# Interop

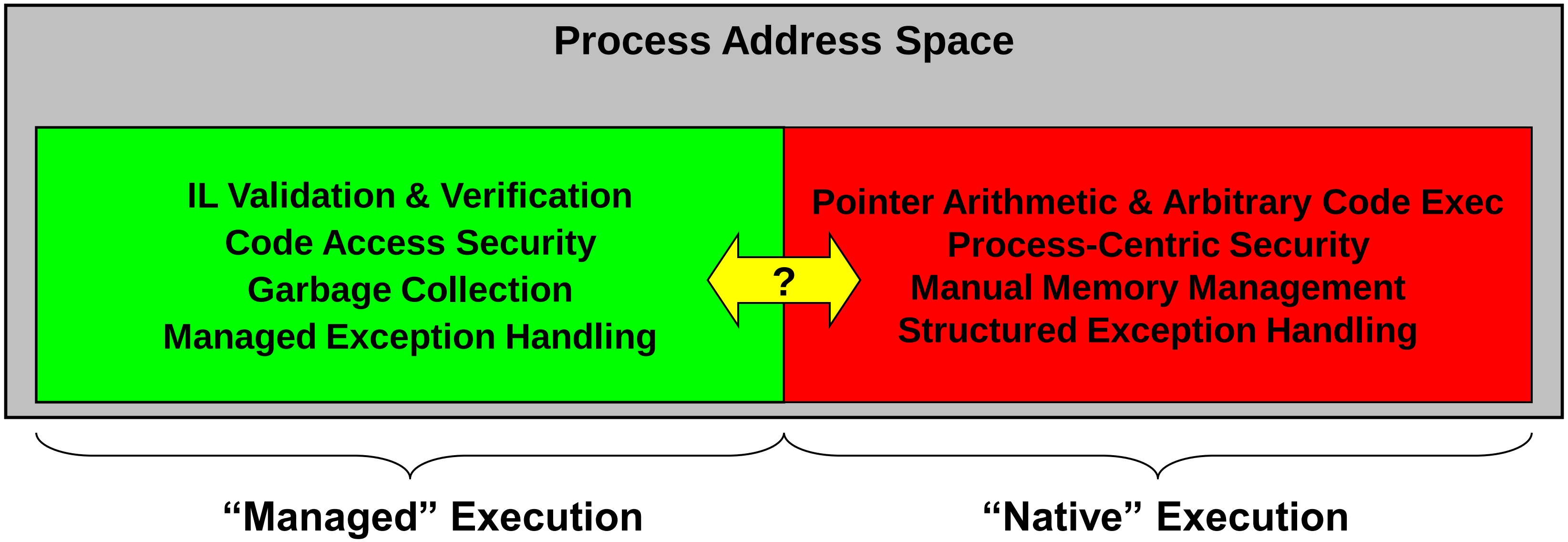
bridging the divide between managed and native code

# Outline

- **CLR support for managed/native code interop**
  - Comparing “managed execution” and “native execution”
  - The metadata-driven partnership
  - Mechanics of managed/native interop
    - CLR => Win32
    - CLR => COM
    - COM => CLR

# Comparing “Managed” & “Native” Execution

- **In practice, .NET apps involve a mix of managed & native code**
  - CLR-based “managed code”
  - Win32/COM-based “native code”
  - Transitions between the two worlds must be carefully coordinated
    - it’s more than simply parameter marshaling



# The Metadata-Driven Partnership

- **Metadata (type information) drives managed/native interop**
  - Managed type info can be derived from native type info
  - Native type info can be derived from managed type info
  - Pro: tools can automate **most** (not all) transformations
  - Con: tools can automate most (**not all**) transformations
- **Interop is a partnership involving you, the compiler, and the CLR**
  - The CLR (and it tools) will automate as much as possible
    - limited only by the fidelity of the native type information that's available
  - You may have to fill in some blanks or fine tune some transformations

