

PROGRAMACION CON ADO.NET EN C#

PRIMERA PARTE - RESUMEN DE EJERCICIOS

- A0** Aplicaciones Windows Forms
- A1** Creación de tablas y columnas
- A2** Clases derivadas de conjuntos de datos
- A3** Definición de conjuntos de datos con XSD
- A4** Enlace de datos simple
- A5** Relaciones entre tablas
- A6** Columnas basadas en expresiones
- A7** Persistencia
- A8** Filtros y búsquedas
- A9** Vistas de datos
- A10** Enlace complejo a datos
- A11** Actualización y versiones
- A12** Eventos de actualización

*When one door closes, another opens;
but we often look so long and so regretfully upon the closed door
that we do not see the one which has opened for us.*

Alexander Graham Bell

HABÍA UNA VEZ UN MUNDO DENTRO de los ordenadores en el que la mano derecha no sabía lo qué hacía la mano izquierda. Dos aplicaciones, incluso si se ejecutaban dentro de una misma máquina, sólo podían comunicarse entre sí a través de complicados protocolos asíncronos. Estos sistemas de señales eran tan primitivos, que para cada conversación era necesario inventar todo un lenguaje. La aplicación A conversaba en hotentote con el programa B, pero para interpelar al servicio C, debía usar el kickapoo, dialecto continental. Era la Torre de Babel, pero sin torre, porque en un entorno tan hostil se hacía imposible edificar más allá de cierta altura.

Sabios y chamanes comprendían cuán desalentadora era la falta de comunicación, aunque encontraban dificultades a la hora de explicar el problema a sus respectivas tribus. Poco a poco fueron ideando sistemas de comunicación más potentes y a la vez más fáciles de usar. Surgieron nuevos protocolos con extraños nombres, como OLE, CORBA, COM, MTS y Java Remoting, y un proceso parecido a la selección natural comenzó a moldear a los más aptos, mientras que los menos afortunados eran devorados y digeridos por los supervivientes.

Una vista panorámica

Aparte de lo que sugieren las siglas mencionadas en el idealizado cuento de hadas que he empezado a relatar, ¿puede usted reconocer a la plataforma .NET como protagonista de dicha historia? Quizás sea una simplificación sostener que los orígenes de .NET tienen que ver en exclusiva con la necesidad de disponer de un modelo de componentes potente y fácil de usar, pero hay un núcleo de verdad en ello. En todo caso, la mayoría de los programadores nos hemos despertado un buen día en medio del ajetreo de una ciudad en construcción. Mi primer contacto serio con la plataforma, por poner el ejemplo más cercano, tuvo lugar a mediados de 2002, cuando Microsoft liberó la primera versión oficial de la misma. Es cierto que por entonces, utilizaba Delphi como herramienta principal de programación, pero en una situación muy parecida se encuentran muchas personas familiarizadas con Visual Studio 6. Y es porque entre esa versión y la siguiente se produjo una verdadera revolución en la programación para Windows.

El objetivo de este curso es describir un sector pequeño pero esencial de la flamante megápolis redmontiana: las técnicas de acceso a datos en la plataforma. Esto no sólo incluye las clases de acceso directo a SQL y las técnicas de caché y edición, sino también los conocimientos necesarios para el acceso remoto, la partición en capas físicas de sistemas grandes y el uso de servicios corporativos para garantizar la escalabilidad de nuestras aplicaciones.

Organización del contenido

El curso completo se ha dividido en tres partes. La primera parte está formada por tres series de ejercicios, mientras que hay una sola serie para cada una de las otras dos partes. Este es el contenido asignado a cada serie:

- ♦ **Serie A**
Conjuntos de datos y enlace a datos.
- ♦ **Serie B**
Acceso directo a servidores SQL: la capa conectada.
- ♦ **Serie C**
Adaptadores de datos.

♦ **Serie D**

Programación en dos capas: un ejercicio práctico.

♦ **Serie E**

Programación en tres capas: .NET Remoting y XML Web Services.

