

Integrating with COM Components

The CLR provides support both for exposing C# objects as COM objects and for using COM objects from C#. Additionally, CLR components can make use of COM+ services and can be used as configured components by CLR and classic COM applications.

Binding COM and C# Objects

Interoperating between COM and C# works through either early or late binding. Early binding allows you to program with types known at compile time, while late binding forces you to program with types via dynamic discovery, using reflection on the C# side and `IDispatch` on the COM side.

When calling COM programs from C#, early binding works by providing metadata in the form of an assembly for the COM object and its interfaces. *TlbImp.exe* takes a COM type library and generates the equivalent metadata in an assembly. With the generated assembly, it's possible to instantiate and call methods on a COM object just as you would on any other C# object.

When calling C# programs from COM, early binding works via a type library. Both *TlbExp.exe* and *RegAsm.exe* allow you to generate a COM type library from your assembly. You can then use this type library with tools that support early binding via type libraries such as Visual Basic 6.

Exposing COM Objects to C#

When you instantiate a COM object, you are actually working with a proxy known as the Runtime Callable Wrapper (RCW). The RCW is responsible for managing the lifetime requirements of the COM object and translating the methods called on it into the appropriate calls on the COM object. When the garbage collector finalizes the RCW, it releases all references to the object it was holding. For situations in which you need to release the COM object without waiting for the garbage collector to finalize the RCW, you can use the static `ReleaseComObject` method of the `System.Runtime.InteropServices.Marshal` type.

The following example demonstrates how to change your MSN Instant Messenger friendly name using C# via COM Interop:

```

// RenameMe.cs - compile with:
// csc RenameMe.cs /r:Messenger.dll
// Run RenameMe.exe "new name" to change your name
// as it is displayed to other users.
// Run TlbImp.exe "C:\Program Files\Messenger\msmsgs.exe"
// to create Messenger.dll
using System;
using Messenger;
class MSNFun {
    static void Main(string[ ] args) {
        MsgrObject mo = new MsgrObject();
        IMsgrService ims = mo.Services.PrimaryService;
        ims.FriendlyName = args[0];
    }
}

```

You can also work with COM objects using the reflection API. This is more cumbersome than using *TlbImp.exe*, but is handy in cases in which it's impossible or inconvenient to run *TlbImp.exe*. To use COM through reflection, you have to get a *Type* from *Type.GetTypeFromProgID()* for each COM type you want to work with. Then, use *Activator.CreateInstance()* to create an instance of the type. To invoke methods or set or get properties, use the reflection API:

```

using System;
using System.Reflection;
public class ComReflect {
    public static void Main() {
        object obj_msword; // Microsoft Word Application
        Type wa = Type.GetTypeFromProgID("Word.Application", true);
        // Create an instance of Microsoft Word
        obj_msword = Activator.CreateInstance(wa);

        // Use the reflection API from here on in...
    }
}

```

Exposing C# Objects to COM

Just as an RCW proxy wraps a COM object when you access it from C#, code that accesses a C# object as a COM object must do so through a proxy as well. When your C# object is marshaled out to COM, the runtime creates a COM Callable Wrapper (CCW). The CCW follows the same lifetime rules as other COM objects, and as long as it is alive, a CCW maintains a traceable reference to the object it wraps. This keeps the object alive when the garbage collector is run.

The following example shows how you can export both a class and an interface from C# and control the Global Unique Identifiers (GUIDs) and Dispatch IDs (DISPIDs) assigned. After compiling *IRunInfo* and *StackSnapshot*, you can register both using *RegAsm.exe*.

```

// IRunInfo.cs
// Compile with:
// csc /t:library IRunInfo.cs
using System;
using System.Runtime.InteropServices;
[GuidAttribute("aa6b10a2-dc4f-4a24-ae5e-90362c2142c1")]
public interface IRunInfo {
    [DispId(1)]
    string GetRunInfo();
}

```

```

// StackSnapshot.cs
// compile with csc /t:library /r:IRunInfo.dll StackSnapShot.cs
using System;
using System.Runtime.InteropServices;
using System.Diagnostics;
[GuidAttribute("b72ccf55-88cc-4657-8577-72bd0ff767bc")]
public class StackSnapshot : IRunInfo {
    public StackSnapshot() {
        st = new StackTrace();
    }
    [DispId(1)]
    public string GetRunInfo() {
        return st.ToString();
    }
    private StackTrace st;
}

```

COM Mapping in C#

When you use a COM object from C#, the RCW makes a COM method look like a normal C# instance method. In COM, methods normally return an HRESULT to indicate success or failure and use an out parameter to return a value. In C#, however, methods normally return their result values and use exceptions to report errors. The RCW handles this by checking the HRESULT returned from the call to a COM method and throwing a C# exception when it finds a failure result. With a success result, the RCW returns the parameter marked as the return value in the COM method signature.

Common COM Interop Support Attributes

The FCL provides a set of attributes you can use to mark up your objects with information needed by the CLR interop services to expose managed types to the unmanaged world as COM objects. See Chapter 37, for documentation about these attributes.

COM+ Support

The CLR also includes special plumbing and interop services that allow CLR classes to be deployed as COM+ configured components. This allows both CLR clients and classic COM clients to make use of COM+ services for building scalable applications.

What Is COM+?

COM+ provides a set of services that are designed to help build scalable distributed systems, such as distributed transaction support, object pooling, Just-In-Time activation, synchronization, role-based security, loosely coupled events, and others.

These services are provided by a runtime environment called the COM+ runtime, and are based on the idea of intercepting new COM object creation and (possibly) method calls to layer in the additional services as needed.

COM classes that use COM+ services are called Configured Components because the exact set of services each COM class requires is controlled and configured using declarative attributes that are stored in a metadata repository called the COM+ Catalog.

The COM+ Catalog groups a set of configured components together into something called an Application, which also has metadata settings that control which process the COM components end up in when they are created at runtime (the options here are Library and Server, where Library components end up in the creator's process, while Server components are hosted in a separate process), the security principal the new process runs as, and other settings.

Using COM+ Services with CLR Classes

Naturally, CLR classes can also take advantage of COM+ services. Although the underlying implementation of this is currently the classic COM+ runtime, the mechanics of it are largely hidden from the .NET programmer, who may choose to work almost entirely in terms of normal CLR base classes, interfaces, and custom attributes.

The bulk of the functionality in .NET for using COM+ services is exposed via the `System.EnterpriseServices` namespace. The most important type in this namespace is the `ServicedComponent` class, which is the base class that all CLR classes must derive from if they wish to use COM+ services (i.e., if they want to be configured components).

There is a suite of custom attributes in this namespace that can control almost all of the configuration settings that would otherwise be stored in the COM+ Catalog. Examples of these attributes include both assembly-level attributes which control the settings for the entire COM+ application, the `ApplicationActivationAttributes`, which controls whether the CLR class is deployed in a COM+ Library or Server application, and component-level attributes, which declare and configure the COM+ services the CLR class wishes to be provided at runtime. Examples of component-level custom attributes include the `TransactionAttribute` (which specifies the COM+ transaction semantics for the class), the `JustInTimeActivationAttribute` (which specifies that the CLR class should have JITA semantics), the `SynchronizationAttribute` (which controls the synchronization behavior of methods), the `ObjectPoolingAttribute` (which controls whether the CLR class is pooled), and many, many others.

