Guillermo "Guille" Som

Interoperabilidad

Yo interopero, tu interoperas,... ¡interoperemos todos!

La interoperabilidad bien entendida hace referencia a cómo utilizar clases de código no administrado desde aplicaciones de .NET y viceversa. Cuando hablamos de código no administrado, nos estamos refiriendo a código que no está generado con los compiladores de la plataforma .NET. El concepto de interoperabilidad lo ampliaremos un poco para aplicarlo también a los ensamblados creados con .NET, de forma que podamos usar las librerías creadas con un lenguaje de .NET desde otro lenguaje también de la familia .NET, típicamente con C# y Visual Basic.

» Interoperabilidad con código administrado y no administrado

Como acabamos de comentar, cuando hablamos de interoperabilidad lo primero que se nos viene a la cabeza son conceptos referidos a cómo "interoperar" con componentes COM (*Component Object Model*) o con funciones escritas con código no administrado, como pueden ser las funciones del API de Windows. De hecho, si buscamos "interoperabilidad" en el índice de la documentación de Visual Studio, lo primero que aparece es la interoperabilidad COM, y debemos "navegar" un poco para toparnos con las pocas entradas que tratan el tema de la interoperabilidad entre lenguajes de .NET.

En este artículo nos ocuparemos de estas tres formas de "interoperar" o comunicarnos con otros lenguajes, ya sean de la propia plataforma .NET (usando código administrado) o bien de otras plataformas, principalmente en la "comunicación" con lenguajes o código generado por lenguajes que utilizan código no administrado, es decir, el código que no está pensado para usar las convenciones del CLR o *runtime* de .NET, y que por tanto se ejecuta sin el amparo de .NET Framework.

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

Interoperabilidad con código administrado

Cuando hablamos de código administrado estamos refiriéndonos al código creado con lenguajes de .NET, por ejemplo C#, y que lo utilizamos desde otro lenguaje, también de .NET, por ejemplo Visual Basic. Esta comunicación entre lenguajes de la plataforma .NET es posible, y fácil de realizar, básicamente porque todos los lenguajes de .NET utilizan las clases definidas por la propia plataforma, y lo más importante, siguen unas normas previamente definidas, o al menos deberían seguirlas, para que esa comunicación sea totalmente efectiva.

> Lo que debería hacer un lenguaje de .NET para que el código generado pueda ser utilizado desde otros lenguajes de .NET es garantizar la compatibilidad con el CLS (*Common Language Specifications o especificaciones comunes de .NET*)

El condicional que hemos utilizado en la frase anterior es algo que debemos tener en cuenta para que los ensamblados de dos lenguajes de .NET se puedan comunicar, o lo que es lo mismo, podamos

usar en nuestros proyectos de .NET los ensamblados generados desde otro lenguaje también basado en .NET.

Lo que debería hacer un lenguaje de .NET para que el código generado pueda ser utilizado desde otros lenguajes de .NET es garantizar la compatibilidad con el CLS (*Common Language Specifications* o especificaciones comunes de .NET). Estas especificaciones marcan una serie de normas o condiciones para que esa comunicación entre lenguajes diferentes sea factible. Por ejemplo, el código generado con Visual Basic, (al menos hasta la versión 7.0 que es la incluida en las versiones anteriores a Visual Studio 2005), siempre es compatible con esas especificaciones, mientras que el código generado por C#, e incluso por Visual Basic 2005, no tiene porqué ser siempre totalmente compatible con el CLS, por tanto si queremos generar código que sea 100% compatible con el resto de lenguajes de .NET debemos asegurarnos de que dicho código es **CLS-compatible**. La mejor forma de asegurarnos esa compatibilidad es utilizar el atributo `CLSCompliant` indicándole un valor verdadero.

Las aplicaciones de Visual Basic (anteriores a la versión 8.0 o VB2005), siempre incluyen ese atributo en el fichero `AssemblyInfo.vb`, algo que no ocurre en los proyectos de C# (o de VB2005), por tanto, si queremos asegurarnos esa compatibilidad, tendremos que agregar la siguiente línea de código: `[assembly: System.CLSCompliant(true)]`, en el fichero `AssemblyInfo.cs` o cualquiera de los ficheros de código incluidos en el proyecto.