Las tres primeras series se han organizado siguiendo el siguiente hilo argumental:

- ADO.NET ofrece la clase *DataSet*, que funciona como una pequeña base de datos en memoria.
- Paralelamente, existe una interfaz muy sencilla para el acceso directo a servidores SQL.
- Es posible combinar las dos interfaces anteriores, con la ayuda de *adaptadores de datos*, para traer registros desde el servidor SQL a la memoria de las estaciones clientes. Esta información puede editarse localmente para luego reenviarla al servidor SQL con la ayuda de los mismos adaptadores de datos.

Las otras dos series tienen una estructura muy diferente. Toda la serie D consiste en el desarrollo de una misma aplicación: un programa de facturación muy simple, pero que muestra unas cuantas técnicas prácticas, como el uso de gestores de ventanas, de menús, el uso de reflexión para el descubrimiento de tipos y el sistema básico de impresión en .NET. Además de este ejercicio, se incluye el código fuente completo de una aplicación “paralela”, esta vez desarrollada con los componentes de Developer Express para Windows Forms. Por supuesto, los distintos sistemas de gestión desarrollados a lo largo de los ejercicios se adaptan para funcionar con estos componentes.

La serie E, en cambio, combina ejercicios de presentación de la teoría con la aplicación de lo aprendido al proyecto desarrollado en la serie anterior.

Requisitos para el seguimiento

Los ejercicios de este curso han sido desarrollados con Visual C# Standard 2003, con el objetivo de que los requisitos para el seguimiento fuesen los mínimos. Necesitará instalar la versión 1.1 de .NET, y Visual C# Standard 2003, o Visual Studio 2003, en cualquiera de sus variantes, siempre que se asegure de instalar el soporte para C#.

Como base de datos, he usado el *Microsoft Data Engine 2000* (MSDE): una versión reducida del motor de bases de datos utilizado por SQL Server 2000. Esto se debe a que Visual C# Standard sólo puede acceder a bases de datos ubicadas en una instancia local del MSDE. El mayor problema del MSDE 2000 es que carece de herramientas gráficas de administración. Si debe usar Visual C#, en vez de Visual Studio, una solución aceptable consiste en instalar primero el MSDE en la máquina de desarrollo, y luego instalar las herramientas clientes de la versión completa de SQL Server. Existe una edición de SQL Server 2000, la llamada SQL Server Developer, que se puede conseguir por un precio muy ajustado, en torno a los 50 euros. Si va a usar Visual Studio, puede usar la versión completa de SQL Server, sin importar si la instancia es local o remota.

Las tres primeras series del curso utilizan la base de datos de ejemplos *Northwind*, que viene con SQL Server. Las dos series restantes utilizan una base de datos propia incluida con los ejercicios.

He intentado incluir toda la teoría necesaria para resolver cada ejercicio. No obstante, es aconsejable tener a mano algún libro sobre ADO.NET... y sí, claro, le recomiendo que éste sea **La Cara Oculta de C#**.

Convenios

A pesar de que cada año nuestros ordenadores son más rápidos y baratos, la impresión de libros sigue siendo cara: es raro ver un libro técnico impreso en colores. Nos hemos acostumbrado a leer en blanco, negro y gris... y puede que ahora le extrañe el que haya usado colores sin complejo alguno a lo ancho y largo de este manual. Me siento justificado: no lo he hecho por el dudoso efecto estético, sino para mejorar la legibilidad dentro de lo posible.

Este pequeño fragmento de código muestra algunas de las convenciones que he utilizado:

```
public class Triangular : Chip, IOscilador, IConfigurable
{
    public double Voltaje(double tiempo)
    {
        // Implementación de la onda triangular ...
    }
}
```

Al igual que ocurre en el nuevo editor de código de Visual Studio 2005, he resaltado en cian los identificadores que corresponden a nombres de tipos, sin importar si se trata de una clase, una estructura, una interfaz, un tipo enumerativo o delegado. Es un indicio útil para lector porque no puede deducirse siempre de la sintaxis. Por el contrario, he seguido marcando las palabras reservadas mediante negritas, en vez de usar texto de color azul como en Visual Studio. En mi opinión, al resaltar de esta manera las palabras claves se facilita la identificación de la estructura sintáctica del texto. Por último, he usado itálicas en color verde para los comentarios. Podría haber usado el gris para comentarios, pero en las pruebas con impresoras monocromas he comprobado que el verde se traduce mejor, además de ser también el color usado por Visual Studio 2005.