# Interop Facilities

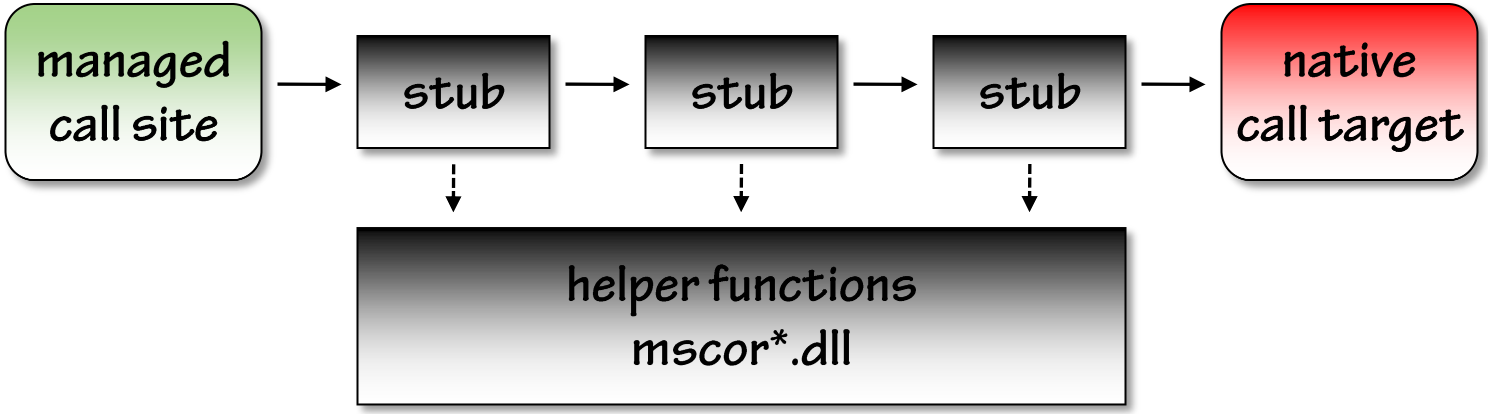
- **The CLR supports two kinds of managed/native interop**
  - Interop with Win32 DLLs
    - Platform Invocation Services
    - “P/Invoke”
  - Interop with COM components
    - “COM interop”

# P/Invoke

- **Managed code can load & call into Win32 DLLs**
  - Type information for Win32 DLLs is sorely lacking
    - export table contains only names & relative addresses of symbols
    - does not indicate whether exported symbol is a function or global variable
    - does not indicate "shape" of functions (its signature) or variables (its type)
  - The partnership
    - programmer describes function location & signature in terms of CLR types
    - managed compiler produces assembly/type metadata that drives JIT compilation
    - JIT compiler uses metadata to build "stubs" that perform CLR/Win32 transition
      - load the requested DLL (LoadLibrary)
      - locate the target function in memory (GetProcAddress)
      - marshal parameters to/from the target function

# P/Invoke Mechanics

- **The CLR/Win32 transition is carried out by *stubs* & *helpers***
  - *stubs* are little chunks of native code emitted by the JIT compiler
    - specific to the marshaling requirements dictated by the p/invoke declaration
  - *helpers* are native functions typically found in mscor*.dll
    - general purpose functions that don't need to be tuned specifically to the target
    - examples:
      - convert a CLR System.String parameter into NUL-terminated ANSI string
      - given a collection of CAS permissions, perform a demand/assert/etc



# P/Invoke Metadata

- **Programmer-supplied metadata drives P/Invoke**
  - Method prototype describes the native function in terms of CLR types
    - managed compiler emits metadata for the managed method (empty body)
    - managed method can be called like any other static method
  - JIT compiler uses metadata to load DLL and locate the exported method
  - JIT compiler uses metadata to build stubs that handle the transition

```
using System.Runtime.InteropServices;

class Program
{
    static void Main()
    {
        int sum = Add(2, 2);
    }

    [DllImport("nativecalc.dll")]
    static extern int Add(int a, int b);
}
```

# P/Invoke Fine Tuning

- **Stub generation can be fine-tuned as needed**
  - Using properties of DllImportAttribute
    - EntryPoint
    - CharSet
    - SetLastError
    - Others
  - By applying additional attributes to parameters and/or types
    - MarshalAsAttribute
    - StructLayoutAttribute

# Example: P/Invoke Fine Tuning

THE GOAL
Call this function from C#:

```
// Exported by weirdtextutils.dll
//
BSTR WINAPI Concat( wchar_t *s1, char *s2 );
```

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

class Program
{
    static void Main()
    {
        string s = Concat("Hello, ", "world!");
        Console.WriteLine(s);
    }

