RICKY LEEKS PRESENTS THE TOP 5

TIPS FOR UNDERSTANDING MANAGED-UNMANAGED INTEROPERABILITY IN .NET

By: Michael B. McLaughlin

INTRODUCTION

Interop. The term alone can inspire fear, confusion, and uncertainty in many software developers. But there's really nothing to fear from interoperability between managed and unmanaged code, especially if you learn a bit about what's going on under the hood when you write interop code. Indeed, quite a few parts of the .NET Framework on Windows are implemented *as* platform invoke (P/Invoke) interop with native Windows DLLs[1].

P/Invoke is one of the two[2] types of interop you will come across as a developer. You will most commonly use it to access C-style functions in Win32 DLLs, and C and C++ code in in-house and 3rd party libraries. Considering the magical things it does, it is often surprisingly easy to implement P/Invoke in order to call some specific bit of native functionality that your application needs.

The other type of interop is COM interop, which Visual Studio actually provides many tools to make easy. COM has a long history and there are many components out there that use it. This article is primarily about the memory aspects of interop so there are many aspects of COM interop that we'll only briefly touch on and quite a few that we won't see at all (e.g. differences in error handling mechanisms[3]). If you need to dive deep into COM interop, the MSDN documentation is one of the best sources of information there is: http://msdn.microsoft.com/en-us/library/bd9cdfyx.aspx . I also recommend searching around for materials about creating Microsoft Office add-ins using C# and VB since that's a very rich source of COM interop information.

Interop code of either sort lets you access functionality in existing native and COM libraries that is not exposed by the .NET Framework. With it, you can take advantage of the speed and ease of .NET without needing to rewrite existing code that your projects depend on.

---

[1] Windows Forms, for example, is primarily a P/Invoke-based wrapper around ComCtl32.dll. See: http://msdn.microsoft.com/en-us/magazine/dd315414.aspx .

[2] In Windows 8, there's a new sheriff in Interop City. We'll talk more about this just a little further on.

[3] See: http://msdn.microsoft.com/en-us/library/9ztbc5s1.aspx .

A RELATIVELY BRIEF NOTE ON WINDOWS 8

Windows 8 introduces a new interface to the operating system called Windows Runtime (WinRT); iIt is the interface for writing Metro-style games and apps. At its core, WinRT is a friendlier, better, .NET-like version of COM. Now don't be scared! You don't need to learn COM or abandon .NET to use WinRT - That's where the "friendlier" and "better" parts come in. Interop in WinRT is designed to just work[4].

When using WinRT components, the CLR takes care of everything internally, regardless of whether it is native or .NET. The metadata files (i.e. those things with the .winmd extension) are consumed and the publicly exposed members are available just as if they were part of the .NET Framework itself.

From a memory management perspective, WinRT interop is primarily between different models of automatic memory management: reference counting (COM) and mark-and-sweep (.NET). WinRT natively uses reference counting to handle object life spans, which is where an internal count of the number of references to an object is maintained. When you create an object, the count starts at one. When something else references it, it goes up by one, and when that reference is released, it goes *down* by one. When the count reaches zero, the object is destroyed right then and there (this is what's referred to as "deterministic destruction", since you know exactly when it will happen). The CLR knows all about this and takes care of all of the reference counting for you when you use a native (non-CLR) WinRT component.

Surprisingly, this does not have much impact on the concepts examined in the remainder of this article, largely because WinRT is used only for Metro style apps. The Desktop-style interface in Windows 8 (the other mode) is designed to be an incremental improvement upon Windows 7 both in terms of design and display. Users of previous version of Windows will instantly recognize it, and Windows 8 seamlessly transitions between the two interfaces when you switch between Metro and Desktop applications. Metro style apps are only a good fit for some types of applications, and Win32, classic COM, WPF, Silverlight, and .NET applications and libraries will continue to be an important part of Windows for many years to come. So while you don't need to write any special code for WinRT interop, Desktop interop (P/Invoke and COM interop) will remain relevant and useful well into the future both for working with legacy code and for creating new Desktop style apps for Windows 8.

There's much more than can be said about Windows 8. I'm personally extremely impressed and excited by it having worked with it for a little while now. But this article is about *today's* interop, and so we had best get on with the topic at hand.