> **NOTA**
>
> En Visual Basic, los atributos se indican encerrándolos entre signos mayor y menor, en el caso de `CLSCompliant` sería: `<Assembly: CLSCompliant(True)>`. Debido a que en los artículos de *dnm.inicio* siempre mostramos el código fuente en C#, cuando comentemos algunos de los atributos que tendremos que usar, estos se harán para ese lenguaje, pero en los ficheros `.zip` con el código de ejemplo, encontraremos tanto el código de C# como de Visual Basic.

La compatibilidad con CLS nos asegura que el código generado con nuestro compilador favorito podrá ser utilizado con cualquier otro lenguaje que cree ensamblados compatibles con dichas especificaciones, y el atributo `CLSCompliant` nos servirá para que el compilador nos avise cuando usemos características no compatibles, como es la utilización de tipos sin signo en los parámetros o valores devueltos por los métodos o propiedades, y con la llegada de la versión 2.0 de .NET Framework, dicha falta de compatibilidad se ve ampliada, por ejemplo, con los tipos *generic*.

Aunque si trabajamos con lenguajes que aceptan esos tipos de datos "no compatibles", podremos interoperar entre ellos, por ejemplo, Visual Basic 2005 al igual que C# 2.0 acepta tipos enteros sin signo o tipos *generic*, por tanto podremos crear ensamblados con uno de esos lenguajes que utilice esas características no contempladas en CLS y usarlas desde el otro lenguaje sin mayores problemas. Esto último lo podremos comprobar en el código de ejemplo para Visual Studio 2005 que incluimos en los ZIP y que podemos bajar desde el sitio Web de la revista.

Interoperabilidad con código no administrado

Las otras dos formas de interoperabilidad son mediante el uso de COM (interoperabilidad COM) y lo que se conoce como *Platform Invoke* (`PInvoke`) o invocación de plataforma.

La primera es mediante el uso de componentes COM, también conocidos como componentes de automatización OLE o simplemente ActiveX. Este tipo de componentes los podemos generar con lenguajes capaces de crear controles o librerías ActiveX, como Visual Basic 6.0. Esta forma de comunicación la podemos realizar en dos direcciones, una para poder acceder desde .NET a componentes COM, y la otra para acceder a componentes creados en .NET desde lenguajes capaces de utilizar COM. Como veremos, esta sería una solución bastante razonable para permitir el uso de librerías creadas en cualquier lenguaje de .NET desde aplicaciones de VB6, evitándonos la tan temida migración de aplicaciones de VB6 al mundo de .NET Framework, aunque siempre tendremos que hacerlo siguiendo unas reglas; reglas que explicaremos en este y otros artículos.

La invocación de plataforma (`PInvoke`) es la posibilidad de utilizar funciones declaradas en librerías de la plataforma (sistema operativo) en la que se ejecuta nuestra aplicación de .NET, por ejemplo para acceder a cierta funcionalidad no proporcionada por la librería de clases de .NET. Actualmente las plataformas a las que tenemos acceso son Windows (Win32) y los sistemas basados en Windows CE (Pocket PC o Smartphone), aunque en este artículo, y en los próximos sobre interoperabilidad de la sección *dnm.inicio.taller*, nos basaremos en el sistema operativo Windows.

Interoperabilidad entre lenguajes de .NET

(O de cómo usar ensamblados de .NET desde otros lenguajes de .NET)

Esta sería la forma más fácil de permitir que otros programadores utilicen nuestro código, pero sin necesidad de que tengan acceso al código fuente, sino al resultado ya compilado y listo para usar.

La ventaja de que otros utilicen nuestro código compilado es doble, una es que así no tendrán acceso directo al código fuente, de esta forma nos aseguramos no solo de que no tendrán detalles de cómo lo hemos implementado, sino de que no lo modifiquen directamente, ya que podían echar al traste todo nuestro trabajo, por poner solo unos casos. La otra ventaja es que podrán usarlo desde otros lenguajes distintos al que nosotros hemos utilizado, (o incluso el mismo lenguaje, ya que una vez compilado da igual el lenguaje usado), consiguiendo de esta forma que cada cual utilice el lenguaje con el que más a gusto se sienta programando. Por supuesto, también está la situación inversa: que nosotros utilicemos en nuestros proyectos lo que otros han desarrollado utilizando cualquier lenguaje de la plataforma .NET.