Although `ServicedComponent` serves as a special base class which signals the .NET Framework that a class needs COM+ services, it also provides other capabilities. In classic COM+ work, COM classes implement interfaces such as `IObjectConstruct` and `IObjectControl` to customize aspects of their behavior. When using COM+ services in .NET, your classes can override virtual methods provided by `ServicedComponent` that mirror the functionality in `IObjectConstruct` and `IObjectControl`, allowing a very natural, .NET-centric way of accomplishing the same thing.

Other important classes in the `System.EnterpriseServices` namespace include `ContextUtil` and `SecurityCallContext`. These classes provide static methods that allow a CLR-configured component to access COM+ context. This is used to control things like transaction status and to access information such as the security role a caller is in.

Lastly, let's discuss deployment. Deploying traditional COM+ applications requires one to configure the component's COM+ Catalog settings. This is typically done using either the COM+ Explorer (by hand, really only suitable for toy applications) or using custom registration code. When configuring CLR classes, there are two different approaches.

The first approach is using the *RegSvcs.exe* command-line tool. This tool performs all the relevant COM Interop and COM+ Catalog configuration, using both command-line options and the custom attributes applied to your assembly and classes to control the COM+ metadata. While this requires an extra step, arguably this approach is the most powerful and flexible, resulting in CLR-configured classes that can be used from both COM and .NET clients.

Alternatively, the .NET COM+ integration is able to automatically register classes that derive from *ServiceComponent* in the COM+ catalog when they are first instantiated. This has the advantage of not requiring any additional setup, but also has several disadvantages, most notably that the client code that indirectly causes the registration to occur needs elevated privileges, and until the class is configured, it is invisible to COM clients.

A simple C# configured class might look like this:

```
using System;
using System.EnterpriseServices;

[assembly:ApplicationName("MyCOMPlusApplication")]
[assembly:ApplicationActivation(ActivationOption.Server)]

[ObjectPooling(true), Transaction(TransactionOption.Required)]
public class MyConfiguredComponent : ServiceComponent {
    public void DoDBWork() {
        ContextUtil.SetAbort();
        // ... do database work...
        ContextUtil.SetComplete();
    }
    public override bool CanBePooled() {
        return true;
    }
}
```

System.Runtime.InteropServices

The types in this namespace work with unmanaged code using either PInvoke or COM. PInvoke (short for Platform Invoke) lets you access functions that reside in underlying operating system-specific shared libraries (on Win32, these are DLLs). COM (Component Object Model) is a Win32 legacy component architecture that is used throughout Windows and Windows applications. Many programmers experience COM through the object models of applications such as Microsoft Office, Exchange, and SQL Server. The COM support in .NET lets you access COM components as though they were native .NET classes.

We have omitted some of the more esoteric parts of this namespace, so there are some classes that aren't discussed here. For the most part, you will not need those classes unless you are developing specialized code that handles marshaling data types between managed and unmanaged code. If so, you should consult the MSDN .NET reference materials.

ArrayWithOffset

This class converts an array of value type instances to an unmanaged array. Your unmanaged code accesses this array as a pointer that initially points to the first array element. Each time the pointer increments, it points to the next element in the array. The constructor takes a mandatory **offset** argument that specifies which element should be the first element in the unmanaged array. If you want to pass the whole array, specify an offset of zero.

```
public struct ArrayWithOffset { // Public Constructors
    public ArrayWithOffset(object array, int offset); // Public Instance Methods
    public override bool Equals(object obj); // overrides ValueType
    public object GetArray();
    public override int GetHashCode(); // overrides ValueType
    public int GetOffset();
}
```

Hierarchy

System.Object > System.ValueType > ArrayWithOffset

AssemblyRegistrationFlags

This enumeration specifies the flags you can use with `IRegistrationServices.RegisterAssembly()`.

```
public enum AssemblyRegistrationFlags {
    None = 0x00000000,
    SetCodeBase = 0x00000001
}
```

Hierarchy

System.Object > System.ValueType > System.Enum(System.IComparable, System.IFormattable, System.IConvertible) > AssemblyRegistrationFlags

Passed To

`IRegistrationServices.RegisterAssembly()`,
`RegistrationServices.RegisterAssembly()`

BestFitMappingAttribute

This class determines whether Unicode characters are converted to ANSI characters by selecting the closest matching character.

```
public sealed class BestFitMappingAttribute : Attribute { // Public Constructors
    public BestFitMappingAttribute(bool BestFitMapping); // Public Instance Fields
    public bool ThrowOnUnmappableChar; // Public Instance Properties
    public bool BestFitMapping { get; }
}
```

Hierarchy

System.Object > System.Attribute > BestFitMappingAttribute

Valid On

Assembly, Class, Struct, Interface

CallingConvention

This enumeration specifies the calling convention to use when you invoke a function. `DllImportAttribute` uses this in its `CallingConvention` parameter.

`Cdecl` specifies the standard calling convention used by C++ and C programs. This is required for functions that take a variable number of arguments, such as `printf()`. `FastCall` attempts to put function arguments into registers. `StdCall` is the convention used for calling Win32 API functions. `ThisCall` is the calling convention used by C++ member functions taking fixed arguments. Use the `Winapi` calling convention for function calls that use `PASCAL` or `__far` `__pascal`.

```
public enum CallingConvention {
    Winapi = 1,
    Cdecl = 2,
    StdCall = 3,
    ThisCall = 4,
    FastCall = 5
}
```

Hierarchy

System.Object > System.ValueType > System.Enum(System.IComparable, System.IFormattable, System.IConvertible) > CallingConvention

Passed To

```
System.Reflection.Emit.ILGenerator.EmitCalli(),
System.Reflection.Emit.ModuleBuilder.DefinePInvokeMethod(),
System.Reflection.Emit.SignatureHelper.GetMethodSigHelper(),
System.Reflection.Emit.TypeBuilder.DefinePInvokeMethod()
```

CharSet

This enumeration specifies the character set that is used for marshaled strings. It is used by `DllImportAttribute` and `StructLayoutAttribute`.

`Ansi` marshals strings using one-byte ANSI characters, while `Unicode` uses two bytes to represent a single Unicode character. The `Auto` value is used only for `PInvoke` and specifies that `PInvoke` should decide how to marshal the strings based on your operating system (Unicode for Windows NT/2000/XP and ANSI for Windows 9x/Me).

```
public enum CharSet {
    None = 1,
    Ansi = 2,
    Unicode = 3,
    Auto = 4
}
```

Hierarchy

System.Object > System.ValueType > System.Enum(System.IComparable, System.IFormattable, System.IConvertible) > CharSet

Passed To

```
System.Reflection.Emit.ModuleBuilder.DefinePInvokeMethod(),
System.Reflection.Emit.TypeBuilder.DefinePInvokeMethod()
```

ClassInterfaceAttribute

This attribute specifies the interface that should be exposed to COM when you generate a type library. See `ClassInterfaceType` for the possible arguments to this attribute.

```
public sealed class ClassInterfaceAttribute : Attribute { // Public Constructors
    public ClassInterfaceAttribute(ClassInterfaceType classInterfaceType);
    public ClassInterfaceAttribute(short classInterfaceType); // Public Instance Properties
    public ClassInterfaceType Value{get; }
}
```

