# COM Interoperability

COM was designed for the very purpose of writing components in any COM-capable language/platform and consumption thereof in any (other) COM-capable language/platform. .NET is no exception, and it allows you to easily consume COM objects and expose .NET types as COM objects.

With COM interop, the basic idea is the same as in P/Invoke: you declare a managed representation of the COM object and the CLR created a wrapper object that handles the marshaling. There are two kinds of wrappers: Runtime Callable Wrapper (RCW) which enables managed code to use COM objects (see Figure 8-4), and COM Callable Wrapper (CCW) that enables COM code to call managed types (see Figure 8-5). Third party COM components are often shipped with a Primary Interop Assembly, which contains vendor-approved interop definitions and is strongly named (signed) and installed in the GAC. Other times, you can use the tlbimp.exe tool, which is part of the Windows SDK to auto-generate an interop assembly based on the information contained in the type library.

COM interop re-uses the P/Invoke parameter marshaling infrastructure, with some changes in defaults (e.g. a string is marshaled to `BSTR` by default), so the advice provided in the P/Invoke section of this chapter applies here as well.

COM has its own performance issues resulting from COM-specific particularities such as the apartment threading model and a mismatch between the the reference counted nature of COM and the .NET garbage collected approach.

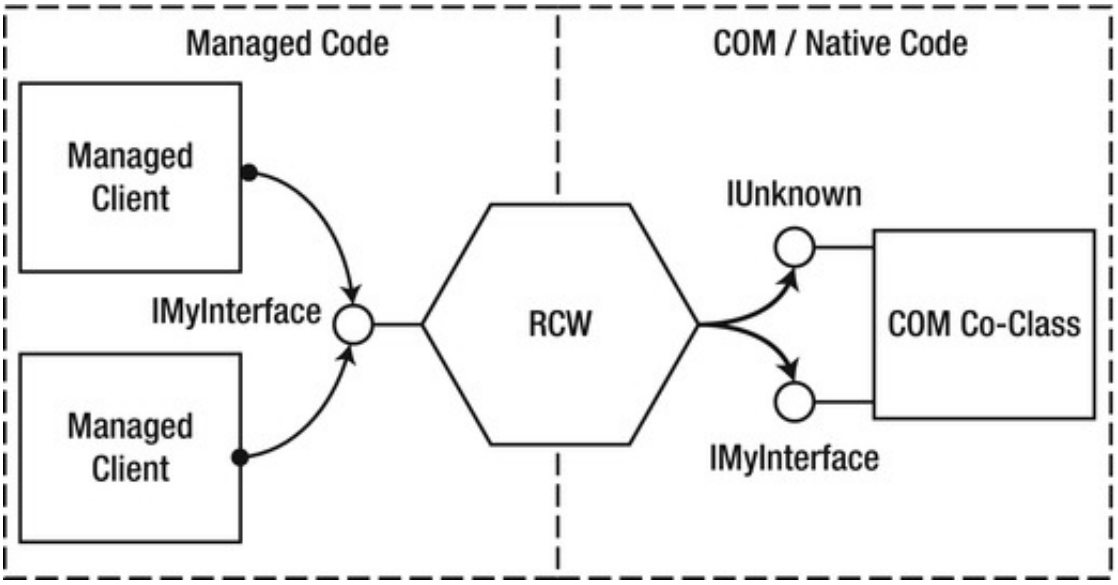


*Figure 8-4. Managed client calling unmanaged COM object*

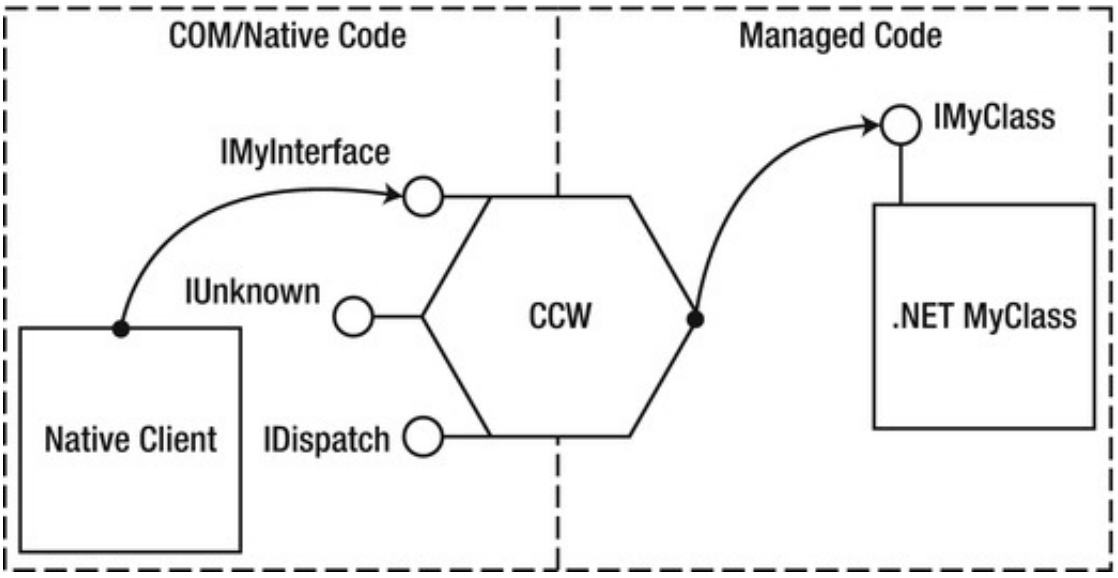


*Figure 8-5. Unmanaged client calling a managed COM object*

## Lifetime Management

When you hold a reference to a COM object in .NET, you are actually holding a reference to an RCW. The RCW always holds a single reference to the underlying COM object and there is only one RCW instance per COM object. The RCW maintains its own reference count,

separate from the COM reference count. This reference count's value is usually one but can be greater if a number of interface pointers have been marshaled or if the same interface has been marshaled by multiple threads.