En cualquier caso, ese código deberá estar compilado y generado como una librería (ensamblado DLL), o en un módulo (`.netmodule`), aunque estos últimos serán menos habituales, principalmente porque actualmente no están soportados para usarlos desde el IDE de Visual Studio, sino para la compilación desde la línea de comandos, (de los módulos puede que nos ocupemos en otra ocasión), por tanto en adelante utilizaremos solo ficheros con extensión `.dll`, que son los que podemos referenciar en nuestros proyectos.

Porque de eso se trata, de crear una referencia en nuestro proyecto que nos permita acceder a los tipos definidos en el ensamblado, de esta forma, la librería que queremos usar se comportará de la misma forma que lo hacen las librerías del propio .NET que habitualmente utilizamos en nuestros proyectos.

Crear una librería de .NET

Si queremos crear una libraría para usarla como componente en otros proyectos de .NET, tendremos que crear un nuevo proyecto del tipo “Biblioteca de clases”, aunque en realidad podremos crear cualquier tipo de proyecto de los que tengamos disponibles, ya que no solo debemos restringir nuestra “interoperabilidad” a ese tipo de proyecto, porque de lo que se trata es que nosotros podamos crear componentes de .NET que puedan ser usados por otros, aunque básicamente, al menos para aplicaciones de escritorio o aplicaciones Windows, los tres tipos que podemos crear son:

- Aplicación para Windows.
- Biblioteca de clases.
- Biblioteca de controles para Windows.

Con los dos últimos no tendremos ningún tipo de problema, ya que el tipo de ensamblado creado es del tipo DLL, sin embargo, con el primero, en principio no podríamos utilizarlo, ya que no podemos crear referencias a ejecutables (`.exe`).

Si nuestra intención es utilizar formularios desde otro proyecto, la mejor forma de crearlos es mediante un proyecto del tipo “Aplicación para Windows”. Aunque a la hora de compilar ese proyecto tendremos que indicar que se genere un fichero del tipo DLL en lugar de EXE, para ello tendremos que indicar en las propiedades del proyecto que el tipo de resultado es “Biblioteca de clases” en lugar de “Aplicación de Windows” que será el valor predeterminado. Como sabemos, en la versión 2005 de Visual Studio, las propiedades del proyecto se presentan de forma diferente que en las versiones anteriores, pero en definitiva lo que importa es que se genere una DLL en lugar de un ejecutable, en la figura 1 podemos ver la ventana de propiedades de un proyecto de C# creado con Visual Studio 2003.

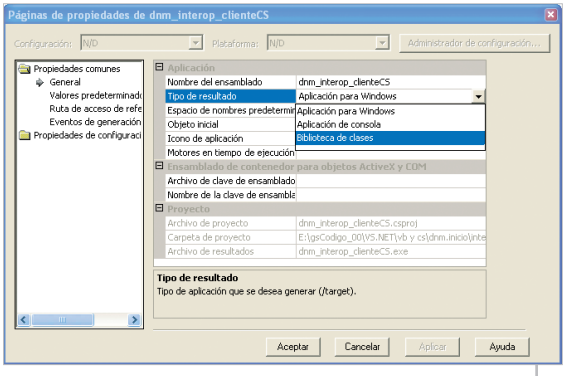

Figura 1. Desde las propiedades del proyecto podemos indicar el tipo de ensamblado que queremos generar.

Una vez generado el ensamblado DLL podremos referenciarlo desde cualquier otro proyecto, y podremos acceder a los formularios que contenga de la misma forma que lo haríamos con el resto de clases o tipos definidos en esa librería, ya que los formularios son clases al fin y al cabo.

Referenciar una librería de .NET

Para poder utilizar las librerías creadas con cualquier lenguaje de .NET desde nuestro proyecto,

tendremos que seguir los mismos pasos que para añadir referencias de cualquier otra librería: Por medio del elemento "Referencias" en el explorador de soluciones.

Si la librería ya está compilada tendremos que pulsar en el botón "Examinar" y localizar el ensamblado (que ya debe estar compilado como `.dll`), seleccionarlo y agregarlo. Si esa librería que queremos referenciar es un proyecto creado por nosotros, podemos añadirlo a la solución actual y agregar la referencia indicando el proyecto, desde la ficha "Proyectos", de esta forma tendremos oportunidad de depurar la librería junto a nuestra aplicación cliente.