Hierarchy

System.Object > System.Attribute > ClassInterfaceAttribute

Valid On

Assembly, Class

ClassInterfaceType

This enumeration contains values to use as arguments for **ClassInterfaceAttribute**. **AutoDispatch** specifies that a dispatch-only interface should be generated. **AutoDual** specifies that a dual interface should be generated, and **None** specifies that no interface should be generated.

```
public enum ClassInterfaceType {
    None = 0,
    AutoDispatch = 1,
    AutoDual = 2
}
```

Hierarchy

System.Object > System.ValueType > System.Enum(System.IComparable, System.IFormattable, System.IConvertible) > ClassInterfaceType

Returned By

ClassInterfaceAttribute.Value

Passed To

ClassInterfaceAttribute.ClassInterfaceAttribute()

CoClassAttribute

This attribute describes the class ID of a coclass that was imported from a type library.

```
public sealed class CoClassAttribute : Attribute { // Public Constructors
    public CoClassAttribute(Type coClass); // Public Instance Properties
    public Type CoClass{get; }
}
```

Hierarchy

System.Object > System.Attribute > CoClassAttribute

Valid On

Interface

ComAliasNameAttribute

This attribute is automatically added when COM type libraries are imported into the .NET runtime. COM uses alias names for various data types (such as `typedef [public] int SHOE_SIZE`). When you import a COM object that uses such an alias, .NET automatically decorates each parameter, property, field, and return value with this attribute. If you need to know the name of the COM alias, use the `System.Reflection` API to see if this custom attribute has been attached to the parameter, property, field, or return value you are interested in.

Since .NET automatically converts COM aliases to the underlying .NET types when it imports a type library, you do not need to use this attribute for typical applications (tool developers will find this attribute useful, though).

```
public sealed class ComAliasNameAttribute : Attribute { // Public Constructors
    public ComAliasNameAttribute(string alias); // Public Instance Properties
    public string Value { get; }
}
```

Hierarchy

`System.Object` > `System.Attribute` > `ComAliasNameAttribute`

Valid On

Property, Field, Parameter, ReturnValue

ComCompatibleVersionAttribute

This attribute tells COM clients that the types in the assembly are backward-compatible with those from an earlier version.

```
public sealed class ComCompatibleVersionAttribute : Attribute { // Public Constructors
    public ComCompatibleVersionAttribute(int major, int minor, int build, int revision); // Public Instance Properties
    public int BuildNumber { get; }
    public int MajorVersion { get; }
    public int MinorVersion { get; }
    public int RevisionNumber { get; }
}
```

Hierarchy

`System.Object` > `System.Attribute` > `ComCompatibleVersionAttribute`

Valid On

Assembly

ComConversionLossAttribute

The presence of this attribute indicates that information about a type was lost as it was imported from a type library.

```
public sealed class ComConversionLossAttribute : Attribute { // Public Constructors
    public ComConversionLossAttribute();
}
```

Hierarchy

System.Object > System.Attribute > ComConversionLossAttribute

Valid On

All

COMException

When a COM error occurs, .NET tries to map it to an exception in the .NET Framework and throws that exception. If the COM error does not map to any exception in the .NET Framework, this exception is thrown instead. It's the “couldn't find an exception” exception.

```
public class COMException : ExternalException { // Public Constructors
    public COMException();
    public COMException(string message);
    public COMException(string message, Exception inner);
    public COMException(string message, int errorCode); // Protected Constructors
    protected COMException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context); // Public Instance Methods
    public override string ToString(); // overrides Exception
}
```

Hierarchy

System.Object > System.Exception(System.Runtime.Serialization.ISerializable) > System.SystemException > ExternalException > COMException

ComImportAttribute

This attribute indicates that the type decorated by this attribute is in fact an unmanaged type defined in a previously published type library and should be treated differently internally to support that.

This attribute is necessary only if the .NET type definition—the class definition in C#—This interface indicates a type whose members can be removed or added. The members are represented as `System.Reflection.MemberInfo` objects.

is merely a “shim” for interacting with the unmanaged version. In most cases, .NET programmers only use this type when interacting with existing COM APIs, such as when building Explorer Shell Extensions.

```
public sealed class ComImportAttribute : Attribute { // Public Constructors
    public ComImportAttribute();
}
```

Hierarchy

System.Object > System.Attribute > ComImportAttribute

Valid On

Class, Interface

ComInterfaceType

This enumeration specifies the COM interface type. Use this attribute with `InterfaceTypeAttribute` to specify how your .NET interfaces are exposed to COM.

```
public enum ComInterfaceType {  
    InterfaceIsDual = 0,  
    InterfaceIsIUnknown = 1,  
    InterfaceIsIDispatch = 2  
}
```

Hierarchy

`System.Object` | `System.ValueType` > `System.Enum`(`System.IComparable`, `System.IFormattable`, `System.IConvertible`) > `ComInterfaceType`

Returned By

`InterfaceTypeAttribute.Value`

Passed To

`InterfaceTypeAttribute.InterfaceTypeAttribute()`

ComMemberType

This enumeration describes a COM member. `Method` indicates that the member is an ordinary method. `PropGet` and `PropSet` identify methods that get and set the values of properties (getters and setters).

```
public enum ComMemberType {  
    Method = 0,  
    PropGet = 1,  
    PropSet = 2  
}
```

Hierarchy

`System.Object` > `System.ValueType` > `System.Enum`(`System.IComparable`, `System.IFormattable`, `System.IConvertible`) > `ComMemberType`

Passed To

`Marshal.GetMethodInfoForComSlot()`

ComRegisterFunctionAttribute

This attribute is attached to a static method to indicate that it should be invoked when the enclosing assembly is registered with COM. The method should take two string arguments. The first is the name of the registry key being updated, and the second is the namespace-qualified name of the type being registered (such as `System.String`). There can only be one registration function in each assembly.

Microsoft suggests that you do not use this feature and includes it only for backward compatibility. If you use this feature to specify a registration method, you must also specify an

unregistration method (see `ComUnregisterFunctionAttribute`) that reverses all changes you made in the registration function.

```
public sealed class ComRegisterFunctionAttribute : Attribute {  
    // Public Constructors  
    public ComRegisterFunctionAttribute();  
}
```

Hierarchy

System.Object > System.Attribute > ComRegisterFunctionAttribute

Valid On

Method

ComSourceInterfacesAttribute

This attribute indicates the unmanaged event (using the COM `IConnectionPoint` architecture) interfaces that are available on the decorated type. For each method defined in the COM interface, the type must provide a corresponding “event” instance that the COM architecture will plug into.

This attribute is only necessary when building .NET objects for plugging into COM event-aware systems, such as ActiveX control containers.

```
public sealed class ComSourceInterfacesAttribute : Attribute {  
    // Public Constructors  
    public ComSourceInterfacesAttribute(string sourceInterfaces);  
    public ComSourceInterfacesAttribute(Type sourceInterface);  
    public ComSourceInterfacesAttribute(Type sourceInterface1, Type sourceInterface2);  
    public ComSourceInterfacesAttribute(Type sourceInterface1, Type sourceInterface2,  
        Type sourceInterface3);  
    public ComSourceInterfacesAttribute(Type sourceInterface1, Type sourceInterface2,  
        Type sourceInterface3, Type sourceInterface4);  
    // Public Instance Properties  
    public string Value { get; }  
}
```

Hierarchy

System.Object > System.Attribute > ComSourceInterfacesAttribute

Valid On

Class

ComUnregisterFunctionAttribute

This attribute is attached to a static method to indicate that it should be invoked when the enclosing assembly is unregistered from COM. There can only be one unregistration function in each assembly.

