Guillermo "Guille" Som

Interoperabilidad con código no administrado

En este artículo nos centraremos en la interoperabilidad con código no administrado (*unmanaged code*), que es la que nos permitirá usar funciones declaradas en librerías escritas con lenguajes que no pertenecen a la plataforma .NET, (por ejemplo funciones del API de Windows), desde un lenguaje que usa el código administrado de .NET Framework.

» Interoperabilidad con código no administrado mediante invocación de plataforma (*PInvoke*)

Como comentábamos en el número anterior de **dotNetManía**, existen dos formas de utilizar código no administrado desde los lenguajes de .NET, es decir, usar código que no está escrito con un lenguaje que se ciñe a las normas impuestas por .NET Framework, una de ellas es la interoperabilidad COM, de la que nos ocuparemos próximamente en otro artículo, y la otra forma es mediante la invocación de plataforma, *PInvoke* o *Platform Invoke*, esta última es la que veremos a lo largo de este artículo

Declaración de las funciones no administradas para usar desde .NET

Para poder acceder a las funciones no administradas desde el código de nuestro lenguaje de .NET debemos usar el atributo `DllImport`, este atributo está definido en el espacio de nombres `System.Runtime.InteropServices`, por tanto, o bien debemos realizar una importación de ese espacio de nombres o bien debemos indicarlo delante de este atributo.

La forma más básica de usar este atributo es indicando la librería (DLL) en la que se encuentra la función a la que queremos acceder y el nombre de la función en cuestión, por ejemplo, para acceder a la función `SendMessage` del API de Windows, lo podemos hacer de esta forma:

```
[DllImport("user32.dll", EntryPoint = "SendMessage")]
private extern static void
            SendMessage( IntPtr hWnd, int wMsg,
                         int wParam, int lParam);
```

Como vemos el nombre de la función que queremos usar la indicamos mediante la propiedad `EntryPoint` del atributo `DllImport`, a continuación declaramos la función como `extern` y `static`, esto siempre debe ser así, `extern` lo usamos para indicar al compilador de C# que es una función que está declarada externamente a nuestro código; por otro lado, el uso de `static` es porque todas las decla-

Aunque sea casi una constante en cada uno de los artículos de esta sección, vuelvo a aclarar que el código mostrado en el artículo es en C#, pero que el de VB se puede conseguir en el ZIP de código que acompaña a estos artículos, aunque en este caso la aclaración es más necesaria que en otras ocasiones, ¿por qué? porque la forma que tiene VB de utilizar las funciones de código no administrado almacenadas en librerías del API de Windows es totalmente diferente a como lo tiene que hacer C#, diferente y más "simple", pero como Visual Basic también permite que utilicemos la forma en que C# debe hacerlo, no será complicado convertir el código mostrado. En el caso de las funciones del API que necesitan trabajar con punteros a funciones (métodos) de .NET, siempre tendremos que usar delegados, aunque desde VB la indicación de que usamos ese delegado es diferente a C#, (hay que usar `AddressOf`), será fácilmente comprensible.

Guillermo "Guille" Som es Microsoft MVP de Visual Basic desde 1997. Es redactor de dotNetManía, miembro de *Ineta Speakers Bureau Latin America*, mentor de *Solid Quality Learning* Iberoamérica y autor del libro *Manual Imprescindible de Visual Basic .NET*.
http://www.elguille.info

raciones de funciones no administradas deben estar declaradas como compartidas, por tanto no serán funciones de instancia, sino que pertenecen a la propia clase que las declara.

> **NOTA**
>
> El uso del atributo `EntryPint` es totalmente opcional, al menos si en la definición de la función usamos el mismo nombre que está definido como punto de entrada de esa función del API

La declaración de la función en realidad es lo que se conoce como un "prototipo" de función, es decir, no contiene código ejecutable, ya que ese código está en la librería que contiene la función. Esa declaración (o definición) debe incluir los parámetros adecuados que la función original ha definido. Por tanto, la mejor práctica para usar este tipo de funciones "no manejadas" es buscar la definición en la documentación del SDK de la plataforma (*platform SDK*) el cual se incluye con la documentación de Visual Studio, en dicha documentación veremos la definición en C de la función y los tipos de parámetros que define, por ejemplo, la función `SendMessage` está definida de esta forma:

```
LRESULT SendMessage(
  HWND hWnd,      // handle of destination window
  UINT Msg,       // message to send
  WPARAM wParam, // first message parameter
  LPARAM lParam  // second message parameter
);
```

Como podemos apreciar en la definición original se usan los tipos `UINT`, `WPARAM` y `LPARAM` que hemos "traducido" como `int`, incluso el tipo `HWND` también lo podríamos haber definido como `int`, esto en realidad no es importante, ya que lo que debemos tener en cuenta es que esos tipos en realidad son valores de 32 bits, por tanto cualquier tipo de 32 bits sería válido.