    [DllImport("weirdtextutils.dll")]
    [return: MarshalAs(UnmanagedType.BStr)]
    static extern string Concat(
        [MarshalAs(UnmanagedType.LPWStr)] string s1,
        [MarshalAs(UnmanagedType.LPStr)]  string s2
    );
}
```

Marshal the return value as a Basic string

Marshal s1 as a Unicode string.

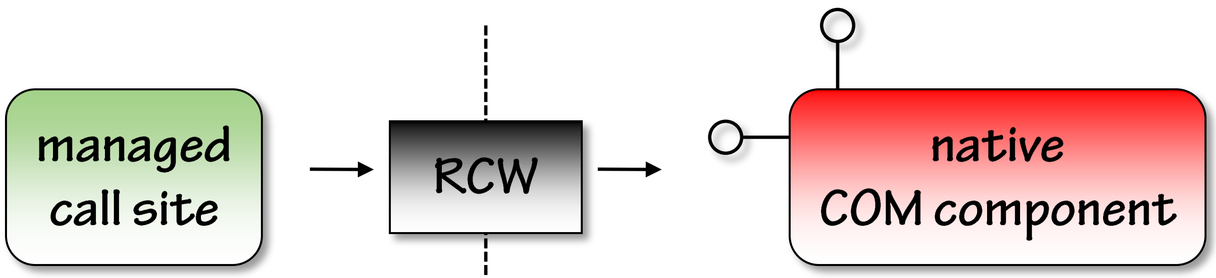
Marshal s2 as an ANSI string.

# COM Interop: CLR=>COM

- **Managed code can interact fairly easily with COM code**
  - COM type information (TLB) is fairly complete
- **The partnership**
  - TLBIMP.EXE translates COM type info into CLR type info
    - .TLB => TLBIMP.EXE => .DLL ("interop assembly")
    - COM coclass => CLR class (concrete)
    - COM interface => CLR interface
    - automated by VS.NET Add Reference wizard
  - Programmer adds a reference to interop assembly
  - CLR/JIT compiler use metadata to construct "runtime-callable wrappers"
    - RCWs

# Runtime-Callable Wrappers (RCW)

- RCWs bridge the divide between two very different worlds




| | | |
|---|---|---|
| Instantiation: | operator new | CLSID/ProgId registry mappings<br>CoCreateInstance |
| Type navigation: | is/as/cast | IUnknown::QueryInterface |
| Error handling: | System.Exception<br>throw/try/catch/finally | IErrorInfo<br>(Set\|Get)ErrorInfo<br>Win32 SEH |
| Memory mgmt: | shared heap<br>garbage collection | CoTaskMem(Alloc\|Free)<br>IUnknown::AddRef/Release |

# COM Type Information Deficiencies

- **COM type information (TLB) does not quite offer full fidelity**
- **Example**
  - COM IDL supports C-style "conformant arrays"
    - one parameter is a pointer to the first element in an array
    - another parameter specifies the element count
  - COM type libraries cannot represent conformant arrays
    - result is a pointer to a single entity, not the start of an array of entities

```
interface ICalc : IUnknown
{
  HRESULT Sum( [in, size_is(count)] long *pValues,
               [in] long count,
               [out, retval] long *pResult );
};
```

IDL

midl.exe

```
interface ICalc : IUnknown
{
  HRESULT Sum( [in] long *pValues,
               [in] long count,
               [out, retval] long *pResult );
};
```

TLB

tlbimp.exe

interop assembly

```
public interface ICalc
{
  int Sum( ref int pValues,
           int count );
};
```

Wrong!

# COM Interop Fine Tuning

- **RCW metadata can be provided manually when needed ala P/Invoke**
  - programmer manually describes types in CLR terms
  - attributes are used to provide required instantiation and marshaling details
  - CLR/JIT compiler uses programmer-supplied metadata to construct RCW

```
[ComImport, Guid("37DE3F74-99BE-4DD9-A06D-422203752987") ]
class CalcClass
{
  // empty
}


[ Guid("22E8E9BF-552E-4A09-8B93-33778020F240"),
  InterfaceType(ComInterfaceType.InterfaceIsIUnknown) ]
public interface ICalc
{
  int Sum(
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 1)] int[] values,
    int count
  );
}
```

Value taken from IDL/TLB

Adjust how RCW marshals the "values" parameter