Normally, when the last managed reference to the RCW is gone and there is a subsequent garbage collection at the generation where the RCW resides; the RCW's finalizer runs and it decrements the COM objects's reference count (which was one) by calling the `Release` method on the `IUnknown` interface pointer for the COM underlying object. The COM object subsequently destroys itself and releases its memory.

Since the .NET GC runs at non-deterministic times and is not aware of the unmanaged memory burden caused by it holding the RCWs and subsequently COM objects alive, it will not hasten garbage collections and memory usage may become be very high.

If necessary, you can call the `Marshal.ReleaseComObject` method to explicitly release the object. Each call will decrement RCW's reference count and when it reaches zero, the underlying COM object's reference count will be decremented (just like in the case of the RCW's finalizer running), thus releasing it. You must ensure that you do not continue to use the RCW after calling `Marshal.ReleaseComObject`. If the RCW reference count is greater than zero, you will need to call `Marshal.ReleaseComObject` in a loop until the return value equals zero. It is a best practice to call `Marshal.ReleaseComObject` inside a finally block, to ensure that the release occurs even in the case of an exception being thrown somewhere between the instantiation and the release of the COM object.

## Apartment Marshaling

COM implements its own thread synchronization mechanisms for managing cross-thread calls, even for objects not designed for multi-threading. These mechanisms can degrade performance if one is not aware of them. While this issue is not specific for interoperability with .NET, it is nonetheless worth discussing as it is a common pitfall, likely because developers that are accustomed to typical .NET thread synchronizations conventions might be unaware of what COM is doing behind the scenes.

COM assigns objects and threads *apartments* which are boundaries across which COM will marshal calls. COM has several apartment types:

- Single–threaded apartment (STA), each hosts a single thread, but can host any number of objects. There can be any number of STA apartments in a process.
- Multi–threaded apartment (MTA), hosts any number of threads and any number of objects,but there is only one MTA apartment in a process. This is the default for .NET threads.
- Neutral-threaded apartment (NTA), hosts objects but not threads. There is only one NTA apartment in a process.

