P/Invoke

Platform Invoke, better known as P/Invoke, enables calling C-style functions that are exported by DLLs from managed code. To use P/Invoke, the managed caller declares a `static extern` method, with a signature (parameter types and return value type) equivalent to those of the C function. The method is then marked with the `DllImport` attribute, while specifying at a minimum, the DLL which exports the function.

```
// Native declaration from WinBase.h:
HMODULE WINAPI LoadLibraryW(LPCWSTR lpLibFileName);

// C# declaration:
class MyInteropFunctions {
  [DllImport("kernel32.dll", SetLastError = true)]
  public static extern IntPtr LoadLibrary(string fileName);
}
```

In the preceding code, we define `LoadLibrary` as a function taking a `string` and returning an `IntPtr`, which is a pointer type that cannot be dereferenced directly, thus using it does not render the code unsafe. The `DllImport` attribute specifies that the function is exported by kernel32.dll (which is the main Win32 API DLL) and that the Win32 last error code should be saved so Thread Local Storage so that it will not be overwritten by calls to Win32 functions not done explicitly (e.g. internally by the CLR) . The `DllImport` attribute can also be used to specify the C function's calling convention, string encoding, exported name resolution options, etc.

If the native function's signature contains complex types such as C structs, then equivalent structures or classes must be defined by the managed code, using equivalent types for each field. Relative structure field order, field types and alignment must match to what the C code expects. In some cases, you will need to apply the `MarshalAs` attribute on fields, on function parameters or the return value to modify default marshaling behavior. For example, the managed `System.Boolean` (`bool`) type can have multiple representations in native code: the Win32 `BOOL` type is four bytes long and a true value is any non-zero value, while in C++ the `bool` value is one byte long and the true value equals 1.

In the following code listing, the `StructLayout` attribute that is applied to the `WIN32_FIND_DATA` struct, specifies that a sequential in-memory field layout is desired; without it, the CLR is free to re-arrange fields for better efficiency. The `MarshalAs` attributes applied to the `cFileName` and `cAlternativeFileName` fields specify that the strings should be marshaled as fixed-size strings embedded within the structure, as opposed to being just pointers to a string external to the structure.

```
// Native declaration from WinBase.h:
typedef struct _WIN32_FIND_DATAW {
  DWORD dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD nFileSizeHigh;
  DWORD nFileSizeLow;
  DWORD dwReserved0;
  DWORD dwReserved1;
  WCHAR cFileName[MAX_PATH];
  WCHAR cAlternateFileName[14];
} WIN32_FIND_DATAW;

HANDLE WINAPI FindFirstFileW(__in LPCWSTR lpFileName,
  __out LPWIN32_FIND_DATAW lpFindFileData);

// C# declaration:
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
struct WIN32_FIND_DATA {
  public uint dwFileAttributes;
  public FILETIME ftCreationTime;
  public FILETIME ftLastAccessTime;
  public FILETIME ftLastWriteTime;
  public uint nFileSizeHigh;
  public uint nFileSizeLow;
  public uint dwReserved0;
  public uint dwReserved1;
  [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)]
  public string cFileName;
  [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 14)]
  public string cAlternateFileName;
}

[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
static extern IntPtr FindFirstFile(string lpFileName, out WIN32_FIND_DATA lpFindFileData);
```