1. UNDERSTAND THE DIFFERENCE BETWEEN REFERENCES AND POINTERS.

In order to understand why much of the interop infrastructure works the way it does, we'll need a basic understanding of what references are (in .NET), and how the GC operates on them. One good way to do that is to first look at pointers, the ancestor of references, and then contrast them against .NET's

[4] http://blogs.msdn.com/b/somasegar/archive/2011/10/26/visual-studio-11-net-advances.aspx

references. We'll finish up with a brief look at reference-counting and mark-and-sweep (the two most common models for automatic memory management), and how .NET's choice of mark-and-sweep impacts interoperability with native code.

Pointers and References

References share enough similarities with pointers that it's easy to think of them *as* pointers, especially if you've come to .NET from writing native code in C/C++, where pointers are common place. But references are both more and less than pointers. While they both primarily hold memory addresses, they differ in the information they provide and the operations they enable.

A pointer is a variable that holds a memory address, but it has no concept of what you might find at that memory address. You can use pointer arithmetic to directly modify the memory address, which can be handy when the pointer is set to point to the beginning of an array. But the pointer doesn't have any idea of how many things are *in* that array, and you can set it to whatever value you like. However, should it ever start pointing to somewhere that you didn't intend, bad things will occur, the least of which would be your program immediately crashing[5].

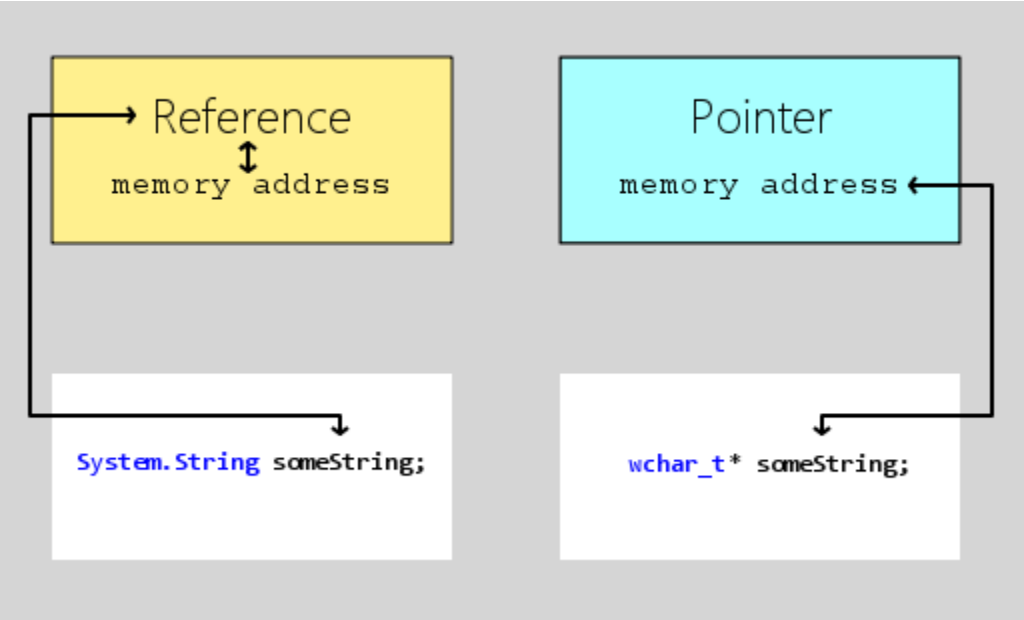

Figure 1 - A reference encapsulates a memory address, limiting the operations that can be performed on the address value to a language-specified subset. A pointer gives you unfettered access to the address itself, enabling all operations that can legally be performed on a native integer.

[5] In the event that your program doesn't crash, you will instead likely find yourself overwriting other data your program has created or, in the absence of something like Data Execution Prevention (http://msdn.microsoft.com/en-us/library/aa366553(VS.85).aspx),writing out something which will at some point be treated as executable code, thereby creating a security problem.

In .NET, a reference is also a variable that holds a memory address, but that memory address is not directly accessible from your code and can't be fiddled with. Because .NET code is type safe and garbage collected, the CLR always knows what type of data is at a given memory location. Using that information, the CLR prevents you from setting a reference to point to a location that isn't compatible with that reference's type, and it blocks you from running past the beginning or end of an array.