A thread is assigned to an apartment when a call is made to `CoInitialize` or `CoInitializeEx` to initialize COM for that thread. Calling `CoInitialize` will assign the thread to a new STA apartment, while `CoInitializeEx` allows you to specify either an STA or an MTA assignment. In .NET, you do not call those functions directly, but instead mark a thread's entry point (or `Main`) with the `STAThread` or `MTAThread` attribute. Alternatively, you can call `Thread.SetApartmentState` method or the `Thread.ApartmentState` property before a thread is started. If not otherwise specified, .NET initializes threads (including the main thread) as MTA.

COM objects are assigned to apartments based on the *ThreadingModel* registry value, which can be:

- Single – object resides in the default STA.
- Apartment (STA) – object must reside in any STA, and only that STA's thread is allowed to call the object directly. Different instances can reside in different STA.
- Free (MTA) – object resides in the MTA. Any number of MTA threads can call it directly and concurrently. Object must ensure thread-safety.
- Both – object resides in the creator's apartment (STA or MTA). In essense, it becomes either STA-like or MTA-like object once created.
- Neutral – object resides in the neutral apartment and never requires marshaling. This is the most efficient mode.

Refer to Figure 8-6 for a visual representation of apartments, threads and objects.

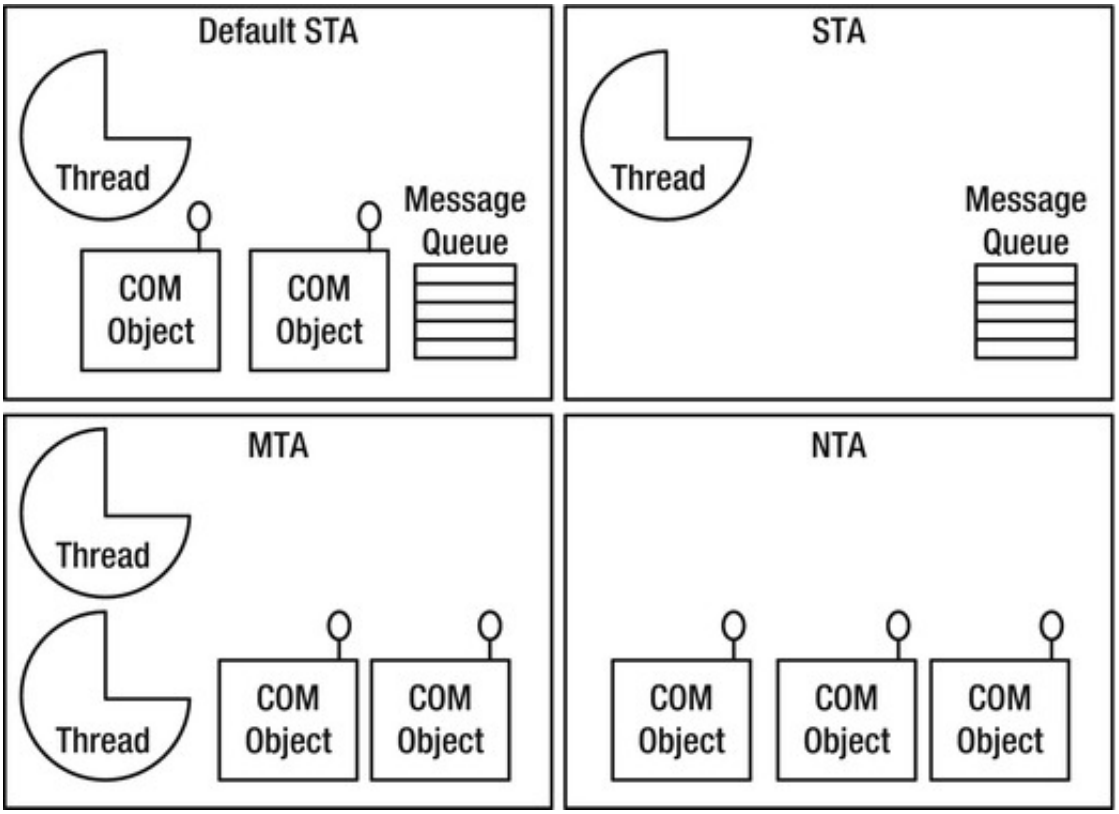


*Figure 8-6. Process division into COM apartments*

If you create an object with a threading model incompatible with that of the creator thread's apartment, you will receive an interface pointer which actually points to a proxy. If the COM object's interface needs to be passed to a different thread belonging to a different apartment, the interface pointer should not be passed directly, but instead needs to be marshaled. COM will return a proxy object as appropriate.