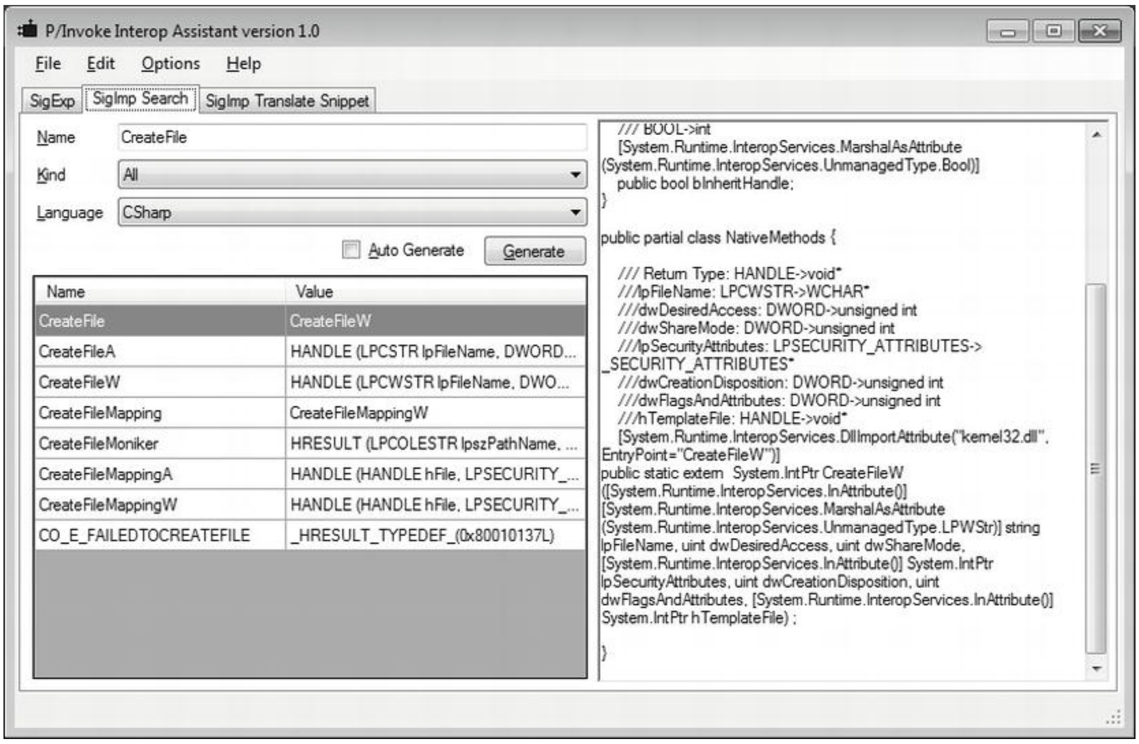
When you call the `FindFirstFile` method in the preceding code listing, the CLR loads the DLL that is exporting the function (kernel32.dll), locates the desired function (`FindFirstFile`), and translates parameter types from their managed to native representations (and vice versa). In this example, the input `lpFileName` string parameter is translated to a native string, whereas writes to the `WIN32_FIND_DATAW` native structure pointed to by the `lpFindFileData` parameter are translated to writes to the managed `WIN32_FIND_DATA` structure. In the following sections, we will describe in detail each of these stages.

PInvoke.net and P/Invoke Interop Assistant

Creating P/Invoke signatures can be hard and tedious. There are many rules to obey and many nuances to know. Producing an incorrect signature can result in hard to diagnose bugs. Fortunately, there are two resources which make this easier: the PInvoke.net website and the P/Invoke Interop Assistant tool.

PInvoke.net is a very helpful Wiki-style website, where you can find and contribute P/Invoke signatures for various Microsoft APIs. PInvoke.net was created by Adam Nathan, a senior software development engineer at Microsoft who previously worked on the .NET CLR Quality Assurance team and has written an extensive book about COM interoperability. You can also download a free Visual Studio add-on to access P/Invoke signatures without leaving Visual Studio.

P/Invoke Interop Assistant is a free tool from Microsoft downloadable from CodePlex, along with source code. It contains a database (an XML file) describing Win32 functions, structures, and constants which are used to generate a P/Invoke signature. It can also generate a P/Invoke signature given a C function declaration, and can generate native callback function declaration and native COM interface signature given a managed assembly.