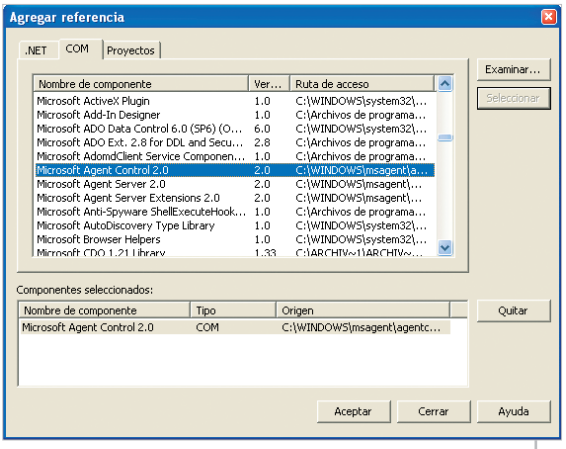

Figura 2. Añadir un componente COM a las referencias de nuestro proyecto.

> **NOTA**
>
> Las librerías (ensamblados DLL) creadas con los lenguajes basados en la versión 2.0 de .NET Framework (Visual Studio 2005) no las podemos usar desde proyectos de las versiones anteriores, aunque desde Visual Studio 2005 (o las versiones Express) sí podremos crear referencias a ensamblados creados con las versiones anteriores de .NET.

Interoperabilidad COM

Parte I: De cómo usar componentes COM desde aplicaciones .NET

Trabajar con componentes COM (o componentes ActiveX) desde Visual Studio .NET (o VS2005), es relativamente fácil y no requiere hacer nada especial por nuestra parte, ya que el proceso es el mismo que el indicado anteriormente: agregar una referencia en nuestro proyecto al componente COM (ya sea una DLL un control OCX e incluso un EXE), que queramos usar. De forma predeterminada, y tal como funciona COM, todos los controles (OCX) y las librerías (DLL) deben estar previamente registrados en el sistema, por tanto esos componentes registrados serán los que se muestren en la pestaña COM del cuadro de diálogo de agregar referencias, seleccionamos el que nos interese, y será el propio Visual Studio el que se encargue de los "pormenores" para que podamos usar ese componente en nuestro proyecto de Visual Studio.

Por ejemplo, si desde un proyecto de Visual Studio queremos utilizar el control `Agent`, para mostrar un "muñeco" al estilo de los asistentes de Office, tendremos que utilizar el control ActiveX: *Microsoft Agent Control*. Para ello haremos lo que acabamos de indicar, pulsamos en agregar referencia, seleccionamos la ficha "COM" y buscamos el componente que queramos, en este caso, *Microsoft Agent Control*, tal como vemos en la figura 2, pulsamos en el botón "Seleccionar" y aceptamos.

Si miramos las referencias, veremos una nueva entrada: *AgentObjects*. Si anteriormente no hemos usado ese control, tendremos que añadirlo al cuadro de herramientas, ya que no se añade de forma automática. Para añadirlo debemos pulsar con el botón secundario del ratón (normalmente el botón derecho), sobre el cuadro de herramientas, y del menú emergente, seleccionar "Agregar o quitar elementos", (sería recomendable añadir una nueva ficha o bien mostrarlas todas y si tenemos la ficha "Componentes Extras" añadirlo a esa ficha, con idea de no mezclar estos controles COM con los normales de Visual Studio), del cuadro de diálogo que nos muestra, seleccionaremos la ficha "COM" y buscaremos el control en cuestión, marcamos la casilla que hay junto al control y pulsamos "Aceptar".

Una vez añadido el control en el cuadro de herramientas, lo añadimos al formulario y veremos que en las referencias tenemos otra nueva: `AxAgentObjects`. Cuando compilemos el proyecto, comprobaremos que en el directorio bin (en el que se crea el ejecutable) tenemos dos nuevas librerías: `AxInterop.AgentObjects.dll` y `Interop.AgentObjects.dll`, estas librerías son generadas automáticamente por Visual Studio y contienen las definiciones de las clases del control, pero preparadas o convertidas para usar desde una aplicación .NET.