También podemos apreciar que la definición de esta función devuelve un valor del tipo `LRESULT`, este valor devuelto, según qué mensajes nos puede interesar, en otros no tanto, si no queremos usar el valor devuelto por una función del API la podemos definir como hemos visto en nuestro ejemplo, usando `void`, de esta forma, no tendremos en cuenta el valor devuelto, pero en los casos que nos interese saber el valor devuelto, podemos usar cualquier tipo de 32 bits, por ejemplo, `SendMessage` puede devolver un valor verdadero si todo fue bien, en ese caso podemos definirla como de tipo `int` o `bool`.

En la tabla 1 podemos ver algunos de los tipos definidos en el API de Windows y los equivalentes en .NET.

Tipo del API	Descripción	Equivalencia en .NET
`HWND`	*Handle* de la ventana de destino	`Int32` o `System.IntPtr`
`WORD`	Entero de 16 bits sin signo	`UInt16` o `Int16`
`DWORD`	Entero de 32 bit sin signo	`UInt32` o `Int32`
`BOOL`	Entero, (0 ó 1)	`Boolean` o `Int32`
`UINT`	Entero sin signo	`Int32` o `UInt32`
`WPARAM`	Cuando se usa en *Send/PeekMessage*, el primer parámetro del mensaje	`Int32`
`LPARAM`	El segundo parámetro del mensaje	`Int32`
`LPCTSTR`	Dirección de una cadena (no modificable)	`String` o `StringBuilder`
`LPTSTR`	Dirección de una cadena (modificable)	`String` o `StringBuilder`
`LPMSG`	Puntero a una estructura del tipo `MSG`	`Estructura`
`CALLBACK`	Definición de la función `CallBack`	`Delegate`

Tabla 1. Equivalencias de tipos del API con .NET

Algunos de los tipos del API mostrados en esta tabla, permiten que utilicemos indistintamente un tipo de datos u otro, por ejemplo, `UINT` en realidad es un tipo `unsigned int` de C, pero salvo que queramos pasar un valor entero positivo mayor del permitido por `Int32` (`Int32.Maxvalue`), podemos usar cualquiera de los dos tipos, ya que en realidad, a la función de C lo que le importa es que sea un tipo de 32 bits, al menos en las plataformas de 32 bits, ya que en las plataformas de 64 bits, esperará un entero de 64 bits, porque el tipo `int` de C depende de la plataforma, (y/o del compilador), a diferencia del tipo `long` que siempre será un entero de 32 bits, trabajemos en la plataforma que trabajemos.

> **NOTA**
>
> En .NET los tipos de datos siempre son de tamaño fijo independientemente de la plataforma en que se usen, a excepción de `IntPtr` y `UIntPtr` que dependen de la plataforma, por tanto en plataformas de 32 bits serán de 32 bits y en plataformas de 64 serán de 64 bits.

Funciones que envían cadenas al API

La función `SendMessage` es precisamente una de las funciones más versátiles del API de Windows, precisamente como se usa para enviar mensajes a las ventanas, estos pueden ser de tipos muy distintos, y precisamente por esa razón, los parámetros también variarán con frecuencia. Por ejemplo, si queremos asignar un texto a una ventana usaremos esa función indicando como mensaje `WM_SETTEXT`, la definición del último parámetro sería de la siguiente forma:

```
// address of window-text string
(LPARAM)(LPCTSTR)lpsz;
```

El último parámetro de la función es el que contiene el texto que queremos asignar a la ventana cuyo *handle* indicamos en el primer parámetro, por tanto debemos modificar la definición de esa función para que pueda "enviar" el texto. En este caso podemos usar indistintamente un tipo `string` o `StringBuilder`, pero si lo que la función hará es obtener un texto, deberíamos usar `StringBuilder`, en cualquiera de los dos casos debemos rellenarlos con los caracteres suficientes, los que esperamos recibir, por regla general 255 es suficiente.