For more details, see `ComRegisterFunctionAttribute`.

```
public sealed class ComUnregisterFunctionAttribute : Attribute {  
    // Public Constructors  
    public ComUnregisterFunctionAttribute();  
}
```

Hierarchy

System.Object > System.Attribute > ComUnregisterFunctionAttribute

Valid On

Method

ComVisibleAttribute

By default, all public assemblies, types, and members that are registered with COM are visible to COM. This attribute is used with a false argument to hide an assembly, type, or member from COM. This attribute has a cascading effect: if you hide an assembly, all the public types in that assembly are hidden as well.

You can override this attribute on individual types. If, for example, you want to make only one public type visible from an assembly, add the attribute [`ComVisible(false)`] to the assembly, but also add [`ComVisible(true)`] to the one type that you want to expose.

```
public sealed class ComVisibleAttribute : Attribute { // Public Constructors
    public ComVisibleAttribute(bool visibility); // Public Instance Properties
    public bool Value { get; }
}
```

Hierarchy

System.Object > System.Attribute > ComVisibleAttribute

Valid On

Assembly, Class, Struct, Enum, Method, Property, Field, Interface, Delegate

CurrencyWrapper

This class is used to create a wrapper around a decimal value. Then, when you pass the newly created `CurrencyWrapper` to an unmanaged method, the object is marshaled as the `VT_CURRENCY` type.

```
public sealed class CurrencyWrapper { // Public Constructors
    public CurrencyWrapper(decimal obj);
    public CurrencyWrapper(object obj); // Public Instance Properties
    public decimal WrappedObject { get; }
}
```

DispatchWrapper

By default, objects are passed to unmanaged methods as the `VT_UNKNOWN` type. This wrapper is used to send an object as type `VT_DISPATCH`.

```
public sealed class DispatchWrapper { // Public Constructors
    public DispatchWrapper(object obj); // Public Instance Properties
    public object WrappedObject { get; }
}
```

DispIdAttribute

Specifies a member's `DispId` when it is exposed to COM.

```

public sealed class DispIdAttribute : Attribute { // Public Constructors
    public DispIdAttribute(int dispId); // Public Instance Properties
    public int Value{get; }
}

```

Hierarchy

System.Object > System.Attribute > DispIdAttribute

Valid On

Method, Property, Field, Event

DllImportAttribute

This attribute (and, in C#, the keyword “extern”) specifies that a method definition is implemented externally (usually in a DLL). Apply this attribute to a method that has been declared (but not defined) to specify the DLL name and entry point in which the method can be found.

The attribute can be customized in a number of different ways to help control the binding against the external method. The **CallingConvention** value dictates how the parameters to the call (and return value coming back) should be sent to the function. **CallingConvention.StdCall** (used for calling into `_stdcall`-declared functions, which is most of the Win32 API set) and **CallingConvention.Cdecl** (used for calling functions declared directly from C or C++) are the two most common values. The **CharSet** value indicates which character set parameters to the call are expected to be, either two-byte Unicode or one-byte ANSI. **EntryPoint** indicates the name of the exported function from the DLL to bind to (normally this is guessed from the name of the .NET-declared method), and **ExactSpelling** indicates whether the .NET compiler should attempt to “best match” a declared **DllImport** method against a possible set of exported functions. The **PreserveSig** value indicates how .NET should treat `[out]`-declared and `[retval]`-declared parameters. By default, the .NET compilers ignore the HRESULT return value on IDL-declared methods and use the `[retval]`-declared parameter as the return value; setting **PreserveSig** to `true` turns this off. **BestFitMapping** is used to define whether the CLR should try to “best fit” Unicode characters to ANSI equivalents, and is turned on by default. **ThrowOnUnmappableChar** indicates whether an exception should be thrown when the interop marshaler attempts to convert an unmappable character, and is turned off by default. Finally, because many Win32 APIs use the **GetLastError** API call to note the exact reason a call fails, the **SetLastError** value indicates whether the caller should use that API to discover the reason for failures.

```

public sealed class DllImportAttribute : Attribute { // Public Constructors
    public DllImportAttribute(string dllName); // Public Instance Fields
    public bool BestFitMapping;
    public CallingConvention CallingConvention;
    public CharSet CharSet;
    public string EntryPoint;
    public bool ExactSpelling;
    public bool PreserveSig;
    public bool SetLastError;
    public bool ThrowOnUnmappableChar; // Public Instance Properties
    public string Value{get; }
}

```

Hierarchy

System.Object > System.Attribute > DllImportAttribute

Valid On

Method

ErrorWrapper

This wrapper is used to force an integer, Exception, or other object to be marshaled as type VT_ERROR.

```
public sealed class ErrorWrapper { // Public Constructors
    public ErrorWrapper(Exception e);
    public ErrorWrapper(int errorCode);
    public ErrorWrapper(object errorCode); // Public Instance Properties
    public int ErrorCode { get; }
}
```

ExtensibleClassFactory

This class exposes the method RegisterObjectCreationCallback(), which specifies a delegate that manufactures instances of a managed type. Use this to build managed types that extend unmanaged types. Since a managed type cannot directly inherit from an unmanaged type, the managed type needs to aggregate an instance of the unmanaged type. The delegate that you register with RegisterObjectCreationCallback() takes care of creating the unmanaged type.

```
public sealed class ExtensibleClassFactory { // Public Static Methods
    public static void RegisterObjectCreationCallback(ObjectCreationDelegate callback);
}
```

ExternalException

This is the base class for COM interop and SEH (Structured Exception Handler) exceptions.

```
public class ExternalException : SystemException { // Public Constructors
    public ExternalException();
    public ExternalException(string message);
    public ExternalException(string message, Exception inner);
    public ExternalException(string message, int errorCode); // Protected Constructors
    protected ExternalException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context); // Public Instance Properties
    public virtual int ErrorCode { get; }
}
```

Hierarchy

System.Object > System.Exception(System.Runtime.Serialization.ISerializable) > System.SystemException > ExternalException

Subclasses

COMException, SEHException

FieldOffsetAttribute

This attribute controls the offset, in bytes, of a field. Use it to match your .NET types to the layout of C and C++ structures exactly. This attribute can be used only within classes that have the `StructLayoutAttribute` attribute where `LayoutKind.Explicit` was used.

```
public sealed class FieldOffsetAttribute : Attribute { // Public Constructors
    public FieldOffsetAttribute(int offset); // Public Instance Properties
    public int Value { get; }
}
```

Hierarchy

System.Object > System.Attribute > FieldOffsetAttribute

Valid On

Field

GCHandle

This class is used when you need to pass a managed object to unmanaged code. To use this class, pass an instance of a .NET-managed type to the `Alloc()` method. The single-argument form of `Alloc()` creates the `GCHandle` with `GCHandleType.Normal`, which ensures that the object will not be freed by the garbage collector. (This means that some kind of user code must also call the `Free()` method in order to release the object.) Managed code can use the `Target` property to access the underlying object.

```
public struct GCHandle { // Public Instance Properties
    public bool IsAllocated { get; }
    public object Target { set; get; } // Public Static Methods
    public static GCHandle Alloc(object value);
    public static GCHandle Alloc(object value, GCHandleType type);
    public static explicit operator GCHandle(IntPtr value);
    public static explicit operator IntPtr(GCHandle value); // Public Instance Methods
    public IntPtr AddrOfPinnedObject();
    public void Free();
}
```