Verá también, de cuando en cuando, números en color gris situados fuera de los márgenes, como el 999 número que acompaña a este párrafo. Se trata de referencias a **La Cara Oculta de C#**, y representan la página donde podrá encontrar más información sobre el tema que se esté explicando.

Agradecimientos

Quiero agradecer, sobre todo, la fe y la paciencia de todas las personas que adquirieron la serie D de este curso por adelantado. Aprovecho también para ofrecer mis más sinceras disculpas por los retrasos: todavía tengo mucho que aprender y hay mucho también por escribir. De este cuento de hadas tecnológico sólo hemos visto una parte insignificante, aunque ya se puede predecir que disfrutaremos de un final feliz.

Ian Marteens
www.marteens.com
Madrid, agosto de 2005

INTRODUCCIÓN A LA PLATAFORMA

ANTES DE PASAR A LOS ejercicios, resumiremos algunas de las características de la plataforma .NET que nos serán necesarias más adelante. No pretendo agotar el tema, como es lógico, y más que profundizar en detalles, me interesa presentar información práctica.

¿Qué es la plataforma .NET?

Una parábola budista cuenta la historia de tres ciegos que querían saber qué era un elefante. Acudieron al palacio real y suplicaron al monarca que les permitiese palpar su elefante blanco. El rey, budista devoto y compasivo, accedió, y un criado llevó a los ciegos al estanque donde se bañaba el paquidermo. A una voz del sirviente, los ciegos se abalanzaron sobre el animal, agarrándolo por distintos puntos de su inmensa anatomía:

- ¡Por la gloria de mi madre! – exclamó el primero – ¡Un elefante es como una columna, pequeña pero muy ancha! – dijo, pues se había aferrado a una de las patas.
- ¡Mientes, bellaco! – se enfadó el segundo – ¡Es una serpiente peluda y se retuerce como un demonio! – explicó mientras intentaba que el elefante no se lo sacudiese de la trompa.
- ¡Hmffff! – se oyó la voz apagada del tercero – ¡que alguien me saque de esta caverna maloliente...!

Estoy convencido de haber contado esta historia en algún otro sitio, aunque creo que el final era diferente. De todos modos, es una historia apropiada, porque la plataforma .NET puede presentar tantas apariencias como el elefante del cuento, incluyendo la caverna maloliente.

Por ejemplo, para un forofo de la programación de sistemas, .NET es la versión 3.0 de COM. Y no le falta razón, al menos desde el punto de vista histórico. Al parecer, las primeras ideas sobre .NET surgieron como parte del diseño de un sucesor para COM+. ¿Qué es COM+? Una serie de protocolos que permiten la colaboración entre objetos desarrollados con diferentes lenguajes y herramientas, y que pueden residir en diferentes procesos o incluso máquinas. ¿Qué es .NET, si lo observamos a través de esta estrecha rendija? Ha adivinado: es lo mismo, pero mejor. Es por eso una ironía que incluso la versión 2.0 de la plataforma no haya logrado sustituir completamente la funcionalidad de COM+, y que deba colaborar con COM+ para poder aprovechar los llamados *servicios corporativos*, que estudiaremos en la última serie de este curso.

Pero yo no soy forofo de la programación de sistemas; al menos, no de la tribu de esta especie que pretende pasar a la fama copiando el funcionamiento de subsistemas del viejo UNIX. Me interesa .NET más por su faceta de herramienta de desarrollo que por cualquier otra cosa.