La definición de esta versión de `SendMessage` quedaría de esta forma:

```
[DllImport("user32.dll")]
private extern static bool
       SendMessage( IntPtr hWnd, int wMsg,
                    int wParam, string lParam);
```

Funciones que reciben cadenas

Veamos ahora una función del API que devuelve una cadena en uno de los parámetros, es la función `GetClassName` y en la documentación del *Platform SDK* está definida como:

```
int GetClassName(
  HWND hWnd,          // handle of window
  LPTSTR lpClassName, // address of buffer for class name
  int nMaxCount       // size of buffer, in characters
);
```

En el segundo parámetro indicamos la variable que recibirá el nombre de la clase, y en el último debemos indicar el tamaño de ese *buffer*, el cual debe ser suficientemente grande para recibir todo el texto, pero como dijimos, con 255 será más que suficiente.

Una de las definiciones de esta función puede ser la siguiente:

```
[DllImport("user32.dll")]
private extern static int
             GetClassName(IntPtr hWnd,
                          StringBuilder lpClassName,
                          int nMaxCount);
```

Y la podemos usar de esta forma:

```
StringBuilder sb = new
             StringBuilder(new string(' ', 255));
int ret = GetClassName(winHandle, sb, sb.Length);
this.txtClassName.Text = sb.ToString().Substring(0, ret);
```

Es importante que nos fijemos en la forma en que está definido el objeto `StringBuilder` que usaremos como argumento receptor del nombre de la clase: Le pasamos al constructor una cadena con 255 caracteres (da igual que sean espacios). Hago esta aclaración porque no sería lo mismo definirlo usando el constructor que le indica la capacidad que tendrá el objeto `StringBuilder`, ya que esa capacidad no asigna ningún tipo de cadena. La función `GetClassName` devuelve el número de caracteres que ha asignado a la cadena, por eso usamos ese valor para recuperar la cadena recibida, ya que el `StringBuilder` seguirá teniendo 255 caracteres, pero sólo nos interesan los "ret" primeros.

En el caso de esta función, si el número de caracteres que pasamos como *buffer* es inferior al valor "real" del nombre de la clase, no ocurrirá nada "malo", simplemente recibiremos los caracteres que hemos indicado, por ejemplo el bloc de notas tiene un nombre de clase llamado *Notepad*, si en el constructor del `StringBuilder` hubiésemos indicado 5 en lugar de 255, la cadena devuelta sería "Note", sí, sólo 4 caracteres, esto es así porque todas las cadenas de `C` acaban con un valor "\0"que es el que indica el final de la cadena, y como sólo hemos indicado que queremos cinco, recibimos esos cinco, aunque el último no es imprimible.

Obtener el handle de una venana

Como acabamos de comprobar en los ejemplos que estamos usando siempre necesitamos el *handle* (o manejador que dicen algunos) de la ventana que queremos manipular. Esas ventanas no tienen porqué ser "ventanas" en toda regla, sino que también pueden ser controles, ya que para Windows todo son ventanas y todas se pueden manipular si sabemos el *handle*.

Si queremos usar controles o formularios de .NET con funciones de Windows, podemos usar la propiedad `Handle` definida en la clase `Control`, y que por tanto todos los controles de .NET tendrán, incluso los formularios. Precisamente el tipo de esa propiedad es `IntPtr`, por eso estamos usando ese tipo de datos para los parámetros que reciben ese tipo de información.

Pero si en lugar de una ventana "administrada" queremos usar una de la que no tenemos control directo, por ejemplo una aplicación que esté en ejecución, podemos usar la función `FindWindow`. Esta función devuelve el *handle* de una ventana. Para saber que ventana queremos "capturar", podemos usar o bien el nombre de la clase o bien el título. Lo más habitual es usando el título, por ejemplo, para saber el *handle* del bloc de notas, podemos hacerlo de esta forma:

```
IntPtr winHandle =
       FindWindow(null, "Sin título - Bloc de notas");
```

El texto indicado debe coincidir exactamente con el del título de la ventana, aunque no se hace distinción entre mayúsculas y minúsculas, salvo en las vocales acentuadas.