Hierarchy

System.Object > System.ValueType > GCHandle

GCHandleType

This enumeration contains values for the two-argument form of `GCHandle.Alloc()`. `Normal` protects the object from being garbage collected, and `Pinned` does the same (but it also enables the `GCHandle.AddrOfPinnedObject()` method). `Weak` and `WeakTrackResurrection` both allow the object to be garbage-collected. However, `Weak` causes the object to be zeroed out before the finalizer runs, but `WeakTrackResurrection` does not zero the object, so the object's finalizer can safely resurrect it.

```
public enum GCHandleType {
    Weak = 0,
    WeakTrackResurrection = 1,
    Normal = 2,
    Pinned = 3
}
```

Hierarchy

System.Object > System.ValueType > System.Enum(System.IComparable, System.IFormattable, System.IConvertible) > GCHandleType

Passed To

GCHandle.Alloc()

GuidAttribute

This attribute is used to specify the GUID of assemblies, modules, or types you expose to COM. If you don't use this attribute to specify a GUID, one is automatically generated. When you apply this attribute, use its full name (`[GuidAttribute()]`) rather than `[Guid()]` to avoid clashes with the `System.Guid` type.

```
public sealed class GuidAttribute : Attribute { // Public Constructors
    public GuidAttribute(string guid); // Public Instance Properties
    public string Value { get; }
}
```

Hierarchy

System.Object > System.Attribute > GuidAttribute

Valid On

Assembly, Class, Struct, Enum, Interface, Delegate

HandleRef

When you pass a managed object into unmanaged code using `PInvoke`, there is a chance that the garbage collector will finalize the object before the unmanaged code is finished with it. This can only happen when your managed code does not reference the object after the `PInvoke` call. Because the garbage collector's reach does not extend into unmanaged code, this fools the garbage collector into thinking that you are finished with it.

This class is used to wrap your managed object before passing it into unmanaged code, and you are guaranteed that the garbage collector will not touch it until the `PInvoke` call returns.

```
public struct HandleRef { // Public Constructors
    public HandleRef(object wrapper, IntPtr handle); // Public Instance Properties
    public IntPtr Handle { get; }
    public object Wrapper { get; } // Public Static Methods
    public static explicit operator IntPtr(HandleRef value);
}
```

Hierarchy

System.Object > System.ValueType > HandleRef

IDispatchImplAttribute

There are multiple implementations of `IDispatch` available for you to expose dual interfaces and dispinterfaces to COM. Attach this attribute to a class or an assembly to specify which `IDispatch` implementation to use. If you apply this attribute to an assembly, it applies to all classes within that assembly. For a list of available `IDispatch` implementations, see `IDispatchImplType`.

```
public sealed class IDispatchImplAttribute : Attribute { // Public Constructors
    public IDispatchImplAttribute(IDispatchImplType implType);
    public IDispatchImplAttribute(short implType); // Public Instance Properties
    public IDispatchImplType Value { get; }
}
```

Hierarchy

System.Object > System.Attribute > IDispatchImplAttribute

Valid On

Assembly, Class

IDispatchImplType

This enumeration contains the values used by `IDispatchImplAttribute`. `SystemDefinedImpl` tells the runtime to decide which `IDispatch` implementation to use. `InternalImpl` tells .NET to use its own `IDispatch` implementation, and `CompatibleImpl` uses an `IDispatch` implementation that is compatible with OLE automation. If you use this last implementation, it requires static type information. Because this information is automatically generated at runtime, `CompatibleImpl` may have an adverse impact on performance.

```
public enum IDispatchImplType {
    SystemDefinedImpl = 0,
    InternalImpl = 1,
    CompatibleImpl = 2
}
```

Hierarchy

System.Object > System.ValueType > System.Enum(System.IComparable, System.IFormattable, System.IConvertible) > IDispatchImplType

Returned By

`IDispatchImplAttribute.Value`

Passed To

`IDispatchImplAttribute.IDispatchImplAttribute()`

InAttribute

This attribute is attached to a parameter to marshal it as an `in` parameter. By default, parameters are marshaled based on their modifiers, so this attribute is only necessary if you want to override the defaults. Parameters with no modifiers are marshaled as `[In]`. Parameters with the `ref` modifier are marshaled as `[In, Out]`. Parameters with the `out` modifier are marshaled as `[Out]`.

```
public sealed class InAttribute : Attribute { // Public Constructors
    public InAttribute();
}
```

Hierarchy

System.Object > System.Attribute > InAttribute

Valid On

Parameter

InterfaceTypeAttribute

This attribute is used to create a .NET interface that maps a COM interface into your managed application. See [ComInterfaceType](#) for the available values.

```
public sealed class InterfaceTypeAttribute : Attribute { // Public Constructors
    public InterfaceTypeAttribute(ComInterfaceType interfaceType);
    public InterfaceTypeAttribute(short interfaceType); // Public Instance Properties
    public ComInterfaceType Value { get; }
}
```

Hierarchy

System.Object > System.Attribute > InterfaceTypeAttribute

Valid On

Interface

InvalidComObjectException

This exception signals that an invalid COM object has been used.

```
public class InvalidComObjectException : SystemException { // Public Constructors
    public InvalidComObjectException();
    public InvalidComObjectException(string message);
    public InvalidComObjectException(string message, Exception inner); // Protected Constructors
    protected InvalidComObjectException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context);
}
```

Hierarchy

System.Object > System.Exception(System.Runtime.Serialization.ISerializable) > System.SystemException > InvalidComObjectException

InvalidOleVariantTypeException

This exception signals that the marshaler failed in an attempt to marshal a variant to managed code.

```

public class InvalidOleVariantTypeException : SystemException { // Public Constructors
    public InvalidOleVariantTypeException();
    public InvalidOleVariantTypeException(string message);
    public InvalidOleVariantTypeException(string message, Exception inner); // Protected Constructors
    protected InvalidOleVariantTypeException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context);
}

```

Hierarchy

System.Object > System.Exception(System.Runtime.Serialization.ISerializable) > System.SystemException > InvalidOleVariantTypeException

IRegistrationServices

This interface defines the interface used by classes that register and unregister assemblies with COM.

```

public interface IRegistrationServices { // Public Instance Methods
    public Guid GetManagedCategoryGuid();
    public string GetProgIdForType(Type type);
    public Type[] GetRegistrableTypesInAssembly(System.Reflection.Assembly assembly);
    public bool RegisterAssembly(System.Reflection.Assembly assembly,
        AssemblyRegistrationFlags flags);
    public void RegisterTypeForComClients(Type type, ref Guid g);
    public bool TypeRepresentsComType(Type type);
    public bool TypeRequiresRegistration(Type type);
    public bool UnregisterAssembly(System.Reflection.Assembly assembly);
}