Memory Management Styles

In C and traditional C++, pointers were all there were. Modern C++, while retaining pointers, also makes it possible to use references and even supports automatic memory management via smart pointer types in the C++ Standard Library. It uses an automatic reference counting model, wherein it keeps track of how many references there are to an object, and frees that object when the count reaches zero[6]. As it happens, COM also uses reference counting to manage object lifetimes.

The .NET GC, in contrast, works on a mark-and-sweep model. Reference-type objects are allocated on one or more heaps maintained by the GC. It will periodically examine the object graph of your program, marking each object as it goes and thereby keeping track of what it has seen (and what is therefore still in-use). When it is done, anything not marked is known to be unreachable in your program, and so the memory occupied by these unmarked objects can safely be finalized[7] and freed (i.e. swept)[8].

At times the GC will also compact the heap, which involves moving objects around to fill in the holes left by previously freed objects. This helps to minimize memory fragmentation, and thus makes more memory available (or rather, usable) to your program. When the GC does a compaction, the memory addresses of any object it has moved will obviously change, and so it must go and fix up all of the references that point to that object so that they are still correct. The same sort of movement and reference-fixing process also happens with object promotion in generational GCs[9].

Knowing that the underlying memory address of a reference can change at any time[10] helps us understand why many of the interop memory operations we will be examining work the way they do. In native code, data never moves in memory unless the developer writes code that moves it. Libraries and

---

[6] For more on reference counting garbage collection, see: http://blogs.msdn.com/b/abhinaba/archive/2009/01/27/back-to-basics-reference-counting-garbage-collection.aspx .

[7] See: http://msdn.microsoft.com/en-us/library/0s71x931.aspx .

[8] Programs have various "roots", which are objects that serve as potential starting points for reaching other objects. This includes things like variables in the local execution stack and global variables. The object graph is the map of all reachable objects starting at these roots. For more on mark-and-sweep garbage collection, see: http://blogs.msdn.com/b/abhinaba/archive/2009/01/30/back-to-basics-mark-and-sweep-garbage-collection.aspx .

[9] Many of the .NET CLRs feature a generational GC which adds in object promotion in order to more quickly process short-lived objects. For more on this, see Clive Tong's article, *The Top 5 .NET Memory Management Misconceptions* (http://www.simple-talk.com/dotnet/performance/the-top-5-.net-memory-management-misconceptions/).

[10] Remember that the CLR will automatically start a GC collection whenever a condition that triggers one is met.

programs written in C and C++ implicitly rely on this behavior (it would be impossible to change without rewriting the languages themselves, which still wouldn't change all of the existing code).

Indeed, the three main parts of interop (from a memory management perspective) come down to handling the fact that data can move in .NET, dealing with different ways of representing common data types in .NET and native code (like strings), and accounting for memory management by reference counting in some native code (particularly COM). We will be examining .NET's machinery for dealing with these in the remainder of this article.

2. KNOW WHAT THE .NET MARSHALER DOES.

When data needs to go from managed code to unmanaged code, or vice-versa, the CLR's marshaling service takes over and makes sure that the data is correctly transferred. Some data types are the same in .NET and native code, such as **System.Single** in .NET and **float** in C++, and **System.Int64** in .NET and **long long** in C++[11]. These are called blittable types because they can be transferred without any conversion[12].

Other types are represented differently in .NET than they are in native code. A .NET **System.String**, for example, is different than a native **char*** string (and also differs slightly from a wchar_t* string)[13]. These types are called non-blittable because of their need for conversion before being transferred. We'll talk more about blittable versus non-blittable types in the next section. One thing to keep in mind about them is their memory implications.

Normally, the marshaler makes a copy of data that it is sending across the .NET – native divide. If you are operating on very large amounts of data, this can quickly become memory intensive. That said, for some data types, where the circumstances are appropriate and no conversion is required (this includes .NET Strings to Unicode native strings even though there is a small formatting difference), the marshaler will instead pin the data (telling the GC not to move it in the event of a compaction) and pass a pointer to the pinned address instead.

---

[12] See: http://msdn.microsoft.com/en-us/library/75dwhxf7.aspx .

[13] .NET strings are most similar to COM's BSTR in that they have an integer prefix that specifies the length of the string in bytes. This, combined with the fact that .NET strings are immutable is what makes a .NET string different from a wchar_t* string.