P/Invoke Interop Assistant version 1.0
File Edit Options Help
SigExp SigImp Search SigImp Translate Snippet
Auto Generate
Generate

Name: CreateFile
Kind: All
Language: CSharp

Name	Value
CreateFile	CreateFileW
CreateFileA	HANDLE (LPCSTR lpFileName, DWORD...
CreateFileW	HANDLE (LPCWSTR lpFileName, DWO...
CreateFileMapping	CreateFileMappingW
CreateFileMoniker	HRESULT (LPCOLESTR lpszPathName, ...
CreateFileMappingA	HANDLE (HANDLE hFile, LPSECURITY_...
CreateFileMappingW	HANDLE (HANDLE hFile, LPSECURITY_...
CO_E_FAILEDTOCREATEFILE	_HRESULT_TYPEDEF_(0x80010137L)

```
    /// BOOL->int
    [System.Runtime.InteropServices.MarshalAsAttribute
(System.Runtime.InteropServices.UnmanagedType.Bool)]
    public bool bInheritHandle;
}

public partial class NativeMethods {

    /// Return Type: HANDLE->void*
    ///lpFileName: LPCWSTR->WCHAR*
    ///dwDesiredAccess: DWORD->unsigned int
    ///dwShareMode: DWORD->unsigned int
    ///lpSecurityAttributes: LPSECURITY_ATTRIBUTES->
_SECURITY_ATTRIBUTES*
    ///dwCreationDisposition: DWORD->unsigned int
    ///dwFlagsAndAttributes: DWORD->unsigned int
    ///hTemplateFile: HANDLE->void*
    [System.Runtime.InteropServices.DllImportAttribute("kernel32.dll",
EntryPoint="CreateFileW")]
public static extern  System.IntPtr CreateFileW
([System.Runtime.InteropServices.InAttribute()]
[System.Runtime.InteropServices.MarshalAsAttribute
(System.Runtime.InteropServices.UnmanagedType.LPWStr)] string
lpFileName, uint dwDesiredAccess, uint dwShareMode,
[System.Runtime.InteropServices.InAttribute()] System.IntPtr
lpSecurityAttributes, uint dwCreationDisposition, uint
dwFlagsAndAttributes, [System.Runtime.InteropServices.InAttribute()]
System.IntPtr hTemplateFile) ;

}
```

Figure 8-3. Screenshot of P/Invoke Interop Assitant showing the P/Invoke signature for CreateFile

Figure 8-3 shows Microsoft's P/Invole Interop Assistant tool showing the search results for "CreateFile" on the left side and the P/Invoke signature along with associated structures is shown on the right. The P/Invoke Interop Assistant tool (along with other useful CLR interop-related tools) can be obtained from `http://clrinterop.codeplex.com/` .

Binding

When you first call a P/Invoke'd function, the native DLL and its dependencies are loaded into the process via the Win32 `LoadLibrary` function (if they've not already been loaded). Next, the desired exported function is searched for, possibly searching for mangled variants first. Search behavior depends on the values of the `CharSet` and `ExactSpelling` fields of `DllImport`.

- If `ExactSpelling` is `true`, P/Invoke searches only for a function with the exact name, taking into account only the calling convention mangling. If this fails, P/Invoke will not continue searching for other name variations and will throw an `EntryPointNotFoundException`.
- If `ExactSpelling` is `false`, then behavior is determined by the `CharSet` property:
    - If set to `CharSet.Ansi` (default), P/Invoke searches the exact (unmangled) name first, and then the mangled name (with "A" appended).
    - If set to `CharSet.Unicode`, P/Invoke searches the mangled name first (with "W" appended), and then the unmanaged name.

The default value for `ExactSpelling` is `false` for C# and `True` for VB.NET. The `CharSet.Auto` value behaves like `CharSet.Unicode` on any modern operating system (later than Windows ME).