La definición de esta función sería:

```
[DllImport("user32.dll")]
private extern static IntPtr
              FindWindow( string lpClassName,
                          string lpWindowName);
```

Usar estructuras con el API de Windows

Otra forma de usar las funciones del API de Windows es mediante estructuras; en estos casos, debemos pasar esas estructuras por referencia, ya que las estructuras pasadas a las funciones del API se pasan como punteros. Por ejemplo la estructura **LVFINDINFO** se utiliza junto con la función **SendMessage** para buscar elementos de un control **ListView**, la definición de esta estructura es la siguiente:

```
typedef struct tagLVFINDINFO {
    UINT flags;
    LPCTSTR psz;
    LPARAM lParam;
    POINT pt;
    UINT vkDirection;
} LVFINDINFO, *LPFINDINFO;
```

En C# y de forma simplificada la podemos definir de esta forma:

```
private struct tagLVFINDINFO
{
    public uint flags;
    public string psz;
    public int lParam;
    public int pt;
    public uint vkDirection;
}
```

El campo **pt** en realidad debería ser de tipo **POINT** (o el equivalente en C#), pero como no lo usaremos y la función espera un puntero, podemos usar un tipo entero. De la misma forma, los campos definidos como **uint** pueden ser de tipo entero, o en el caso del campo **flags**, incluso podríamos definir una enumeración y usarla como tipo de datos, con idea de que se usen los valores permitidos en ese campo de la estructura. Esa enumeración podía ser la siguiente:

```
[FlagsAttribute]
enum FindItemFlags
{
  FullString = 0x02,    // Busca la cadena completa
  StartsWith = 0x08,    // Busca la cadena indicada
                        // desde el principio
  ContSearching = 0x20  // Sigue buscando desde el
                        // principio si no halla nada
}
```

Si nos decidimos a usar esta enumeración, el primer campo de las estructura deberíamos declararlo así:

```
FindItemFlags flags;
```

Y la definición apropiada de **SendMessage** para usar este tipo de datos, que como hemos dicho habría que usarlo por referencia, quedaría así:

```
[DllImport("user32.dll", EntryPoint = "SendMessage")]
private static extern int
   SendMessage( IntPtr hWnd, int msg,
                int iStart, ref tagLVFINDINFO plvfi);
```

Y la podríamos usar de esta manera:

```
const int LVM_FIRST = 0x1000;
const int LVM_FINDITEM = (LVM_FIRST + 13);

tagLVFINDINFO LV_FINDINFO = new tagLVFINDINFO();
LV_FINDINFO.flags = FindItemFlags.StartsWith;
LV_FINDINFO.psz = this.txtBuscarLV.Text;
int index = SendMessage(listView1.Handle,
            LVM_FINDITEM, -1, ref LV_FINDINFO);
if(index > -1)
{
    listView1.Items[index].Selected = true;
}
```

Funciones con devolución de llamada (callback)

Este es otro caso que también es común en algunas funciones del API: Usar funciones *CallBack* (o *de devolución de llamadas* que es como lo traducen en la documentación de Visual Studio). Este tipo de funciones, en realidad se definen como punteros a funciones y, como sabemos, en .NET Framework no existen los punteros, al menos como los de C, la única forma de usar punteros "administrados" es por medio de los delegados, por tanto, cuando nos encontremos con este tipo de funciones, tendremos que usar delegados. Un ejemplo "clásico" es la función **EnumWindows**, esta función está definida de la siguiente forma:

```
BOOL EnumWindows(
    WNDENUMPROC lpEnumFunc,
                  // pointer to callback function
    LPARAM lParam
                  // application-defined value
);
```

Por otro lado, el puntero a la función **callback** está definido de esta forma:

```
BOOL CALLBACK EnumWindowsProc(
    HWND hwnd,      // handle to parent window
    LPARAM lParam   // application-defined value
);
```

En C#, el puntero a la función lo definimos como un delegado, en este caso será una función que devuelva un valor **bool** y reciba dos parámetros, el primero para el *handle* de la ventana y el segundo de tipo entero:

```
private delegate bool
      EnumWindowsDelegate(IntPtr hWnd, int parametro);
```