```
class Program
{
  static void Main()
  {
    ICalc c = (ICalc)new CalcClass();
    int[] values = { 1, 2, 3, 4 };
    int sum = c.Sum(values, values.Length);
  }
}
```

# COM & Threads

- **COM has an "apartment model" construct for dealing with threads**
  - Components declare their preparedness/requirements re calling threads
    - ThreadingModel registry setting
  - Calling threads declare their preparedness/requirements re components
    - CoInitialize(Ex)
  - COM activation returns proxies or not based on those two settings
    - CoCreateInstance(Ex)
- **CLR threads have a property that declares their COM apartment state**
  - System.Threading.Thread.ApartmentState
  - System.STAThreadAttribute
  - System.MTAThreadAttribute

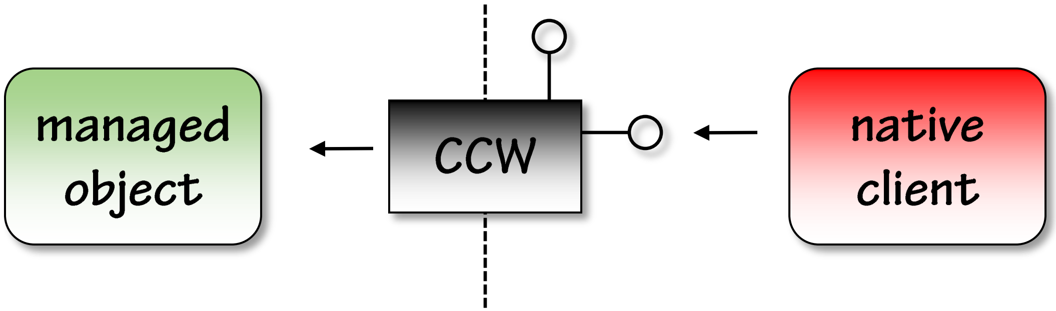
```
using System;

class Program
{
  [MTAThread]
  static void Main()
  {
    ICalc c = (ICalc)new CalcClass();
    int sum = c.Add(2, 3);
  }
}
```

# COM Interop: COM => CLR

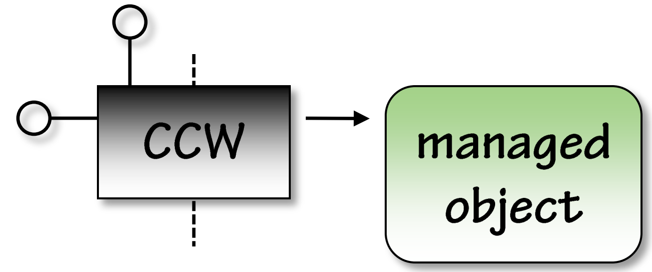
- **CLR type information offers high fidelity**
  - CLR-to-COM metadata transformations are often more easily performed
  - Does not mean every valid type can be represented in COM (e.g.: generics)
- **The partnership**
  - TLBEXP.EXE can be used to translate CLR type info into COM type info
    - .DLL (assembly) => TLBEXP.EXE => .TLB
    - suitable for use by VB6, MSFT C++ & other TLB-aware compilers
  - REGASM.EXE performs COM-required registration
    - all roads lead to MSCOREE.DLL
  - MSCOREE.DLL builds COM-callable wrappers (CCWs)
    - driven by metadata in .NET assembly (not the TLB)
  - .NET programmer can use attributes to fine tune tool operation

# COM-Callable Wrappers (CCW)

- **CCWs are essentially the inverse of RCWs**
  - Constructed by mscoree.dll during COM "activation"
  - Constructed whenever CLR object references are marshaled as parameters
  - COM-induced complexity made relatively seamless by the CLR



# COM Activation




VB6/VBA

```
Dim calc as Object
Set calc = CreateObject("Pluralsight.Calc")
```

JS

```
var calc = new ActiveXObject("Pluralsight.Calc");
```

Registry/HKCR/Pluralsight.Calc/CLSID

(Default) = {clsid}

ole32.dll

```
IUnknown *CoCreateInstanceEx( clsid, iid )
{
  dll = LoadLibrary(reg/Default);
  gco = GetProcAddress(dll, "DllGetClassObject");
  return gco(clsid, iid);
}
```

C++

```
ICalc *pCalc;
CoCreateInstance( CLSID_PluralsightCalc,
                  0, CLSCTX_INPROC_SERVER,
                  IID_ICalc, (void **)&pCalc );
```

Registry/HKCR/{clsid}/InprocServer32

(Default) = mscoree.dll
ThreadingModel = Both
RuntimeVersion = v2.0.50727
Assembly = pscalc, Version=1.0.0.0, Culture=neutral, …
Class = Pluralsight.Calc

mscoree.dll