The rules around this are fairly technical and easier to look up than memorize, so we won't be discussing them here[14]. It's just useful to know that this exists for circumstances in which it might matter (especially if the native side tries messing with a pinned .NET string, since this is a good way to wind up with heap corruption). In most instances, the marshaler works automatically, but sometimes it requires

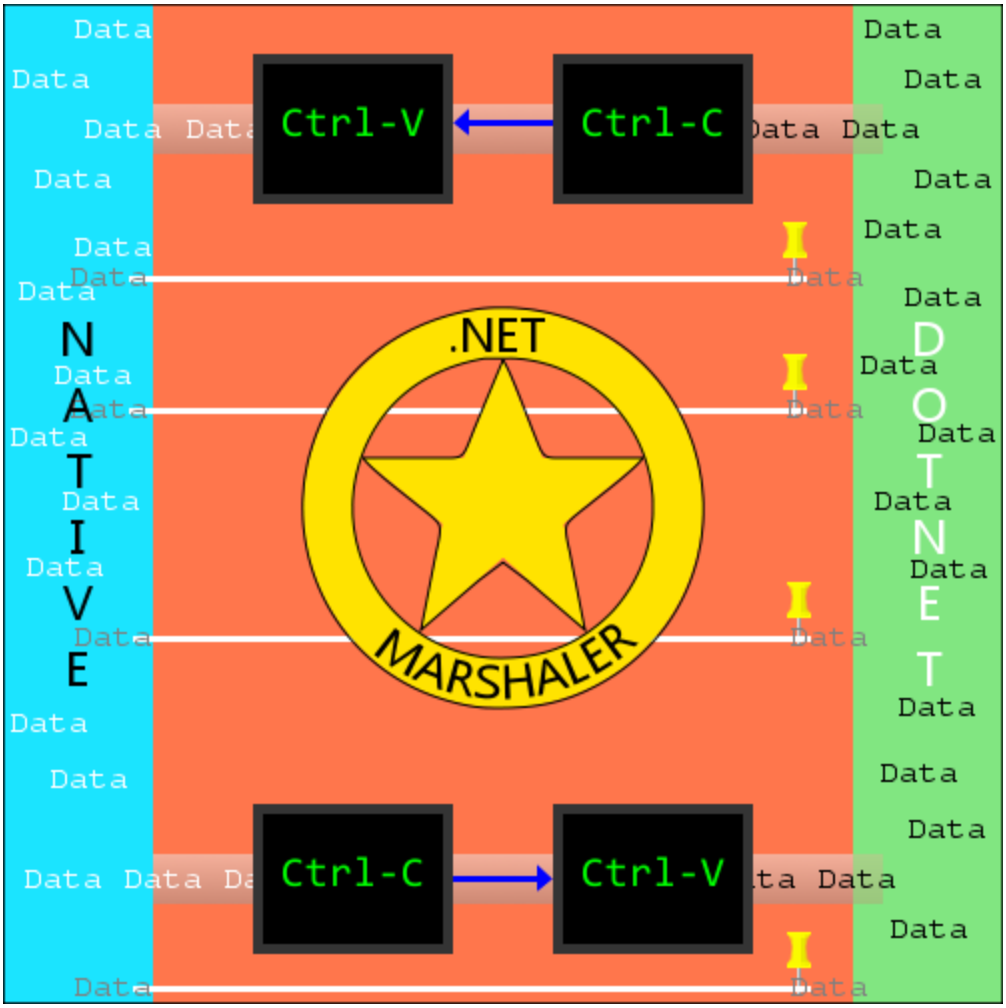

Figure 2 At its core, the .NET Marshaler is responsible for copying data back and forth between native and managed code. Where it is reasonable to do so, it pins .NET data (to prevent the GC from moving it) and passes its memory address directly across.

hints from the programmer via attributes (which we'll discuss a little later). There may also be occasions when you either need or want to access it directly. For these situations, the framework exposes the marshaling service through the **System.Runtime.InteropServices.Marshal** class.

[14] For more on copying and pinning, including the circumstances in which each occurs, see: http://msdn.microsoft.com/en-us/library/23acw07k.aspx .

3. BE AWARE OF THE ATTRIBUTES THAT GOVERN MARSHALING, AND THE DEFAULT TYPE EQUIVALENTS USED IN CONVERTING BETWEEN NATIVE AND MANAGED DATA TYPES.