La declaración de la función `EnumWindows` usará ese delegado como primer parámetro, quedando definida de esta forma:

```
[DllImport("user32.dll")]
private extern static bool
EnumWindows(EnumWindowsDelegate lpfn, int lParam);
```

Esta función lo que hará es "enumerar" todas las ventanas que actualmente estén funcionando, no tienen porqué ser ventanas "físicas", es decir, aplicaciones, sino cualquier ventana que esté actualmente en el sistema, más de las que nos pensamos, como podrás comprobar cuando uses el programa de ejemplo, la forma de enumeración consiste en llamar a la función `callback` pasándole el *handle* de cada una de las ventanas que haya, la función será la que se encargue de manejar esa información, que en nuestro caso las iremos asignando a un *ListView* en el que mostraremos el *handle* y el título.

EnumWindows seguirá llamando a nuestra función mientras haya ventanas que enumerar y reciba un valor verdadero desde nuestra función. Esto último es útil si lo que pretendemos es, por ejemplo, buscar una ventana determinada para saber si está en funcionamiento, y en el caso de que encontremos lo que buscamos, devolvemos `false` para dejar de recorrer el resto de ventanas.

Nuestra función `callback` podría ser como sigue:

```
private bool EnumWindowsProc(IntPtr hWnd,
                             int parametro)
{
  StringBuilder titulo = new StringBuilder(
                               new string(' ', 256));
  int ret;
  string nombreVentana;
  //
  ret = GetWindowText(hWnd, titulo, titulo.Length);
  if(ret == 0)
      return true;
  //
  nombreVentana = titulo.ToString().Substring(0, ret);
  if(nombreVentana != null && nombreVentana.Length >0)
  {
      ListViewItem liv =
                listView1.Items.Add(nombreVentana);
      liv.SubItems.Add(hWnd.ToString());
  }
  // Devolver true para continuar con la enumeración
  return true;
}
```

Para poner en marcha la enumeración de ventanas, lo haremos creando un nuevo delegado que pasaremos como valor del primer parámetro del `EnumWindows`:

```
EnumWindows(new EnumWindowsDelegate(this.EnumWindowsProc), 0);
```

Como sabemos, los delegados simplemente definen prototipos de funciones en los que se especifican los parámetros que dicha función deben definir, esto es lo que hace que los delegados sean más confiables que los punteros de C, ya que si .NET detecta que la función que queremos usar, (en el ejemplo anterior la indicada en el constructor del delegado), no coincide con ese prototipo, no nos dejará compilar la aplicación, con lo cual sabemos que la función no se ciñe a los parámetros que debe recibir.

La definición de la función `GetWindowText` que utilizamos para obtener el texto de la ventana indicada por el *handle* de la ventana la podemos definir de la siguiente forma:

```
[DllImport("user32.dll")]
private extern static int GetWindowText(
                          IntPtr hWnd,
                          StringBuilder lpString, int cch);
```

En la que volvemos a hacer uso de un tipo `StringBuilder`, ya que en el segundo parámetro recibiremos el título de esa ventana.

La clase Marshal y el atributo MarshalAs

(O cómo asegurarnos de pasar los tipos de datos correctos a las funciones del API)

Otra forma que podemos usar para definir los tipos usados en las declaraciones de las funciones del API es usando los tipos definidos en la enumeración `UnmanagedType`, esta enumeración define tipos como `LPStr`, y podemos usarlo de la siguiente forma:

```
[DllImport("user32.dll")]
private extern static int GetClassName(
  IntPtr hWnd,
  [MarshalAs(UnmanagedType.LPStr)] StringBuilder lpClassName,
  int nMaxCount);
```

El atributo `MarshalAs` es útil particularmente cuando estamos trabajando con arrays. Por ejemplo, si en una función del API se espera una estructura en el que uno de los campos es un array de bytes de un tamaño específico, podemos usar ese atributo indicando el tipo de datos que esperamos y el tamaño del mismo, por ejemplo la estructura `OSVERSIONINFO` tiene un campo que es un array de tipo `TCHAR` (`Char` de *Unicode*), veamos la definición de esa estructura en el API de Windows:

```
typedef struct _OSVERSIONINFO{
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR szCSDVersion[ 128 ];
} OSVERSIONINFO;
```

