primary interop assembly

1. At the command prompt, type:

   regasm *assemblyname*

   In this command, *assemblyname* is the file name of the assembly that is registered. Regasm.exe adds an entry for the primary interop assembly under the same registry key as the original type library.

Example

The following example registers the `CompanyA.UtilLib.dll` primary interop assembly.

```
regasm CompanyA.UtilLib.dll
```

See also

- Programming with Primary Interop Assemblies
- Locating Primary Interop Assemblies
- Redistributing Primary Interop Assemblies

Registration-Free COM Interop

4/28/2019 • 2 minutes to read • Edit Online

Registration-free COM interop activates a component without using the Windows registry to store assembly information. Instead of registering a component on a computer during deployment, you create Win32-style manifest files at design time that contain information about binding and activation. These manifest files, rather than registry keys, direct the activation of an object.

Using registration-free activation for your assemblies instead of registering them during deployment offers two advantages:

- You can control which DLL version is activated when more than one version is installed on a computer.
- End users can use XCOPY or FTP to copy your application to an appropriate directory on their computer. The application can then be run from that directory.

This section describes the two types of manifests needed for registration-free COM interop: application and component manifests. These manifests are XML files. An application manifest, which is created by an application developer, contains metadata that describes assemblies and assembly dependencies. A component manifest, created by a component developer, contains information otherwise located in the Windows registry.

Requirements for registration-free COM interop

1. Support for registration-free COM interop varies slightly depending on the type of library assembly; specifically, whether the assembly is unmanaged (COM side-by-side) or managed (.NET-based). The following table shows the operating system and .NET Framework version requirements for each assembly type.

ASSEMBLY TYPE	OPERATING SYSTEM	.NET FRAMEWORK VERSION
COM side-by-side	Microsoft Windows XP	Not required.
.NET-based	Windows XP with SP2	NET Framework version 1.1 or later.

   The Windows Server 2003 family also supports registration-free COM interop for .NET-based assemblies.

   For a .NET-based class to be compatible with registry-free activation from COM, the class must have a default constructor and must be public.

Configuring COM components for registration-free activation

1. For a COM component to participate in registration-free activation, it must be deployed as a side-by-side assembly. Side-by-side assemblies are unmanaged assemblies. For more information, see Using Side-by-side Assemblies.

   To use COM side-by-side assemblies, a .NET-based application developer must provide an application manifest, which contains the binding and activation information. Support for unmanaged side-by-side assemblies is built into the Windows XP operating system. The COM runtime, supported by the operating system, scans an application manifest for activation information when the component being activated is not in the registry.

   Registration-free activation is optional for COM components installed on Windows XP. For detailed instructions on adding a side-by-side assembly to an application, see Using Side-by-side Assemblies.

NOTE

Side-by-side execution is a .NET Framework feature that enables multiple versions of the runtime, and multiple versions of applications and components that use a version of the runtime, to run on the same computer at the same time. Side-by-side execution and side-by-side assemblies are different mechanisms for providing side-by-side functionality.

See also

- How to: Configure .NET Framework-Based COM Components for Registration-Free Activation

How to: Configure .NET Framework-Based COM Components for Registration-Free Activation

4/28/2019 • 4 minutes to read • Edit Online

Registration-free activation for .NET Framework-based components is only slightly more complicated than it is for COM components. The setup requires two manifests:

- COM applications must have a Win32-style application manifest to identify the managed component.
- .NET Framework-based components must have a component manifest for activation information needed at run time.

This topic describes how to associate an application manifest with an application; associate a component manifest with a component; and embed a component manifest in an assembly.

To create an application manifest

1. Using an XML editor, create (or modify) the application manifest owned by the COM application that is interoperating with one or more managed components.

2. Insert the following standard header at the beginning of the file:

   ```
   <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
   <assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
   ```

   For information about manifest elements and their attributes, see Application Manifests.

3. Identify the owner of the manifest. In the following example, `myComApp` version 1 owns the manifest file.

   ```
   <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
   <assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
     <assemblyIdentity type="win32"
                       name="myOrganization.myDivision.myComApp"
                       version="1.0.0.0"
                       processorArchitecture="msil"
     />
   ```