Tip Use the Unicode versions of Win32 functions. Windows NT and beyond are natively Unicode (UTF16). If you call the ANSI versions of Win32 functions, strings are converted to Unicode which results in a performance penalty, and the Unicode version of the function is called. The .NET string representation is also natively UTF16, so marshaling string parameters is faster if they are already UTF16. Design your code and especially the interface to be Unicode-compatible, which has globalization benefits as well. Set `ExactSpelling` to `true`, which will speed up initial load time by eliminating an unnecessary function search.

Marshaler Stubs

When you first call a P/Invoke'd function, right after loading the native DLL, a P/Invoke marshaler stub will be generated on demand, and will be re-used for subsequent calls. The marshaler performs the following steps once called:

1. Checks for callers' unmanaged code execution permissions.
2. Converts managed arguments to their appropriate native in-memory representation, possibly allocating memory.
3. Sets thread's GC mode to pre-emptive, so GC can occur without waiting the thread to reach a safe point.
4. Calls the native function.
5. Restores thread GC mode to co-operative.
6. Optionally saves Win32 error code in Thread Local Storage for later retrieval by `Marshal.GetLastWin32Error`.
7. Optionally converts `HRESULT` to an exception and throws it.
8. Converts native exception if thrown to a managed exception.
9. Converts the return value and output parameters back to their managed in-memory representations.
10. Cleans up any temporarily allocated memory.

P/Invoke can also be used to call managed code from native code. A reverse marshaler stub can be generated for a delegate (via `Marshal.GetFunctionPointerForDelegate`) if it is passed as a parameter in a P/Invoke call to a native function. The native function will receive a function pointer in place of the delegate, which it can call to invoke the managed method. The function pointer points to a dynamically generated stub which in addition to parameter marshaling, also knows the target object's address (`this` pointer).

In .NET Framework 1.x, marshaler stubs consisted of either generated assembly code (for simple signatures) or of generated ML (Marshaling Language) code (for complex signatures). ML is an internal byte code and is executed by an internal interpreter. With the introduction of AMD64 and Itanium support in .NET Framework 2.0, Microsoft realized that implementing a parallel ML infrastructure for every CPU architecture would be a great burden. Instead, stubs for 64-bit versions of .NET Framework 2.0 were implemented exclusively in generated IL code. While the IL stubs were significantly faster than the interpreted ML stubs, they were still slower than the x86 generated assembly stubs, so Microsoft opted to retain the x86 implementation. In .NET Framework 4.0, the IL stub generation infrastructure was significantly optimized, which made IL stubs faster compared even to the x86 assembly stubs. This allowed Microsoft to remove the x86-specific stub implementation entirely and to unify stub generation across all architectures.

Tip A function call that crosses the managed-to-native boundary is at least an order of magnitude slower than a direct call within the same environment. If you're in control of both the native and managed code, construct the interface in a way that minimizes native-to-managed round trips (chatty interfaces). Try to combine several "work items" into one call (chunked interfaces). Similarly, combine invocations of several trivial functions (e.g.trivial Get/Set functions) into a façade which does equivalent work in one call.

Microsoft provides a freely downloadable tool called *IL Stub Diagnostics* from CodePlex along with source code. It subscribes to the CLR ETW IL stub generation/cache hit events and displays the generated IL stub code in the UI.

Below we show an annotated example IL marshaling stub, consisting of five sections of code: initialization, marshaling of input parameters, invocation, marshaling back of return value and/or output parameters and cleanup. The marshaler stub is for the following signature:

```
// Managed signature:
[DllImport("Server.dll")]static extern int Marshal_String_In(string s);
// Native Signature:
unmanaged int __stdcall Marshal_String_In(char *s)
```

In the initialization section, the stub declares local (stack) variables, obtains a stub context and demands unmanaged code execution permissions.

```
// IL Stub:
// Code size    153 (0x0099)
.maxstack 3
// Local variables are:
// IsSuccessful, pNativeStrPtr, SizeInBytes, pStackAllocPtr, result, result, result
.locals (int32,native int,int32,native int,int32,int32,int32)

call native int [mscorlib] System.StubHelpers.StubHelpers::GetStubContext()
// Demand unmanaged code execution permissioncall void [mscorlib] System.StubHelpers.StubHelpers::DemandPermission(native in
```