En C# la definiremos de la siguiente forma:

```
struct OSVERSIONINFO
{
    public int dwOSVersionInfoSize;
    public int dwMajorVersion;
    public int dwMinorVersion;
    public int dwBuildNumber;
    public int dwPlatformId;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst=128)]
    public string szCSDVersion;
}
```

La función con la que podemos usar esta estructura es **GetVersionEx**, a la que pasaremos la estructura por referencia y la declaración la podemos hacer de la siguiente forma:

```
[DllImport("kernel32.dll")]
private static extern int GetVersionEx(ref OSVERSIONINFO osinfo);
```

Y si queremos que quede constancia de que estamos usando una estructura no administrada, podemos aplicarle el atributo **MarshalAs** de esta forma:

```
[DllImport("kernel32.dll")]
private static extern int etVersionEx([MarshalAs(UnmanagedType.Struct)]
                                      ref OSVERSIONINFO osinfo);
```

Como vemos en la definición de la estructura, el primer campo espera el tamaño de dicha estructura, esto también es muy habitual en las estructuras usadas en el API de Windows, ese valor lo podemos obtener también por medio de la clase **Marshal**, concretamente con el método **SizeOf** al que le pasaremos como argumento la estructura (o tipo) del que queremos averiguar el tamaño. Por ejemplo para usar esa función, lo haremos de la siguiente forma:

```
OSVERSIONINFO osv = new OSVERSIONINFO();
osv.dwOSVersionInfoSize = Marshal.SizeOf(osv);
int res = GetVersionEx(ref osv);
if(res != 0)
{
  this.Text = "OS v" + osv.dwMajorVersion + "." + osv.dwMinorVersion +
              "." + osv.dwBuildNumber + " " + osv.szCSDVersion;
}
```

En este caso concreto, el valor devuelto en el campo **szCSDVersion** es el *Service Pack* que tenemos aplicado en nuestro sistema operativo.

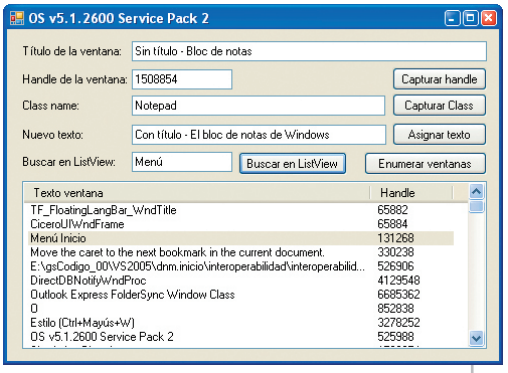

Figura 1. La aplicación de ejemplo en modo ejecución.

En la figura 1 podemos ver la aplicación de ejemplo para este artículo en plena ejecución. El código de esa aplicación (tanto para VB como para C#) lo puedes obtener desde la Web de **dotNetManía**, y está creado con Visual Studio 2005, (los proyectos de VB y C# se pueden usar indistintamente con la versión normal de Visual Studio 2005 o con las versiones Express de cada uno de los lenguajes), además, para aquellos que aún no han instalado la nueva versión de Visual Studio, en esos *zips* con el código, también se incluyen los proyectos de los dos lenguajes creados con la versión 2003 de Visual Studio, de esa forma nadie tendrá la excusa de no poder probar si todo lo dicho es cierto.

Conclusiones

El tema de la interoperabilidad con código no administrado mediante la invocación de plataforma (*PInvoke*) es mucho más amplio de lo que hemos tratado aquí, por tanto he tenido que resumirlo para mostrar lo que podemos usar en el día a día, ya que prácticamente hemos visto todas las posibilidades que normalmente se nos presentarán cuando queramos acceder a funciones definidas en librerías creadas con C/C++ o con cualquier otro compilador que permita la creación de este tipo de aplicaciones.

Por tanto, lo aquí explicado no solo es válido para las librerías incluidas en el sistema operativo, y para demostrarlo en el *zip* con el código, he incluido una librería (y su código fuente) creada para compilarla con el compilador gratuito de Borland C++ 5.5, además de una aplicación tanto para Visual Basic como para C# que usa dicha DLL.

En el próximo artículo veremos la tercera y última parte de esta serie sobre la interoperabilidad, en ese caso será sobre cómo crear componentes de .NET que podamos usar como componentes COM.