```

Implemented By

RegistrationServices

LayoutKind

This enumeration is used to specify how objects are laid out when they are passed to unmanaged code. **Auto** specifies that .NET should choose the best method to lay out the objects. **Explicit** gives you complete control over how the object's data members are laid out. You must use **FieldOffsetAttribute** with each member if you specify **Explicit**. **Sequential** lays out the object's members one after the other, in the same order that they are defined in the class definition.

```

public enum LayoutKind {
    Sequential = 0,
    Explicit = 2,
    Auto = 3
}

```

Hierarchy

System.Object > System.ValueType > System.Enum(System.IComparable, System.IFormattable, System.IConvertible) > LayoutKind

Returned By

StructLayoutAttribute.Value

Passed To

StructLayoutAttribute.StructLayoutAttribute()

LCIDConversionAttribute

This attribute indicates that a parameter within the method's unmanaged signature expects an [Lcid] argument. Pass an integer value to the constructor to specify which parameter, starting with 0 for the first parameter.

```
public sealed class LCIDConversionAttribute : Attribute { // Public Constructors
    public LCIDConversionAttribute(int Lcid); // Public Instance Properties
    public int Value { get; }
}
```

Hierarchy

System.Object > System.Attribute > LCIDConversionAttribute

Valid On

Method

Marshal

This class offers a collection of static methods for working with unmanaged memory and converting managed types to unmanaged types. Unless you are developing specialized code for marshaling types between managed and unmanaged code, you probably do not need to use any of these methods.

GetHRForException() converts a .NET exception to a COM HRESULT. If you are curious about the platform you are running on, you can find out the size of a character with the SystemDefaultCharSize field, which is 1 on an ANSI platform (Windows 9x/Me) and 2 on a Unicode platform (Windows NT, 2000, and XP).

Use the IsComObject() method to determine whether an object is actually an unmanaged COM object. The AddRef() method increments a COM object's reference count.

```
public sealed class Marshal { // Public Static Fields
    public static readonly int SystemDefaultCharSize; // =2
    public static readonly int SystemMaxDBCSCharSize; // =1 // Public Static Methods
    public static int AddRef(IntPtr pUnk);
    public static IntPtr AllocCoTaskMem(int cb);
    public static IntPtr AllocHGlobal(int cb);
    public static IntPtr AllocHGlobal(IntPtr cb);
    public static object BindToMoniker(string monikerName);
    public static void ChangeWrapperHandleStrength(object otp, bool fIsWeak);
    public static void Copy(byte[] source, int startIndex, IntPtr destination, int length);
    public static void Copy(char[] source, int startIndex, IntPtr destination, int length);
    public static void Copy(double[] source, int startIndex, IntPtr destination, int length);
    public static void Copy(short[] source, int startIndex, IntPtr destination, int length);
    public static void Copy(int[] source, int startIndex, IntPtr destination, int length);
    public static void Copy(long[] source, int startIndex, IntPtr destination, int length);
    public static void Copy(IntPtr source, byte[] destination, int startIndex, int length);
    public static void Copy(IntPtr source, char[] destination, int startIndex, int length);
    public static void Copy(IntPtr source, double[] destination, int startIndex, int length);
    public static void Copy(IntPtr source, short[] destination, int startIndex, int length);
}
```

```

public static void Copy(IntPtr source, int[ ] destination, int startIndex, int length);
public static void Copy(IntPtr source, long[ ] destination, int startIndex, int length);
public static void Copy(IntPtr source, float[ ] destination, int startIndex, int length);
public static void Copy(float[ ] source, int startIndex, IntPtr destination, int length);
public static object CreateWrapperOfType(object o, Type t);
public static void DestroyStructure(IntPtr ptr, Type structureType);
public static void FreeBSTR(IntPtr ptr);
public static void FreeCoTaskMem(IntPtr ptr);
public static void FreeHGlobal(IntPtr hGlobal);
public static Guid GenerateGuidForType(Type type);
public static string GenerateProgIdForType(Type type);
public static object GetActiveObject(string progID);
public static IntPtr GetComInterfaceForObject(object o, Type T);
public static object GetComObjectData(object obj, object key);
public static int GetComSlotForMethodInfo(System.Reflection.MemberInfo m);
public static int GetEndComSlot(Type t);
public static int GetExceptionCode();
public static IntPtr GetExceptionPointers();
public static IntPtr GetHINSTANCE(System.Reflection.Module m);
public static int GetHRForException(Exception e);
public static int GetHRForLastWin32Error();
public static IntPtr GetIDispatchForObject(object o);
public static IntPtr GetITypeInfoForType(Type t);
public static IntPtr GetIUnknownForObject(object o);
public static int GetLastWin32Error();
public static IntPtr GetManagedThunkForUnmanagedMethodPtr(IntPtr pfnMethodToWrap,
    IntPtr pbSignature, int cbSignature);
public static MemberInfo GetMethodInfoForComSlot(Type t, int slot,
    ref ComMemberType memberType);
public static void GetNativeVariantForObject(object obj, IntPtr pDstNativeVariant);
public static object GetObjectForIUnknown(IntPtr pUnk);
public static object GetObjectForNativeVariant(IntPtr pSrcNativeVariant);
public static object[ ] GetObjectsForNativeVariants(IntPtr aSrcNativeVariant, int cVars);
public static int GetStartComSlot(Type t);
public static Thread GetThreadFromFiberCookie(int cookie);
public static object GetTypedObjectForIUnknown(IntPtr pUnk, Type t);
public static Type GetTypeForITypeInfo(IntPtr piTypeInfo);
public static string GetTypeInfoName(UCOMITypeInfo pTI);
public static Guid GetTypeLibGuid(UCOMITypeLib pTLB);
public static Guid GetTypeLibGuidForAssembly(System.Reflection.Assembly asm);
public static int GetTypeLibLcid(UCOMITypeLib pTLB);
public static string GetTypeLibName(UCOMITypeLib pTLB);
public static IntPtr GetUnmanagedThunkForManagedMethodPtr(IntPtr pfnMethodToWrap,
    IntPtr pbSignature, int cbSignature);
public static bool IsComObject(object o);
public static bool IsTypeVisibleFromCom(Type t);
public static int NumParamBytes(System.Reflection.MethodInfo m);
public static IntPtr OffsetOf(Type t, string fieldName);
public static void Prelink(System.Reflection.MethodInfo m);
public static void PrelinkAll(Type c);
public static string PtrToStringAnsi(IntPtr ptr);
public static string PtrToStringAnsi(IntPtr ptr, int len);
public static string PtrToStringAuto(IntPtr ptr);
public static string PtrToStringAuto(IntPtr ptr, int len);
public static string PtrToStringBSTR(IntPtr ptr);
public static string PtrToStringUni(IntPtr ptr);
public static string PtrToStringUni(IntPtr ptr, int len);
public static object PtrToStructure(IntPtr ptr, Type structureType);
public static void PtrToStructure(IntPtr ptr, object structure);
public static int QueryInterface(IntPtr pUnk, ref Guid iid, out IntPtr ppv);
public static byte ReadByte(IntPtr ptr);