Many Win32 API function calls already have P/Invoke wrapper code available for them through http://pinvoke.net/ and through the Windows API Code Pack[15]. Should you need to write your own P/Invoke methods (e.g. for a third party library or some in-house code), you'll likely find yourself needing to create .NET versions of native data structures. The marshaler already knows how to handle quite a few data types, so your job is mostly to create a struct with the .NET equivalents of the native types. For this you will want to bookmark and refer back to the MSDN Library's Windows Data Types page[16]. Not everything is intuitive (for example, BOOL is a System.Int32 but BOOLEAN is a System.Byte), so it's best to always consult the chart.

DIY Data Structures

When you *do* need to create a data structure, you'll need to apply a **System.Runtime.InteropServices.StructLayoutAttribute** to it, and specify a **LayoutKind** of either **LayoutKind.Sequential** or (more rarely) **LayoutKind.Explicit**. The CLR controls how data members in classes and structs are stored in memory, and will reorder them for more efficiency. By using a **StructLayoutAttribute** with one of those two layout kinds, you inform the CLR that the order you specified is important. **LayoutKind.Sequential** always ensures that your specified layout is used when data structures are passed to native code, but only uses it on the managed side when the struct consists entirely of blittable types (otherwise it uses its own order).

If you want to *guarantee* that you will always have the exact layout you've specified in both native and managed memory, you should use **LayoutKind.Explicit**. The downside of this is that 'explicit' means that you need to decorate each data field with a **System.Runtime.InteropServices.FieldOffsetAttribute** in order to tell the CLR the offset (in bytes) from the beginning of the struct's memory that the field should appear at. This requires you to know how many bytes each data type occupies and that you get all the calculations right! Sequential layout will normally fit your needs, but remember that explicit is an option if you need it for some reason.

Behavioral Control

In COM interfaces, you use Interface Definition Language attributes to specify various conditions such as what the interface methods are expecting from each of the parameters they receive, as well as whether the methods reserve the right to modify those parameters[17]. Outside of COM, the Win32 API and many

[15] See: http://archive.msdn.microsoft.com/WindowsAPICodePack .
[16] http://msdn.microsoft.com/en-us/library/aa383751(VS.85).aspx
[17] See: http://msdn.microsoft.com/en-us/library/aa367042(VS.85).aspx .

other native libraries make use of Source Code Annotation Language ("SAL") attributes to convey the same sort of information[18].

Of primary interest to us is the specification of **_in** and **_out** (or **_In_** and **_Out_**, etc.). Allow me to explain: The marshaler has default behaviors it uses for marshaling the parameters of a P/Invoke call. For instance, when dealing with a reference type that has non-blittable data members, the marshaler normally only marshals that data as 'in' data (i.e. data that only gets passed to the native function without any changes being propagated back to the .NET side)[19]. It does this because copying data across is expensive, in terms of both the time taken to convert data and the time taken to copy it.

The managed attributes **System.Runtime.InteropServices.InAttribute** and **System.Runtime.InteropServices.OutAttribute** exist to tell the marshaler to *ignore* its default behavior and handle marshaling function parameters based on which attributes you've applied. In general, I think it's a good idea to apply these attributes to all parameters for each P/Invoke method you write. Doing so makes it clear exactly how the data should be marshaled, rather than relying on behaviors that you may or may not remember correctly. It also makes it easier to track down bugs related to marshaling, since you can look right at the attributes, rather than back and forth between the MSDN documentation to try to figure out what might be going wrong.

4. KNOW HOW THE CLR MAKES IT POSSIBLE FOR COM AND .NET TO WORK WITH EACH OTHER.

As we discussed earlier, COM uses reference counting to manage object lifetimes. In COM, all objects derive from the **IUnknown** interface, which provides the **AddRef**, **QueryInterface**, and **Release** methods. **AddRef** and **Release** are used to increase and decrease the object's internal reference count respectively, and when the reference count gets to zero, the object is destroyed. The marshaler knows all about COM, and so takes care of increasing and decreasing reference counts for you.

Indeed, Visual Studio and .NET seamlessly automate most aspects of COM interop for you. In Visual Studio, you can include COM components in your project's references just like you would include .NET assemblies[20]. If you need finer control over the resulting go-between assembly, you can use **tlbimp.exe** (a command line tool) to control how a COM type library (**.TLB** file) should be converted. Since the conversion process creates a standard .NET assembly, you can reference it from your projects just like normal. And if there are a few small bits of the conversion that didn't work quite right (perhaps some parameter that came in as the wrong type) and you're feeling a bit brave, you can decompile the resulting assembly that **tlbimp.exe** creates, make changes to it, and recompile it, all with **ilasm.exe**[21].