In the marshaling section, the stub marshals input parameters native function. In this example, we marshal a single string input parameter. The marshaler may call helper types under the `System.StubHelpers`namespace or the `System.Runtime.InteropServices.Marshal` class to assist in converting of specific types and of type categories from managed to native representation and back. In this example, we call `CSTRMarshaler::ConvertToNative` to marshal the string.

There's a slight optimization here: if the managed string is short enough, it is marshaled to memory allocated on the stack (which is faster). Otherwise, memory has to be allocated from the heap.

```
   ldc.i4     0x0     // IsSuccessful = 0 [push 0 to stack]
   stloc.0     //     [store to IsSuccessful]
IL_0010:
       nop //     argument {
       ldc.i4     0x0     // pNativeStrPtr = null [push 0 to stack]
       conv.i     //     [convert to an int32 to "native int" (pointer)]
       stloc.3     //     [store result to pNativeStrPtr]
       ldarg.0     // if (managedString == null)
       brfalse     IL_0042     // goto IL_0042
       ldarg.0     // [push managedString instance to stack]
       // call the get Length property (returns num of chars)
       call     instance int32 [mscorlib] System.String::get_Length()
   ldc.i4     0x2     // Add 2 to length, one for null char in managedString and
       // one for an extra null we put in [push constant 2 to stack]
       add     // [actual add, result pushed to stack]
       //     load static field, value depends on lang. for non-Unicode
       //     apps system setting
   ldsfld System.Runtime.InteropServices.Marshal::SystemMaxDBCSCharSize
       mul     // Multiply length by SystemMaxDBCSCharSize to get amount of
   // bytes
   stloc.2 // Store to SizeInBytes
   ldc.i4 0x105 // Compare SizeInBytes to 0x105, to avoid allocating too much
   // stack memory [push constant 0x105]
       //     CSTRMarshaler::ConvertToNative will handle the case of
       //     pStackAllocPtr == null and will do a CoTaskMemAlloc of the
   // greater size
   ldloc.2 // [Push SizeInBytes]
   clt // [If SizeInBytes > 0x105, push 1 else push 0]
   brtrue IL_0042 // [If 1 goto IL_0042]
   ldloc.2 // Push SizeInBytes (argument of localloc)
   localloc // Do stack allocation, result pointer is on top of stack
   stloc.3 // Save to pStackAllocPtr
IL_0042:
   ldc.i4 0x1 // Push constant 1 (flags parameter)
       ldarg.0     //     Push managedString argument
   ldloc.3 // Push pStackAllocPtr (this can be null)
   // Call helper to convert to Unicode to ANSI
   call  native int [mscorlib]System.StubHelpers.CSTRMarshaler::ConvertToNative(int32,string,   native   int)
     stloc.1     // Store result in pNativeStrPtr,
   // can be equal to pStackAllocPtr
       ldc.i4 0x1     // IsSuccessful = 1 [push 1 to stack]
   stloc.0 // [store to IsSuccessful]
   nop
   nop
   nop
```

In the next section, the stub obtains the native function pointer from the stub context and invokes it. The call instruction actually does more work than we can see here, such as changing the GC mode and catching the native function's return in order to suspend execution of managed code while the GC is in progress and is in a phase that requires suspension of managed code execution.

```
   ldloc.1     // Push pStackAllocPtr to stack,
               //     for the user function, not for GetStubContext
   call     native int [mscorlib] System.StubHelpers.StubHelpers::GetStubContext()
   ldc.i4     0x14 // Add 0x14 to context ptr
   add     //     [actual add, result is on stack]
   ldind.i // [deref ptr, result is on stack]
   ldind.i     //     [deref function ptr, result is on stack]
   calli     unmanaged stdcall int32(native int) // Call user function
```