```

```

public static byte ReadByte(IntPtr ptr, int ofs);
public static byte ReadByte(in object ptr, int ofs);
public static short ReadInt16(IntPtr ptr);
public static short ReadInt16(IntPtr ptr, int ofs);
public static short ReadInt16(in object ptr, int ofs);
public static int ReadInt32(IntPtr ptr);
public static int ReadInt32(IntPtr ptr, int ofs);
public static int ReadInt32(in object ptr, int ofs);
public static long ReadInt64(IntPtr ptr);
public static long ReadInt64(IntPtr ptr, int ofs);
public static long ReadInt64(in object ptr, int ofs);
public static IntPtr ReadIntPtr(IntPtr ptr);
public static IntPtr ReadIntPtr(IntPtr ptr, int ofs);
public static IntPtr ReadIntPtr(in object ptr, int ofs);
public static IntPtr ReAllocCoTaskMem(IntPtr pv, int cb);
public static IntPtr ReAllocHGlobal(IntPtr pv, IntPtr cb);
public static int Release(IntPtr pUnk);
public static int ReleaseComObject(object o);
public static void ReleaseThreadCache();
public static bool SetComObjectData(object obj, object key, object data);
public static int SizeOf(object structure);
public static int SizeOf(Type t);
public static IntPtr StringToBSTR(string s);
public static IntPtr StringToCoTaskMemAnsi(string s);
public static IntPtr StringToCoTaskMemAuto(string s);
public static IntPtr StringToCoTaskMemUni(string s);
public static IntPtr StringToHGlobalAnsi(string s);
public static IntPtr StringToHGlobalAuto(string s);
public static IntPtr StringToHGlobalUni(string s);
public static void StructureToPtr(object structure, IntPtr ptr, bool fDeleteOld);
public static void ThrowExceptionForHR(int errorCode);
public static void ThrowExceptionForHR(int errorCode, IntPtr errorInfo);
public static IntPtr UnsafeAddrOfPinnedArrayElement(Array arr, int index);
public static void WriteByte(IntPtr ptr, byte val);
public static void WriteByte(IntPtr ptr, int ofs, byte val);
public static void WriteByte(in object ptr, int ofs, byte val);
public static void WriteInt16(IntPtr ptr, char val);
public static void WriteInt16(IntPtr ptr, short val);
public static void WriteInt16(IntPtr ptr, int ofs, char val);
public static void WriteInt16(IntPtr ptr, int ofs, short val);
public static void WriteInt16(in object ptr, int ofs, char val);
public static void WriteInt16(in object ptr, int ofs, short val);
public static void WriteInt32(IntPtr ptr, int val);
public static void WriteInt32(IntPtr ptr, int ofs, int val);
public static void WriteInt32(in object ptr, int ofs, int val);
public static void WriteInt64(IntPtr ptr, int ofs, long val);
public static void WriteInt64(IntPtr ptr, long val);
public static void WriteInt64(in object ptr, int ofs, long val);
public static void WriteIntPtr(IntPtr ptr, int ofs, IntPtr val);
public static void WriteIntPtr(IntPtr ptr, IntPtr val);
public static void WriteIntPtr(in object ptr, int ofs, IntPtr val);
}

```

MarshalAsAttribute

This optional attribute is used to explicitly specify the unmanaged type a parameter, field, or return value should be marshaled to. If you do not specify this attribute, .NET uses the type's default marshaler. The `UnmanagedType` enumeration contains the unmanaged types you can marshal to with this attribute.

```

public sealed class MarshalAsAttribute : Attribute { // Public Constructors
    public MarshalAsAttribute(short unmanagedType);
    public MarshalAsAttribute(UnmanagedType unmanagedType); // Public Instance Fields
    public UnmanagedType ArraySubType;
    public string MarshalCookie;
    public string MarshalType;
    public Type MarshalTypeRef;
    public VarEnum SafeArraySubType;
    public Type SafeArrayUserDefinedSubType;
    public int SizeConst;
    public short SizeParamIndex; // Public Instance Properties
    public UnmanagedType Value { get; }
}

```

Hierarchy

System.Object > System.Attribute > MarshalAsAttribute

Valid On

Field, Parameter, ReturnValue

MarshalDirectiveException

This exception is thrown when the marshaler encounters an unsupported MarshalAsAttribute.

```

public class MarshalDirectiveException : SystemException { // Public Constructors
    public MarshalDirectiveException();
    public MarshalDirectiveException(string message);
    public MarshalDirectiveException(string message, Exception inner); // Protected Constructors
    protected MarshalDirectiveException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context);
}

```

Hierarchy

System.Object > System.Exception(System.Runtime.Serialization.ISerializable) > System.SystemException > MarshalDirectiveException

ObjectCreationDelegate

Use this delegate with the ExtensibleClassFactory.RegisterObjectCreationCallback() method to create a COM object.

```

public delegate IntPtr ObjectCreationDelegate(IntPtr aggregator);

```

Passed To

ExtensibleClassFactory.RegisterObjectCreationCallback()

OptionalAttribute

This attribute is attached to a parameter to indicate that it is optional.

```

public sealed class OptionalAttribute : Attribute { // Public Constructors
    public OptionalAttribute();
}

```

Hierarchy

System.Object > System.Attribute > OptionalAttribute

Valid On

Parameter

OutAttribute

This attribute is attached to a parameter to cause it to be marshaled as an `out` parameter. See `InAttribute` for more details, including information on the default behavior.

```
public sealed class OutAttribute : Attribute { // Public Constructors
    public OutAttribute();
}
```

Hierarchy

System.Object > System.Attribute > OutAttribute

Valid On

Parameter

PreserveSigAttribute

When .NET converts a managed method signature to an unmanaged signature, it changes the return value to a parameter that has the `out` and `retval` COM attributes. Instead of the original return value, the unmanaged method returns a COM HRESULT. If you want to override this behavior, attach the `PreserveSigAttribute` to the method.

Something similar happens when you call unmanaged methods from managed code. In that case, the `[out, retval]` parameter on the COM side becomes the return value, and an HRESULT that indicates an error condition is translated into a .NET exception. If you want to be able to access the HRESULT as a `long` return value, use the `PreserveSigAttribute` on the methods in your COM interface declaration (see `InterfaceTypeAttribute`).

```
public sealed class PreserveSigAttribute : Attribute { // Public Constructors
    public PreserveSigAttribute();
}
```

Hierarchy

System.Object > System.Attribute > PreserveSigAttribute

Valid On

Method

ProgIdAttribute

This attribute is attached to a class to specify its COM ProgID.

```
public sealed class ProgIdAttribute : Attribute { // Public Constructors
    public ProgIdAttribute(string progId); // Public Instance Properties
    public string Value { get; }
}
```

Hierarchy

System.Object > System.Attribute > ProgIdAttribute

Valid On

Class

RegistrationServices

This class is responsible for registering and unregistering assemblies with COM.

```
public class RegistrationServices : IRegistrationServices { // Public Constructors
    public RegistrationServices(); // Public Instance Methods
    public virtual Guid GetManagedCategoryGuid(); // implements IRegistrationServices
    public virtual string GetProgIdForType(Type type); // implements IRegistrationServices
    public virtual Type[] GetRegistrableTypesInAssembly(System.Reflection.Assembly assembly); // implements
IRegistrationServices
    public virtual bool RegisterAssembly(System.Reflection.Assembly assembly,
        AssemblyRegistrationFlags flags); // implements IRegistrationServices
    public virtual void RegisterTypeForComClients(Type type, ref Guid g); // implements IRegistrationServices
    public virtual bool TypeRepresentsComType(Type type); // implements IRegistrationServices
    public virtual bool TypeRequiresRegistration(Type type); // implements IRegistrationServices
    public virtual bool UnregisterAssembly(System.Reflection.Assembly assembly) // implements IRegistrationServices
}
```