---

[18] There was a change to SAL between Visual Studio 2005 and Visual Studio 2008. Newer code will likely use the modern version - http://msdn.microsoft.com/en-us/library/ms235402.aspx - while older code will normally use the old version - http://msdn.microsoft.com/en-us/library/ms235402(VS.80).aspx . For our purposes it doesn't matter which version is used; just know that there are two so you can look at the correct one as needed.

[19] See: http://msdn.microsoft.com/en-us/library/23acw07k.aspx .

[20] See: http://msdn.microsoft.com/en-us/library/fwawt96c.aspx .

[21] See: http://msdn.microsoft.com/en-us/library/56kh4hy7.aspx .

Working with raw CIL is perhaps a scary concept (it's the .NET equivalent of assembly code, after all). Using a tool like .NET Reflector can make these changes much easier to apply, since you can see the decompiled code in your language of choice and find the area that needs to be changed much more quickly. You can even switch between your language and CIL within Reflector to find the CIL equivalents of what you would expect to see in your preferred .NET language.

Whether you choose Visual Studio or **tlbimp.exe**, both create an assembly that contains something called a Runtime Callable Wrapper (RCW). An RCW *looks* like an ordinary .NET class, but actually serves as an intermediary between your .NET code and the COM component you want to use[22]. If, for some reason, the COM component needs to call one of your .NET methods (perhaps as a callback method), then the runtime can also create a COM Callable Wrapper (CCW) to pass to the COM object. As you might expect, the CCW wraps your .NET class and methods, and makes them look like a COM component to the COM object you are working with. No matter which wrapper is being used, the CLR tends to handle the memory aspects just fine.

5. UNDERSTAND FINALIZERS, IDISPOSABLE, AND 'USING' STATEMENTS.

In .NET, the release of previously allocated resources (such as memory) is non-deterministic. Non-deterministic is a fancy word for "*you don't know when those resources will become available again*". The reason for this is that resources are only freed up when the GC decides to release them, and the ways of the GC are dark and mysterious. Normally it releases resources when it runs a collection; objects that aren't identified as reachable in the mark phase then proceed to be finalized in the sweep phase.

Finalizing

All .NET objects have a Finalize method which they inherit from **System.Object**[23]. The default finalizer does nothing but, when necessary, you can override it and make it do something[24]. The GC only knows about managed resources. It has no idea about any unmanaged resources you might be working with (such as file streams and windows), but creating a finalizer provides you with a way to handle releasing those resources. Whenever you have an unmanaged resource that remains open outside the scope of the method in which it is allocated, you should implement a custom finalizer to make sure that resource is properly dealt with (i.e. freed up and returned once the object is no longer reachable within your program).

---

[22] See: http://msdn.microsoft.com/en-us/library/5dxz80y2.aspx and http://msdn.microsoft.com/en-us/library/8bwh56xe.aspx .

[23] See: http://msdn.microsoft.com/en-us/library/0s71x931.aspx .

[24] C# and C++/CLI both disallow a direct override of the Finalize method. Instead they each use a special syntax to note the finalizer. C# uses a destructor syntax - http://msdn.microsoft.com/en-us/library/66x5fx1b.aspx . C++/CLI uses a custom syntax that allows a class to have a normal C++ destructor in addition to a finalizer - http://msdn.microsoft.com/en-us/library/ms177197.aspx . Visual Basic lets you simply declare a Sub Finalize() method - http://msdn.microsoft.com/en-us/library/hks5e2k6.aspx .

But don't go writing finalizers for every class. The GC has a special mechanism for keeping track of objects which have a custom finalizer, and once they're subject to collection, they get added to a finalizer queue which runs finalizers sequentially on a separate thread. As a consequence, these objects wind up living longer than expected because the CLR itself keeps them alive[25]. On generational GCs, it also means they get promoted to the next generation, which can cause them to live for a *lot* longer than you would've expected. Write a finalizer if you need one, but if you can redesign your code to avoid needing one, consider doing that instead. You may find that you end up with cleaner code as a by-product.