Todo este proceso que acabamos de describir es solo necesario si utilizamos controles generados

con aplicaciones COM, (estos controles pueden estar en librerías con extensiones `.ocx` o `.dll`, ya que en el fondo son lo mismo), pero si utilizamos componentes "normales", simplemente añadiremos la referencia a la librería y Visual Studio creará las librerías de interoperabilidad necesarias para acceder a los tipos de datos que contenga.

Pero debemos estar advertidos que no siempre esas librerías se comportarán de la misma forma que si las utilizáramos desde compiladores capaces de trabajar directamente con componentes COM, por tanto siempre deberíamos hacer las comprobaciones suficientes para asegurarnos un 100% de funcionalidad.

Parte II: De cómo usar componentes .NET desde aplicaciones COM

Esta es la parte más interesante de la interoperabilidad COM, el poder usar componentes creados con los lenguajes .NET desde lenguajes que no saben nada de .NET pero si pueden trabajar con componentes COM, como es el caso de Visual Basic 6.0.

Para lograr esta interoperabilidad, con la que podremos usar algunas de las características de .NET desde lenguajes de generaciones anteriores, solamente tendremos que indicarle en las propiedades del proyecto que queremos registrar el componente para interoperabilidad COM. Si trabajamos con Visual Studio 2003, esta opción la encontraremos en "Propiedades de generación>Generar" dentro de la ventana de propiedades del proyecto; si trabajamos con Visual Studio 2005, estará en la ficha de generación de la ventana de propiedades, en cualquier caso, solamente estará habilitada si el tipo de proyecto es "Biblioteca de clases".

Una vez hemos seleccionado dicha opción (o le hemos asignado un valor verdadero), al compilar el proyecto, veremos que además de la librería (`.dll`) tendremos un fichero con la extensión `.tlb` que es la librería de tipos compatibles con COM, y que estará registrada en nuestro equipo, pero en otros equipos tendremos que registrarla para poder utilizarla.

Una vez que tenemos la librería de tipos (`.tlb`), la podemos registrar en el equipo de destino utilizando un fichero con extensión .reg, (para registrar los tipos COM), el cual podemos generar con la utilidad `Regasm.exe` (*registro de ensamblados*) indicándole el parámetro `/regfile`. También podemos generar esa librería de tipos y registrarla al mismo tiempo usando la opción `/tlb` en esa misma utilidad.

Una vez registrado nuestro componente, podremos usarlo desde compiladores que sepan manejar los componentes COM. Pero tal como tenemos nuestro componente COM, la comprobación de los métodos y demás miembros se realizará en tiempo de ejecución, lo que se conoce como *late binding*. Si queremos acceder a esos miembros expuestos por nuestra clase, tendremos que indicarle al compilador de .NET que genere las interfaces necesarias para poder tener acceso en tiempo de compilación, para ello debemos aplicar a nuestra clase el atributo `ClassInterface` con el valor `AutoDual`, por tanto, aplicaremos el siguiente atributo a la clase (o clases) que queramos exportar:

```
[ClassInterface(ClassInterfaceType.AutoDual)]
```

Hay más detalles que deberíamos tener en cuenta, entre ellos la posibilidad de detectar eventos o habilitar la compatibilidad binaria, pero de estos detalles nos ocuparemos en la sección *dnm.inicio.taller* de otro número de **dotNetManía**.

> **NOTA**
>
> Los métodos o miembros estáticos (o compartidos) no se exportan a COM, por tanto, si tenemos algunos en nuestras clases, deberíamos definirlos como métodos o miembros de instancia. De igual forma, debemos definir siempre un constructor sin parámetros en todas las clases de las que queramos crear nuevas instancias desde un lenguaje COM, ya que es la única forma de crear nuevos objetos COM.

Interoperabilidad con invocación de plataforma (PInvoke)

(O de cómo utilizar funciones definidas en librerías del API de Windows)

Para finalizar con este artículo sobre los fundamentos de la interoperabilidad entre lenguajes de .NET y otros lenguajes que no trabajan con este marco de trabajo, veremos cómo podemos usar las funciones definidas en librerías creadas con lenguajes que no están amparados bajo las tecnologías expuestas por .NET Framework. A pesar de que la extensión de esas librerías también sea `.dll` no debemos confundirlas con las librerías generadas por medio de automatización (COM o ActiveX), ya que a las que nos estamos refiriendo son librerías "normales", que podremos generar con lenguajes como C/C++ o Delphi, pero no

con Visual Basic 6.0. El caso más habitual es el uso de las funciones del API del sistema operativo Windows.