```
IUnknown *DllGetClassObject( clsid, iid )
{
  clr = StartCLR(reg/RuntimeVersion);
  asm = clr.LoadAssembly(reg/Assembly);
  type = asm.GetType(reg/Class);
  obj = asm.CreateInstance(type);
  return BuildRCWForCLRObject(type, obj);
}
```

# COM Activation and Assembly Resolution

- **COM activation & Win32 DLL loading rules affect COM=>CLR interop**
  - MSCOREE.DLL!DllGetClassObject can be called in arbitrary process contexts
  - MSCOREE.DLL resides in %SystemRoot%\System32
  - MSCOREE.DLL needs to locate (on disk or in memory) and load assemblies
    - i.e., perform assembly resolution & everything that entails
- **Implications**
  - Four-part assembly names are stored in HKCR/{clsid}/InprocServer32
  - Assemblies intended to be used from COM should be signed & in GAC
  - Assemblies referenced/used by “top level” assembly should also be in GAC
  - REGASM.EXE can add a CodeBase hint to the registry
    - intended only for quick & dirty testing
    - always results in a warning
    - may still experience failures at runtime

# Native Resource Management

- **Native resource reclamation is manual/explicit**
  - free (C), delete (C++), CloseHandle (Win32), IUnknown::Release (COM)
- **Managed resource reclamation is more intelligent**
  - Garbage collection takes care of reclaiming unreachable object memory
- **Managed classes that interact directly with native code need to help**
  - P/Invoke clients ***should*** override System.Object.Finalize
    - i.e., make your type *finalizable* (language-specific syntax)
    - use P/Invoke to release native resource(s)
    - do ***not*** interact with any managed objects you may be referencing
  - RCW & P/Invoke clients ***may*** implement System.IDisposable
    - enables aggressive/early native resource cleanup
    - many languages provide an exception-aware IDisposable idiom

# Native Resource Management: CLR => COM

- **RCWs handle the direct interaction with COM objects**
  - Hold the actual interface pointer
  - Are finalizable
  - Will perform the final IUnknown::Release when finalized during GC
- **Classes that interact with RCWs may optionally spur final Release**
  - System.Runtime.InteropServices.Marshal.ReleaseComObject

Reference to RCW

Reference to managed object that implements IDisposable

```
class Widget : IDisposable
{
    INativeWidget _w = new NativeWidgetClass();
    Stream _s = File.OpenWrite(@"c:\widget.log");
    bool _disposed = false;

    public void Dispose()
    {
        if (!_disposed)
        {
            Marshal.ReleaseComObject(_w);
            _s.Dispose();
            _disposed = true;
        }
    }
}
```

NOTE
ReleaseCOMObject is not just *a call* to IUnknown::Release, it is *the final call* to IUnknown::Release.

# Native Resource Management: CLR => Win32

```
partial class Widget : IDisposable
{
  [DllImport("kernel32.dll")]
  static extern IntPtr CreateFileMapping(string filename);

  [DllImport("kernel32.dll")]
  static extern bool CloseHandle(IntPtr h);
}
```

p/invoke declarations simplified for brevity

```
partial class Widget : IDisposable
{
  IntPtr _h = CreateFileMapping(@"c:\widgetdata.bin");
  Stream _s = File.OpenWrite(@"c:\widget.log");
  bool _disposed = false;

  public void Dispose()
  {
    if (!_disposed)
    {
      CloseHandle(_h);
      _s.Dispose();
      GC.SuppressFinalize(this);
      _disposed = true;
    }
  }

  ~Widget()
  {
    CloseHandle(_h);
  }
}
```

Holds a Win32 HANDLE

Reference to managed object that implements IDisposable

release native resource
forward call to Dispose
suppresses finalization

Finalizer releases only the native resource

# Summary

- **Managed/native interop is a metadata-driven process**
  - Programmers supply all metadata for Win32 interop
  - Tools supply most metadata for COM interop
  - Attributes can be used to fine-tune stub, RCW and CCW construction
  - Some awareness of COM idioms is required
  - Leverage finalization & IDisposable as needed in first-level interop classes

# References

- **Useful references**
  - *.NET and COM: The Complete Interoperability Guide*
    - Adam Nathan, Microsoft Principal Developer
  - P/Invoke Interop Wiki
    - http://www.pinvoke.net
  - CloudBerry S3 Utility
    - http://www.cloudberrylab.com