RuntimeEnvironment

This type exposes static methods you can use to get information about the CLR's environment.

```
public class RuntimeEnvironment { // Public Constructors
    public RuntimeEnvironment(); // Public Static Properties
    public static string SystemConfigurationFile { get; } // Public Static Methods
    public static bool FromGlobalAccessCache(System.Reflection.Assembly a);
    public static string GetRuntimeDirectory();
    public static string GetSystemVersion();
}
```

SafeArrayRankMismatchException

This exception signals that a SAFEARRAY's rank does not match the rank in the method signature; it might be thrown when invoking a managed method.

```

public class SafeArrayRankMismatchException : SystemException { // Public Constructors
    public SafeArrayRankMismatchException();
    public SafeArrayRankMismatchException(string message);
    public SafeArrayRankMismatchException(string message, Exception inner); // Protected Constructors
    protected SafeArrayRankMismatchException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context);
}

```

Hierarchy

System.Object > System.Exception(System.Runtime.Serialization.ISerializable) > System.SystemException > SafeArrayRankMismatchException

SafeArrayTypeMismatchException

This exception signals that a SAFEARRAY's type does not match the type in the method signature; it might be thrown when invoking a managed method.

```

public class SafeArrayTypeMismatchException : SystemException { // Public Constructors
    public SafeArrayTypeMismatchException();
    public SafeArrayTypeMismatchException(string message);
    public SafeArrayTypeMismatchException(string message, Exception inner); // Protected Constructors
    protected SafeArrayTypeMismatchException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context);
}

```

Hierarchy

System.Object > System.Exception(System.Runtime.Serialization.ISerializable) > System.SystemException > SafeArrayTypeMismatchException

SEHException

This class is used as a wrapper for an unmanaged C++ exception that was thrown.

```

public class SEHException : ExternalException { // Public Constructors
    public SEHException();
    public SEHException(string message);
    public SEHException(string message, Exception inner); // Protected Constructors
    protected SEHException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context); // Public Instance Methods
    public virtual bool CanResume();
}

```

Hierarchy

System.Object > System.Exception(System.Runtime.Serialization.ISerializable) > System.SystemException > ExternalException > SEHException

StructLayoutAttribute

Use this attribute to control how the members of a class are laid out in memory. See [LayoutKind](#) for the possible values you can use with this attribute.

```

public sealed class StructLayoutAttribute : Attribute { // Public Constructors
    public StructLayoutAttribute(short layoutKind);
    public StructLayoutAttribute(LayoutKind layoutKind); // Public Instance Fields
    public CharSet CharSet;
    public int Pack;
    public int Size; // Public Instance Properties
    public LayoutKind Value { get; }
}

```

Hierarchy

System.Object > System.Attribute > StructLayoutAttribute

Valid On

Class, Struct

TypeLibVersionAttribute

This attribute specifies the exported type library's version.

```

public sealed class TypeLibVersionAttribute : Attribute { // Public Constructors
    public TypeLibVersionAttribute(int major, int minor); // Public Instance Properties
    public int MajorVersion { get; }
    public int MinorVersion { get; }
}

```

Hierarchy

System.Object > System.Attribute > TypeLibVersionAttribute

Valid On

Assembly

UnknownWrapper

Use this wrapper to pass a managed object into unmanaged code as type VT_UNKNOWN.

```

public sealed class UnknownWrapper { // Public Constructors
    public UnknownWrapper(object obj); // Public Instance Properties
    public object WrappedObject { get; }
}

```

UnmanagedType

This enumeration contains constant values that represent various unmanaged types.

```

public enum UnmanagedType {
    Bool = 2,
    I1 = 3,
    U1 = 4,
    I2 = 5,
    U2 = 6,
    I4 = 7,
    U4 = 8,
    I8 = 9,
    U8 = 10,
}

```

```

R4 = 11,
R8 = 12,
Currency = 15,
BStr = 19,
LPStr = 20,
LPWSTR = 21,
LPWSTR = 22,
ByValTStr = 23,
IUnknown = 25,
IDispatch = 26,
Struct = 27,
Interface = 28,
SafeArray = 29,
ByValArray = 30,
SysInt = 31,
SysUInt = 32,
VByRefStr = 34,
AnsiBStr = 35,
TBStr = 36,
VariantBool = 37,
FunctionPtr = 38,
AsAny = 40,
LPArray = 42,
LPStruct = 43,
CustomMarshaler = 44,
Error = 45
}

```

Hierarchy

System.Object > System.ValueType > System.Enum(System.IComparable, System.IFormattable, System.IConvertible) > UnmanagedType

Returned By

System.Reflection.Emit.UnmanagedTypeMarshal.{BaseType, GetUnmanagedType}, MarshalAsAttribute.Value

Passed To

System.Reflection.Emit.UnmanagedTypeMarshal.{DefineLPArray(), DefineSafeArray(), DefineUnmanagedTypeMarshal()}, MarshalAsAttribute.MarshalAsAttribute()

VarEnum

This enumeration contains constants that can be used with `MarshalAsAttribute.SafeArraySubType` to specify how to marshal arrays that are passed from managed to unmanaged code.

```

public enum VarEnum {
    VT_EMPTY = 0,
    VT_NULL = 1,
    VT_I2 = 2,
    VT_I4 = 3,
    VT_R4 = 4,
    VT_R8 = 5,
    VT_CY = 6,
    VT_DATE = 7,
    VT_BSTR = 8,
}

```

```

VT_DISPATCH = 9,
VT_ERROR = 10,
VT_BOOL = 11,
VT_VARIANT = 12,
VT_UNKNOWN = 13,
VT_DECIMAL = 14,
VT_I1 = 16,
VT_UI1 = 17,
VT_UI2 = 18,
VT_UI4 = 19,
VT_I8 = 20,
VT_UI8 = 21,
VT_INT = 22,
VT_UINT = 23,
VT_VOID = 24,
VT_HRESULT = 25,
VT_PTR = 26,
VT_SAFEARRAY = 27,
VT_CARRAY = 28,
VT_USERDEFINED = 29,
VT_LPSTR = 30,
VT_LPWSTR = 31,
VT_RECORD = 36,
VT_FILETIME = 64,
VT_BLOB = 65,
VT_STREAM = 66,
VT_STORAGE = 67,
VT_STREAMED_OBJECT = 68,
VT_STORED_OBJECT = 69,
VT_BLOB_OBJECT = 70,
VT_CF = 71,
VT_CLSID = 72,
VT_VECTOR = 4096,
VT_ARRAY = 8192,
VT_BYREF = 16384
}

```

Hierarchy

System.Object > System.ValueType > System.Enum(System.IComparable, System.IFormattable, System.IConvertible) > VarEnum

IExpando

This interface indicates a type whose members can be removed or added. The members are represented as `System.Reflection.MemberInfo` objects.

```

public interface IExpando : System.Reflection.IReflect { // Public Instance Methods
    public FieldInfo AddField(string name);
    public MethodInfo AddMethod(string name, Delegate method);
    public PropertyInfo AddProperty(string name);
    public void RemoveMember(System.Reflection.MemberInfo m);
}

```