4. Identify dependent assemblies. In the following example, `myComApp` depends on `myManagedComp`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity type="win32"
                  name="myOrganization.myDivision.myComApp"
                  version="1.0.0.0"
                  processorArchitecture="x86"
                  publicKeyToken="8275b28176rcbbef"
  />
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32"
                  name="myOrganization.myDivision.myManagedComp"
                  version="6.0.0.0"
                  processorArchitecture="X86"
                  publicKeyToken="8275b28176rcbbef"
      />
    </dependentAssembly>
  </dependency>
</assembly>
```

5. Save and name the manifest file. The name of an application manifest is the name of the assembly executable followed by the .manifest extension. For example, the application manifest file name for myComApp.exe is myComApp.exe.manifest.

You can install an application manifest in the same directory as the COM application. Alternatively, you can add it as a resource to the application's .exe file. For additional information, For more information, see About Side-by-Side Assemblies.

To create a component manifest

1. Using an XML editor, create a component manifest to describe the managed assembly.

2. Insert the following standard header at the beginning of the file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
```

3. Identify the owner of the file. The `<assemblyIdentity>` element of the `<dependentAssembly>` element in application manifest file must match the one in the component manifest. In the following example, `myManagedComp` version 1.2.3.4 owns the manifest file.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
       <assemblyIdentity
                  name="myOrganization.myDivision.myManagedComp"
                  version="1.2.3.4"
                  publicKeyToken="8275b28176rcbbef"
                  processorArchitecture="msil"
       />
```

4. Identify each class in the assembly. Use the `<clrClass>` element to uniquely identify each class in the managed assembly. The element, which is a subelement of the `<assembly>` element, has the attributes described in the following table.

ATTRIBUTE	DESCRIPTION	REQUIRED
`clsid`	The identifier that specifies the class to be activated.	Yes

ATTRIBUTE	DESCRIPTION	REQUIRED
`description`	A string that informs the user about the component. An empty string is the default.	No
`name`	A string that represents the managed class.	Yes
`progid`	The identifier to be used for late-bound activation.	No
`threadingModel`	The COM threading model. "Both" is the default value.	No
`runtimeVersion`	Specifies the common language runtime (CLR) version to use. If you do not specify this attribute, and the CLR is not already loaded, the component is loaded with the latest installed CLR prior to CLR version 4. If you specify v1.0.3705, v1.1.4322, or v2.0.50727, the version automatically rolls forward to the latest installed CLR version prior to CLR version 4 (usually v2.0.50727). If another version of the CLR is already loaded and the specified version can be loaded side-by-side in-process, the specified version is loaded; otherwise, the loaded CLR is used. This might cause a load failure.	No
`tlbid`	The identifier of the type library that contains type information about the class.	No

All attribute tags are case-sensitive. You can obtain CLSIDs, ProgIDs, threading models, and the runtime version by viewing the exported type library for the assembly with the OLE/COM ObjectViewer (Oleview.exe).

The following component manifest identifies two classes, `testClass1` and `testClass2`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
       <assemblyIdentity
                    name="myOrganization.myDivision.myManagedComp"
                    version="1.2.3.4"
                    publicKeyToken="8275b28176rcbbef"
       />
       <clrClass
                    clsid="{65722BE6-3449-4628-ABD3-74B6864F9739}"
                    progid="myManagedComp.testClass1"
                    threadingModel="Both"
                    name="myManagedComp.testClass1"
                    runtimeVersion="v1.0.3705">
       </clrClass>
       <clrClass
                    clsid="{367221D6-3559-3328-ABD3-45B6825F9732}"
                    progid="myManagedComp.testClass2"
                    threadingModel="Both"
                    name="myManagedComp.testClass2"
                    runtimeVersion="v1.0.3705">
       </clrClass>
       <file name="MyManagedComp.dll">
       </file>
</assembly>
```

5. Save and name the manifest file. The name of a component manifest is the name of the assembly library followed by the .manifest extension. For example, the myManagedComp.dll is myManagedComp.manifest.

You must embed the component manifest as a resource in the assembly.

To embed a component manifest in a managed assembly

1. Create a resource script that contains the following statement:

   ```
   RT_MANIFEST 1 myManagedComp.manifest
   ```

   In this statement, `myManagedComp.manifest` is the name of the component manifest being embedded. For this example, the script file name is `myresource.rc`.

2. Compile the script using the Microsoft Windows Resource Compiler (Rc.exe). At the command prompt, type the following command:

   ```
   rc myresource.rc
   ```

   Rc.exe produces the `myresource.res` resource file.

3. Compile the assembly's source file again and specify the resource file by using the **/win32res** option:

   ```
   /win32res:myresource.res
   ```

   Again, `myresource.res` is the name of the resource file containing embedded resource.

See also

- Registration-Free COM Interop
- Requirements for Registration-Free COM Interop
- Configuring COM Components for Registration-Free Activation
- Registration-Free Activation of .NET-Based Components: A Walkthrough