Marshaling involves translating the function's call (including parameters) to a message which will be posted to the recipient STA apartment's message queue. For STA objects, this is implemented as a hidden window whose window procedure receives the messages and dispatches calls to the COM object via a stub. In this way, STA COM objects are always invoked by the same thread, which is obviously thread-safe.

When the caller's apartment is incompatible with the COM object's apartment, a thread switch and cross-thread parameter marshaling occurs.

**Tip** Avoid the cross-thread performance penalty by matching the COM object's apartment with the creating thread's apartment. Create and use apartment threaded (STA) COM objects on STA threads, and free-threaded COM objects on MTA threads. COM objects marked as supporting both modes can be used from either thread without penalty.

**CALLING STA OBJECTS FROM ASP.NET**

ASP.NET executes pages on MTA threads by default. If you call STA objects, they undergo marshaling. If you predominantly call STA objects, this will degrade performance. You can remedy this by marking pages with the `ASPCOMPAT` attribute, as follows:

```
<%@Page Language = "vb" AspCompat = "true" %>
```

Note that page constructors still executes in an MTA thread, so defer creating STA objects to `Page_Load` and `Page_Init` events.

## TLB Import and Code Access Security

Code Access Security does the same security checking as in P/Invoke. You can use the `/unsafe` switch with the tlbimp.exe utility which will emit the `SuppressUnmanagedCodeSecurity` attribute on generated types. Use this only in full-trust environments as this may introduce security issues.

## NoPIA

Prior to .NET Framework 4.0, you had to distribute interop assemblies or Primary Interop Assemblies (PIA) alongside with your application or add-in. These assemblies tend to be large (even more so compared to the code that uses them), and they are not typically installed by the vendor of the COM component; instead they are installed as redistributable packages, because they are not required for the operation of the COM component itself. Another reason for not installing the PIAs is that they have to be installed in the GAC, which imposes a .NET Framework dependency on the installer of an otherwise completely native application.

Starting with .NET Framework 4.0, the C# and VB.NET compilers can examine which COM interfaces and which methods within them are required, and can copy and embed only the required interface definitions into the caller assembly, eliminating the need to distribute PIA DLLs and reducing code size. Microsoft calls this feature NoPIA. It works both for Primary Interop Assemblies and for interop assemblies in general.

PIA assemblies had an important feature called type equivalence. Since they have a strong name and are put into the GAC, different managed components can exchange the RCWs, and from .NET's perspective, they would have equivalent type. In contrast, interop assemblies generated by tlbimp.exe would not have this feature, since each component would have its own distinct interop assembly. With NoPIA, since there is a strongly named assembly is not used, Microsoft came up with a solution which treats RCWs from different assemblies as the same type, as long as the interfaces have the same GUID.

To enable NoPIA, select Properties on the interop assembly under References, and set “Embed Interop Types” to True (see Figure 8-7).

*Figure 8-7. Enabling NoPIA in interop assembly reference properties*

## Exceptions

Most COM interface methods report success or failure via an `HRESULT` return value. Negative `HRESULT` values (with most significant bit set) indicate failure, while zero (`S_OK`) or positive values report success. Additionally, a COM object may provide richer error information by calling the `SetErrorInfo` function, passing an `IErrorInfo` object which is created by calling `CreateErrorInfo`. When calling a COM method via COM interop, the marshaler stub converts the `HRESULT` into a managed exception according to the `HRESULT` value and the data contained inside the `IErrorInfo` object. Since throwing exceptions is relatively expensive, COM functions that fail frequently will negatively impact performance. You can suppress automatic exception translation by marking methods with the `PreserveSigAttribute`. You will have to change the managed signature to return an `int`, and the retval parameter will become an “out” parameter.