The following section is actually structured as two sections that handle the "unmarshaling" (conversion of native types to managed types) of the return value and output parameters, respectively. In this example, the native function returns an `int` which does not require marshaling and is just copied as-is to a local variable. Since there are no output parameters, the latter section is empty.

```
// UnmarshalReturn {
     nop   //   return {
     stloc.s   0x5   // Store user function result (int) into x, y and z
     ldloc.s   0x5
     stloc.s   0x4
     ldloc.s   0x4
     nop   // } return
     stloc.s   0x6
    //   } UnmarshalReturn
  // Unmarshal {
     nop // argument {
     nop // } argument
     leave   IL_007e // Exit try protected block
  IL_007e:
     ldloc.s   0x6   // Push z
     ret   //   Return z
  //   }   Unmarshal
```

Finally, the cleanup section releases memory that was allocated temporarily for the sake of marshaling. It performs cleanup in a `finally` block so that cleanup happens even if an exception is thrown by the native function. It may also perform some cleanup only in case of an exception. In COM interop, it may translate an `HRESULT` return value indicating an error into an exception.

```
// ExceptionCleanup {
IL_0081:
// } ExceptionCleanup
// Cleanup {
   ldloc.0 // If (IsSuccessful && !pStackAllocPtr)
   ldc.i4 0x0 // Call ClearNative(pNativeStrPtr)
   ble IL_0098
   ldloc.3
   brtrue IL_0098
   ldloc.1
   call void [mscorlib] System.StubHelpers.CSTRMarshaler::ClearNative(native int)
IL_0098:
   endfinally
IL_0099:
// } Cleanup
.try IL_0010 to IL_007e finally handler IL_0081 to IL_0099
```

In conclusion, the IL marshaler stub is non-trivial even for this trivial function signature. Complex signatures result in even lengthier and slower IL marshaler stubs.

Blittable Types

Most native types share a common in-memory representation with managed code. These types, called *blittable types*, do not require conversion, and are passed as-is across managed-to-native boundaries, which is significantly faster than marshaling non-blittable types. In fact, the marshaler stub can optimize this case even further by pinning a managed object and passing a direct pointer to the managed object to native code, avoiding one or two memory copy operations (one for each required marshaling direction).

A blittable type is one of the following types:

- `System.Byte` (`byte`)
- `System.SByte` (`sbyte`)
- `System.Int16` (`short`)
- `System.UInt16` (`ushort`)
- `System.Int32` (`int`)
- `System.UInt32` (`uint`)
- `Syste.Int64` (`long`)
- `System.UInt64` (`ulong`)
- `System.IntPtr`
- `System.UIntPtr`
- `System.Single` (`float`)
- `System.Double` (`double`)

In addition, a single-dimensional array of blittable types (where all elements are of the same type) is also blittable, so is a structure or class consisting solely of blittable fields.

A `System.Boolean` (`bool`) is not blittable because it can have 1, 2 or 4 byte representation in native code, a `System.Char` (`char`) is not blittable because it can represent either an ANSI or Unicode character and a `System.String` (`string`) is not blittable because its native representation can be either ANSI or Unicode, and it can be either a C-style string or a COM `BSTR` and the managed string needs to be immutable (which is risky if the native code modifies the string, breaking immutability). A type containing an object reference field is not blittable, even if it's a reference to a blittable type or an array thereof. Marshaling non-blittable types involves allocation of memory to hold a transformed version of the parameter, populating it appropriately and finally releasing of previously allocated memory.