Cleaning Up After Yourself

There's a special interface in .NET called **System.IDisposable** which you should know about. When a class implements it, it normally tells you that the class has unmanaged resources that should be released in a deterministic manner. An oft-used example is a file stream. When you ask .NET to open a file for you, it calls into the operating system, tries to open the file using the parameters you've specified and, assuming it succeeds, returns a nice .NET class representing that open file.

So far all is well. But let's imagine that there's some problem with the data in the file, such that your code throws an exception (or perhaps the disk it's writing to is full; any circumstance that raises an exception or otherwise breaks the control flow will do). Being a good developer, you have exception handling code to deal with just such a situation, so your application pops up a helpful error message informing the user of the problem, and then proceeds on its way. But there's another problem: you never closed that file. So, as far as the operating system is concerned, you still have it open.

The user now goes to try to examine the file in some other program to figure out what was wrong with it. The other program asks the operating system to open the file and… the operating system refuses! The user has no idea what's going on, begins to grow concerned that something horrible has happened to this vital file (maybe they begin thinking they have a virus), and thus panicking commences. All of this because of something that might be as simple as an XML file having some non-parseable header data that someone added to it by accident.

This situation will eventually resolve itself. It's possible that when the GC runs it will take care of it, but who knows when that will be. Calling **GC.Collect** all the time is both a bad idea (in terms of performance) and not guaranteed to fix your problem anyway. When your program closes, the OS will reclaim the leaked resource, but forcing the user to close your program just because an otherwise recoverable file operation has failed is not going to go over well with potential customers.

This is where **IDisposable** comes to the rescue. The **FileStream** class implements **IDisposable**, which in turn requires it to have a **Dispose** method, which is meant to be used to close these sorts of resources when properly implemented. It also makes it possible for C# developers to use a '**using**' statement[26]

[25] See: http://msdn.microsoft.com/en-us/library/ms973858.aspx#highperfmanagedapps_topic7 .
[26] See: http://msdn.microsoft.com/en-us/library/yh598w02.aspx .

with it (for VB programmers it's a '**Using**' statement[27]). This simple construct is translated by the compiler into an appropriate Try-Finally block. Specifically, one which guarantees that the unmanaged resources of the class implementing it (in this case an open file) are properly released, even in the case of an exception. It removes the need for you to write all sorts of calls to Close/Dispose[28] whenever you wish to break out of the scope in which the unmanaged resource wrapper exists, thus removing countless opportunities for bugs to creep in via forgotten calls to **Dispose** before returning or unanticipated exceptions.

When writing a class that requires a finalizer, you should always implement **IDisposable** as well. This gives other users of your code the opportunity to take advantage of the **using** statement syntax. It also lets you avoid a costly trip to the finalization queue. A proper **Dispose** method[29] will call **GC.SuppressFinalize** once it has cleaned up the unmanaged resources. This tells the GC that it doesn't need to run the custom finalizer for the object such that it can treat it like any other object. It will thus bypass a promotion to the next generation and the additional drain on resources that that would entail.

CONCLUSION

Interop is a fascinating world. .NET itself is, in many ways, one giant mass of interop. Whenever you use anything that implements **IDisposable**, there's a good chance that there's some interop going on behind the scenes. Of course, if you're working with .NET, analysis tools like ANTS Memory Profiler can make your life a lot easier, whereas there are fewer shortcuts available if you're working with unmanaged code or interop. That said, with Windows 8 and WinRT looming in the distance, interop is going to become both more important and much easier. Interop is a huge topic and one that can be a bit difficult to grasp if you've never worked with it before. Hopefully this article has helped shed a bit of light upon it for you and made it more approachable.

ABOUT THE AUTHOR

Michael B. McLaughlin is a Microsoft XNA/DirectX MVP, and the president of Bob Taco Industries, a small game and mobile app development firm. Until he retired in order to change careers a few years ago, he was an attorney. He has been programming computers for over 17 years, both as a hobby and professionally. You can reach him via email at mike@bobtacoindustries.com and can follow him on Twitter @mikebmcl.

[27] See: http://msdn.microsoft.com/en-us/library/htd05whh.aspx .

[28] Several .NET Framework classes have both a Close and a Dispose method. In most cases, Close is simply a wrapper for Dispose and is maintained solely for backward compatibility. When in doubt, refer the class library documentation. For more on the history of Close and Dispose, see: http://blogs.msdn.com/b/kimhamil/archive/2008/03/15/the-often-non-difference-between-close-and-dispose.aspx .

[29] See: http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx .