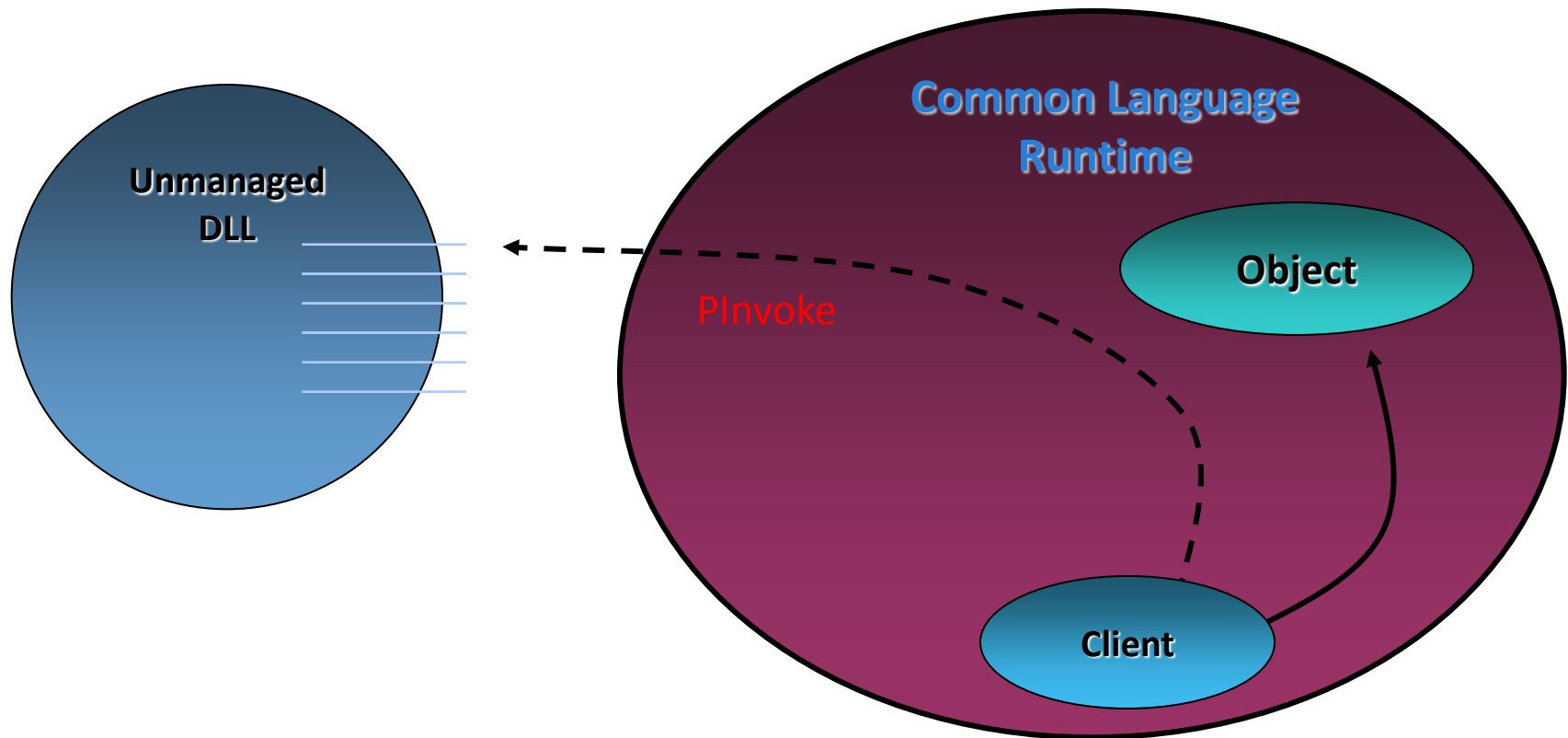


# PlInvoke

*Platform Invocation Service*

# Platform Invoke

Managed application



# Interop Challenges: Different object models

## Unmanaged

- C style functions
- C++ objects

## Managed

- .NET objects

# Interop Challenges: Data Types

- Data isn't the same
  - Different data types
    - Char\* vs. string
  - Different Owners
    - Programmer owns unmanaged data
    - GC owns managed data
- Transformation of data from one type to another type is called Marshalling.

# Platform Invoke (PInvoke)

- Provides access to static entry points in unmanaged DLLs from managed code (e.g. C#)
- Similar to: **LoadLibrary()** + **GetProcAddress()**
- Requires method definition with custom attribute
- Uses same underlying marshaling service as COM Interop
- The Platform Invocation service is part of the CLR.


# The DLL Import Attribute

- The WinAPI function you want to call from C#:

```
// C - prototype  
BOOL MessageBeep( UINT uType /* beep type */ );
```

- The code you'll need to add to a method definition in C# in order to call MessageBeep:

```
using System.Runtime.InteropServices; // Needed  
  
public class Wrapper {  
    [DllImport("User32.dll")]  
    public static extern Boolean MessageBeep(UInt32 beepType);  
}
```



The DllImportAttribute's primary role is to indicate to the CLR which DLL exports the function that you want to call.

- A possible call from managed code might look like this:

```
Wrapper.MessageBeep(0);
```

# DllImportAttribute's optional properties

- EntryPoint
  - You can set this property to indicate the entry point name of the exported DLL function in cases where you do not want your extern managed method to have the same name as the DLL export.
- CharSet
  - Set the CharSet property to CharSet.Auto. This causes the CLR to use the appropriate character set based on the host OS. If you don't explicitly set the CharSet property, then its default is CharSet.Ansi.
- SetLastError
  - This causes the CLR to cache the error set by the API function after each call to the extern method. Then, in your wrapper method, you can retrieve the cached error value by calling the `Marshal.GetLastWin32Error` method.
- CallingConvention
  - This property lets you indicate to the CLR which function calling convention it should use for parameters on the stack.

# What is a CallingConvention

**Calling conventions describe the interface of called code:**

- The order in which parameters are allocated.
- How parameters are passed:
  - pushed on the stack,
  - placed in registers, or
  - a mix of both
- Which registers may be used by the callee without first being saved (i.e. pushed) .
- How the task of setting up for and restoring the stack after a function call is divided between the caller and the callee.



# MS Argument Passing and Naming Conventions

- Calling conventions supported by the Visual C/C++ compiler

Keyword	Stack cleanup	Parameter passing
<b>__cdecl</b>	Caller	Pushes parameters on the stack, in reverse order (right to left)
<b>__clrcall</b>	n/a	Load parameters onto CLR expression stack in order (left to right).
<b>__stdcall</b>	Callee	Pushes parameters on the stack, in reverse order (right to left)
<b>__fastcall</b>	Callee	Stored in registers, then pushed on stack
<b>__thiscall</b>	Callee	Pushed on stack; this pointer stored in ECX

# cdecl CallingConvention

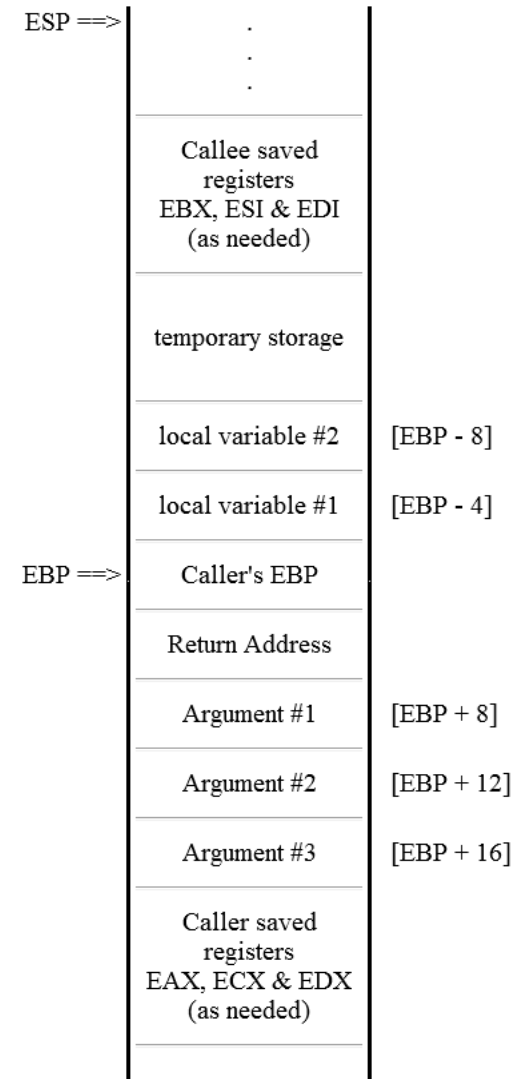
- cdecl (C declaration) is a calling convention that originates from the C programming language.
- Subroutine arguments are passed on the stack – from right to left.
- Integer values and memory addresses are returned in the EAX register,
  - floating point values—in the ST0 x87 register.
- Registers EAX, ECX, and EDX are caller-saved,
  - and the rest are callee-saved.
- The caller must do the stack cleanup
  - Supports vararg functions.
- Note:

There are some compiler variations in the interpretation of cdecl, particularly in how to return values.

# cdecl CallingConvention

```
int Callee(int arg1, int arg2, int arg3)
{
    int var1;
    int var2;
    res = var1 + var2 + var3;
    return res;
}

void Caller()
{
    int x;
    x = Callee(1, 2, 3);
}
```



# stdcall CallingConvention

- stdcall (standard call) calling convention is used to call Win32 API functions.
- Subroutine arguments are passed on the stack – from right to left.
- Integer values and memory addresses are returned in the EAX register,
  - floating point values—in the ST0 x87 register.
- Registers EAX, ECX, and EDX are caller-saved,
  - and the rest are callee-saved.
- The **callee** must do the stack cleanup.
  - vararg functions not possible.
  - Smaller and faster code, but less flexible

# Platform Invoke Example #2

The MessageBox function is defined in the Windows DLL *user32.dll* as follows:

```
int MessageBoxA (HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

```
public class Win32API
{
    [DllImport("User32.dll", EntryPoint="MessageBoxA",
               CharSet=CharSet.Ansi)]
    static extern Boolean MsgBox(IntPtr h, string m,
                                string c, int t);

    public static int Main()
    {
        return MsgBox(IntPtr.Zero, "Hello World!",
"Caption", 0);
    }
}
```

(On 32 bit systems you may map a hWnd to an int)

# Marshalling

- Strings are copied
  - Converted to platform format: ANSI or Unicode
  - Strings are never copied back – use `StringBuilder`
- Interfaces supported
  - Cannot be used as out parameter
- Arrays of primitive types can be used
- Custom marshalling
  - Much can be done using attributes
    - E.g. `[MarshalAs(UnmanagedType.LPArray)]`

# Marshalling Strings

- C Prototype:

```
void PrintString(char* pszString);
```

- C# Function Definition

```
[DllImport("mylib.dll")]  
public static extern void PrintString(  
    [MarshalAs(UnmanagedType.LPStr)] string s);
```

- Marshaler does character conversion
  - Except LPWStr, where it passes pointer to managed string
- You can marshal a string to unmanaged code as either a LPStr, a LPWStr, a LPTStr, or a BStr.
- Use **StringBuilder** for output strings.

# Marshaling Classes and Structs

Sometimes you need to pass a struct to an unmanaged method.

```
[StructLayout(LayoutKind.Sequential)]
```

```
Public struct OSInfo (  
    uint MajorVersion;  
    uint MinorVersion;  
    String VersionString;  
}  
  
public class Win32API {  
    [DllImport("User32.dll")]  
    public static extern Boolean GetVersionEx(OSInfo osi);  
}
```

**LayoutKind.Sequential** means that we want the fields aligned sequentially on pack-size boundaries, just as they would be in a C struct.

The field names here are irrelevant - it's the ordering of fields that's important.



# Showcase for the Standard Marshaller

- Given a method in some DLL  
`int SomeFunc( int sel, void* buf, int size);`
- Standard declaration may need special decoding  
`[DllImport("some.dll")]  
static extern int SomeFunc(int sel,  
[MarshalAs(UnmanagedType.LPArray)] byte[] buf,  
int size);`
- May use tailored versions  
`[DllImport("some.dll", EntryPoint="SomeFunc")]  
static extern int SomeFunc_String(int sel,  
StringBuilder sb, int size);`  
`[DllImport("some.dll", EntryPoint="SomeFunc")]  
static extern int SomeFunc_Int(int sel,  
ref int i, int size);`

# The Marshal Class

- When the `MarshalAs` attribute isn't enough you can turn to the Marshal class.
- The Marshal Class provides a collection of static methods for:
  - allocating unmanaged memory,
  - copying unmanaged memory blocks, and
  - converting managed to unmanaged types,
  - And other miscellaneous methods used when interacting with unmanaged code.
- The methods defined on the Marshal class are essential to working with unmanaged code, e.g.:  
`StringToHGlobalAnsi()`  
*copies ANSI characters from a specified string (in the managed heap) to a buffer in the unmanaged heap. It also allocates the target heap of the right size.*
- Namespace:  
`System.Runtime.InteropServices`

# Unmanaged Memory

- Use the Marshal class to allocate unmanaged memory from within .Net:

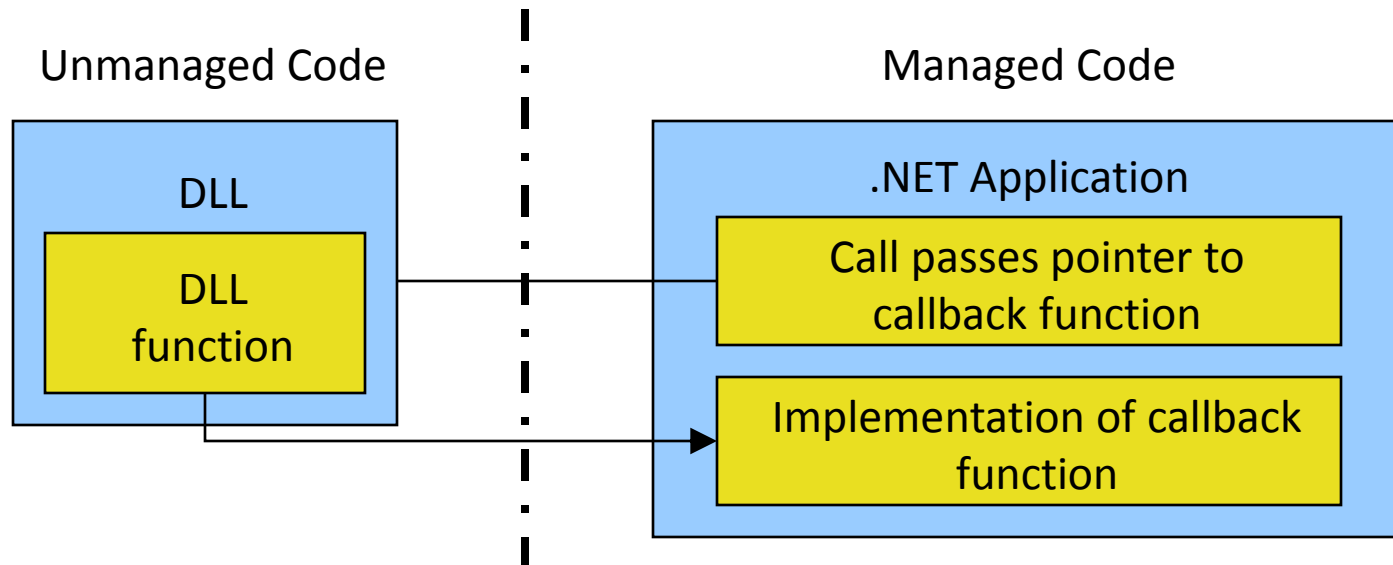
```
// Allocate some unmanaged memory from .Net
IntPtr hglobal = Marshal.AllocHGlobal(100);

// Use the unmanaged memory
// (send the pointer to some unmanaged code).

// And then free the memory again when finished.
Marshal.FreeHGlobal(hglobal);
```

# Callbacks

- Unmanaged code can call back to managed code
  - Unmanaged parameter is function pointer
  - Must supply parameter as delegate
  - PInvoke creates callback thunk
    - Passes address of thunk as callback parameter



# Code Sample

```
public class EnumReport
{
    public bool Report(int hwnd, int lParam)
    { // report the window handle
        Console.Write("Window handle is ");
        Console.WriteLine(hwnd);
        return true;
    }
};

public class SampleClass
{
    delegate bool Callback(int hwnd, int lParam);
    [DllImport("user32")]
    static extern int EnumWindows(Callback x, int y);
    public static void Main()
    {
        EnumReport er = new EnumReport();
        Callback myCallback = new Callback(er.Report);
        EnumWindows(myCallback, 0);
    }
}
```

# Performance Considerations

- Transitions have overhead
  - Roughly 30 instruction per call
- Data marshaling adds additional overhead
  - Depending on type and size of data
  - Isomorphic types are cheap
- Make transitions wisely
  - Chunky calls as opposed to chatty

# References

- MS Argument Passing and Naming Conventions  
<http://msdn.microsoft.com/en-us/library/984x0h58.aspx>
- C Function Call Conventions and the Stack  
<http://www.csee.umbc.edu/~chang/cs313.s02/stack.shtml>
- Wikipedia  
[http://en.wikipedia.org/wiki/X86\\_calling\\_conventions](http://en.wikipedia.org/wiki/X86_calling_conventions)

# Links

- MSDN articles:
  - PInvoke overview  
<http://msdn.microsoft.com/msdnmag/issues/03/07/NET/>
  - Fixed Size Buffers (C# Programming Guide)  
[http://msdn2.microsoft.com/en-us/library/zycewsya\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/zycewsya(VS.80).aspx)
- Overview of the WIN32 API  
<http://www.pinvoke.net/>
- The P/Invoke Wizard:  
<http://www.pauliao.com/resources/tools/pinvoke.asp>



C# 3.0 in a Nutshell, 3rd Edition

[http://proquest.safaribooksonline.com/9780596527570/integrating\\_with\\_native\\_dlls](http://proquest.safaribooksonline.com/9780596527570/integrating_with_native_dlls)