You can get better performance by marshaling string input parameters manually (see the following code for an example). The native callee must take a C-style UTF-16 string and it should never write to the memory occupied by the string, so this optimization is not always possible. Manual marshaling involves pinning the input string and modifying the P/Invoke signature to take an `IntPtr` instead of a `string` and passing a pointer to the pinned `string` object.

```
class Win32Interop {
  [DllImport("NativeDLL.DLL", CallingConvention = CallingConvention.Cdecl)]
  public static extern void NativeFunc(IntPtr pStr); // takes IntPtr instead of string

}


//Managed caller calls the P/Invoke function inside a fixed scope which does string pinning:
unsafe
{
   string str = "MyString";
   fixed (char *pStr = str) {
   //You can reuse pStr for multiple calls.
   Win32Interop.NativeFunc((IntPtr)pStr);
   }
}
```

Converting a native C-style UTF-16 string to a managed string can also be optimized by using `System.String`'sconstructor taking a `char*` as parameter. The `System.String` constructor will make a copy of the buffer, so the native pointer can be freed after the managed string has been created. Note that no validation is done to ensure that the string only contains valid Unicode characters.

Marshaling Direction, Value and Reference Types

As mentioned earlier, function parameters can be marshaled by the marshaler stub in either or both directions. Which direction a parameter gets marshaled is determined by a number of factors:

- Whether the parameter is a value or reference type.
- Whether the parameter is being passed by value or by reference.
- Whether the type is blittable or not.
- Whether marshaling direction-modifying attributes (`System.RuntimeInteropService.InAttribute` and `System.RuntimeInteropService.OutAttribute`) are applied to the parameter.

For the purposes of this discussion, we define the "in" direction to be the managed to native marshaling direction; conversely the "out" direction is the native to managed direction. Below is a list of default marshaling direction rules:

- Parameters passed by value, regardless whether they're value or reference types, are marshaled in the "in" direction only.
    - You do not need to apply the `In` attribute manually.
    - `StringBuilder` is an exception to this rule and is always marshaled "in/out".

- Parameters passed by reference (via the `ref` C# keyword or the `ByRef` VB .NET keyword), regardless whether they're value or reference types, are marshaled "in/out".

Specifying the `OutAttribute` alone will inhibit the "in" marshaling, so the native callee may not see initializations done by the caller. The C# `out` keyword behaves like `ref` keyword but adds an `OutAttribute`.

Tip If parameters are not blittable in a P/Invoke call, and you require marshaling in only the "out" direction, you can avoid unnecessary marshaling by using the `out` C# keyword instead of the `ref` keyword.

Due to the blittable parameter pinning optimization mentioned above, blittable reference types will get an effective "in/out" marshaling, even if the above rules tell otherwise. You should not rely on this behavior if you need "out" or "in/out" marshaling behavior, but instead you should specify direction attributes explicitly, as this optimization will stop working if you later add a non-blittable field or if this is a COM call that crosses an apartment boundary.

The difference between marshaling value types versus reference types manifests in how they are passed on the stack.

- Value types passed by value are pushed as copies on the stack, so they are effectively always marshaled "in", regardless of modifying attributes.
- Value types passed by reference and reference types passed by value are passed by pointer.
- Reference types passed by reference are passed as pointer to a pointer.

Note Passing large value type parameters (more than a dozen or so bytes long) by value is more expensive than passing them by reference. The same goes for large return values, where out parameters are a possible alternative.

Code Access Security

The .NET Code Access Security mechanism enables running partially trusted code in a sandbox, with restricted access to runtime capabilities (e.g. P/Invoke) and BCL functionality (e.g. file and registry access). When calling native code, CAS requires that all assemblies whose methods appear in the call stack to have the `UnmanagedCode` permission. The marshaler stub will demand this permission for each call, which involves walking the call stack to ensure that all code has this permission.

Tip If you run only fully trusted code or you have other means to ensure security, you can gain a substantial performance increase by placing the `SuppressUnmanagedCodeSecurityAttribute` on the P/Invoke method declaration, a class (in which case it applies to contained methods), an interface or a delegate.