Si estamos usando Visual Basic, las declaraciones para acceder a las funciones del API (o de librerías creadas con compiladores de código no administrado), las podemos hacer de dos formas diferentes, aunque en realidad una de ellas, que se mantiene por compatibilidad con VB6, lo que hace es generar el código adecuado de forma automática y totalmente transparente; pero como el compilador de C# no tiene ese automatismo, veremos cómo declarar esas funciones al estilo .NET, que es válido tanto para Visual Basic como C#, ya que se realiza por medio del atributo `DllImport`. Como siempre, en el código fuente de ejemplo, tendremos proyectos para los dos lenguajes.

La forma más simple de usar el atributo `DllImport` es aplicarlo a la declaración de una función que nos servirá de puente para acceder a la función del API, en ese atributo indicaremos como mínimo el nombre de la librería en la que se encuentra dicha función.

Por ejemplo, si tenemos esta declaración del API de Windows:

```
DWORD GetLongPathName(
      LPCTSTR lpszShortPath, LPTSTR
lpszLongPath, DWORD cchBuffer);
```

En C# la podríamos declarar de esta otra:

```
[System.Runtime.InteropServices.DllImport(
                              "kernel32.dll")]
private extern static int GetLongPathName(
      string lpszShortPath,
      System.Text.StringBuilder lpszLongPath,
      int cchBuffer);
```

En el atributo indicamos que esa función está en la librería `kernel32.dll` y la definición de la función la hacemos con `extern static`, (siempre debemos declararlas como compartidas y externas) con los parámetros adecuados a los tipos que .NET utiliza. Y como podemos comprobar, cuando el API espera una cadena, podemos declararla tanto usando el tipo `string` como `StringBuilder`, en los tipos cadena que reciben valores, como es el caso del segundo parámetro de esta función, es preferible hacerlo con un objeto del tipo `StringBuilder` que contenga los caracteres suficientes para recibir el valor, normalmente la cantidad de caracteres dependerá del valor devuelto, en el caso de ser un *path*, se recomienda tener como mínimo 255 caracteres.

Una vez declarada la función, la podremos usar como cualquier otra función de .NET, pero siempre siguiendo las reglas de uso del API, en este ejemplo particular, el segundo parámetro debe tener capacidad suficiente para recibir el nombre largo del fichero indicado en el primer parámetro.

NOTA

De las funciones del API que utilizan cadenas de caracteres, suele haber dos definiciones distintas, según utilicen caracteres ANSI o Unicode, la primera acabará con la letra A y la otra con W, pero el atributo `DllImport` utilizará la más adecuada según el sistema operativo que estemos usando.

Cuando usemos las funciones del API, debemos tener en cuenta que los tipos de datos no son totalmente compatibles entre los definidos en .NET y los utilizados por los otros compiladores, en la declaración original de la función `GetLongPathName`, el último parámetro y el valor devuelto por la función es de tipo `DWORD` que en realidad es un tipo `unsigned long` de C (entero de 32 bits sin signo) y equivale al tipo `UInt32` de .NET, pero como vemos, podemos utilizar un tipo de 32 bits con signo (`int`), por tanto para usar esas funciones desde las aplicaciones .NET tendremos que utilizar tipos de datos compatibles con los tipos de los compiladores de código no administrado. En la documentación de Visual Studio podemos encontrar una relación de las equivalencias de los tipos usados por el API de Windows y los de .NET, pero básicamente cuando en el API se utilice un tipo `long` en .NET usaremos un entero de 32 bits.

Conclusiones

En este artículo hemos intentado explicar las diferentes formas con las que podemos interoperar con el código de otros lenguajes, ya sean del propio .NET como los creados con compiladores que no generan código para esta plataforma, además de cómo utilizar los componentes COM desde .NET y viceversa. En próximos artículos de esta sección veremos ejemplos prácticos de los dos casos en los que debemos "interoperar" con lenguajes que no utilizan los compiladores de .NET.