- 1** Hay toda una familia de lenguajes que pueden utilizarse para desarrollar aplicaciones o componentes. En realidad, tras esa familia concreta de lenguajes, hay un conjunto muy bien definido de reglas del juego que deben ser satisfechas por cualquier aspirante a ingresar en tal selecto club.
- 2** Hay un sistema de soporte para la ejecución suficientemente complejo para que lo dividamos en subsistemas. Tenemos, para empezar, un sistema que carga las aplicaciones y las prepara para su ejecución. Tradicionalmente, esta ha sido una responsabilidad del sistema operativo y, efectivamente, Windows ha necesitado algunos retoques para satisfacer las exigencias de carga de las aplicaciones .NET.
- 3** Las aplicaciones .NET no sólo necesitan una tutela especial durante la carga, sino incluso durante la ejecución. Estas aplicaciones utilizan, por ejemplo, un recolector de basura para la gestión de memoria, y en el caso típico, es necesario traducir el código intermedio que generan los compiladores a código nativo ejecutable, a medida que la ejecución vaya activando una u otra ruta dentro de la aplicación. Hay tipos de datos cuya implementación se sintetiza en tiempo de ejecución: los tipos delegados, que se utilizan en la implementación de eventos y de funciones de respuesta, y los tipos genéricos en la versión 2.0 de la plataforma.

- 4 Por último, están las numerosas bibliotecas de clases necesarias para ejecutar hasta la más simple tarea.

Ahora busque un libro sobre sistemas y vea cómo el autor define qué es un sistema operativo. Se parece mucho a la lista anterior, ¿verdad? Esto ha llevado a que algunos sostengan lo siguiente:

.NET es el nuevo sistema operativo de Microsoft

Se trata de una exageración, por descontado. Por mencionar sólo un argumento: hay muchas áreas de las versiones actuales de Windows que no se adecuan naturalmente al estado actual de la plataforma. Muchos pensarán inmediatamente en los controladores de dispositivos, pero puede agregar el subsistema COM+ a esta lista. No obstante, esta idea de .NET como sistema operativo virtual no es desatinada, y Microsoft ya ha dado pasos importantes hacia esta meta, de momento lejana.

¿Por qué .NET y no Java?

Ha llegado el momento de romper corazones y herir almas sensibles. Si conoce algo sobre Java, aunque sea de oídas, habrá notado las similitudes entre Java y .NET. Hay un lenguaje intermedio, hay un traductor a código nativo, hay un intento de portabilidad... ¿Por qué un individuo como yo, que reniega de Java como afición personal, se muestra tan entusiasmado con algo que parece una imitación? Puedo resumir mis razones:

.NET es Java, pero bien diseñado y mejor implementado

.NET ha tenido la posibilidad de aprender de los errores en el diseño de Java, y doy fe de que se ha aprovechado esta oportunidad. Para demostrarlo, habría que efectuar una comparación área por área y muy bien detallada, pero puedo adelantarle algunos puntos:

- Entorno de ejecución: el *Common Language Runtime* de .NET ha sido, desde sus primeras versiones, más eficiente y funcionalmente completo que la *Java Virtual Machine*. Está fresca aún la enconada discusión sobre las bondades de añadir un mecanismo de tipos delegados a este tipo de máquinas virtuales. En la JVM no hay soporte para técnicas tan útiles como los tipos enumerados o el traspaso de parámetros por referencia. Esto se paga en velocidad de ejecución y en menor productividad, al tener el programador que tratar con un lenguaje menos expresivo.
- Otra manifestación de lo mismo: para no desvirtuar la universalidad de los metadatos en tiempo de ejecución, el equipo de Microsoft que introdujo los tipos genéricos, liderado por Don Syme y Andrew Kennedy, ideó una implementación que respeta toda la información sobre tipos genéricos, parámetros e instancias en tiempo de ejecución. La alternativa propuesta para Java está basada en trucos de compilación. ¿Resultado? Tanto la JVM de Java como el CLR de .NET deben ofrecer aplicaciones verificables. Los tipos genéricos de .NET son verificables y eficientes. Para que las instancias de tipos genéricos en Java sigan siendo verificables, el compilador debe añadir verificaciones de tipo en tiempo de ejecución que tienen un alto coste asociado.
- El propio concepto de código intermedio es muy diferente. El *bytecode* de Java muchas veces termina por ser interpretado. El código IL de .NET *nunca* se interpreta, sino que siempre se transforma en eficiente código nativo. Java presenta una paradoja: su popularidad inicial se debió precisamente a la presunta portabilidad de su *bytecode* y las posibilidades que se abrían respecto a la creación de *applets* para Internet. Sin embargo, los problemas con las interfaces gráficas terminaron por arrinconar el uso universal de *applets*, y Java ha encontrado su nicho ecológico en la programación para servidores... justo donde es más crítica la velocidad de ejecución y respuesta.
- Interfaces gráficas: ¿quién no se ha visto involucrado alguna vez en las tragedias javanesas relacionadas con AWT, Swing y el resto de los engendros visuales? Es cierto que Windows Forms en .NET v1.1 no es gran cosa, pero ha bastado una versión más para que la mejora sea más que notable. Además, podíamos criticar el primer Windows Forms por su fealdad o por la

falta de alguna funcionalidad... pero jamás por su ineficiencia o por estar repleta de *bugs* de todo tipo.

- Interfaz de acceso a datos: las interfaces oficiales de acceso a datos en Java son de muy bajo nivel, en comparación con .NET. Entre los programadores que usan Java es común una extraña adoración por las implementaciones manuales de sistemas de persistencia, especialmente aquellos que se desarrollan en solitario y partiendo siempre de cero absoluto. *DataSet* se traduce como “abominación” y la sola idea del *data binding* provoca espasmos epilépticos incluso a los jávicos más curtidos e insensibles.

La lista podría alargarse indefinidamente, pero voy a mencionar sólo otro punto de comparación: los llamados *servicios corporativos*. Estos serán tratados en la serie E de este curso, y sirven como soporte para aplicaciones divididas en capas que tienen que atender un número suficientemente grande de peticiones concurrentes. Cualquier versión de Windows, a partir de Windows 2000, viene de serie con soporte integrado para estos servicios, como parte de la funcionalidad de COM+. El equivalente en Java se conoce como J2EE... y es uno de los motivos por los que un servidor Linux puede terminar costando más que un servidor Windows, porque las implementaciones de estos servicios para Java suelen pertenecer a grandes empresas de software y costar un ojo de la cara.

Por último, es innegable que .NET cuenta con una ventaja abrumadora sobre Java:

En .NET, el sistema operativo y el entorno de ejecución colaboran estrechamente

¿Es injusto? Me parece que no... pero en cualquier caso, la Informática es mi profesión, no mi afición para el tiempo libre, y mi interés es poder llevar a cabo mi trabajo con la mayor calidad en el menor tiempo posible. Me importa un pimiento quién me suministre esas herramientas. Puedo equivocarme o puedo acertar, pero si me equivoco, las consecuencias las voy a tener que pagar de mi bolsillo. Por lo tanto, quiero ser yo, y no un maldito comisario político de la Unión Europea o un burócrata vitalicio federal, quien decida qué herramientas estoy obligado a usar. Reconozco que he escrito este párrafo con mucha rabia y mala sangre, pero es que detesto el intervencionismo estatal y por experiencia sé que nunca lleva a nada bueno. Le pido disculpas por el tono amargo, pero no he podido evitarlo.

Common Language Runtime

Y ahora, al grano: veamos el mínimo de conceptos necesarios para empezar a trabajar con la plataforma. Ya sabemos, y si no, se lo cuento ahora, que los compiladores .NET traducen el código fuente a un código intermedio llamado... bueno, código intermedio, o IL. No hay una pasada posterior de enlace, como veremos en la siguiente versión, sino que el resultado de esta compilación ya constituye directamente una aplicación ejecutable o una biblioteca de clases, o parte de alguna de estas dos variantes.

He dicho, sin embargo, que este código intermedio jamás se interpreta. ¿Quién es, por consiguiente, el responsable de la traducción a código nativo, y cuándo entra en escena? Hay dos alternativas:

- **Compilación sobre la marcha**

¿Sobre la marcha? Bueno, *just in time* no significa eso exactamente, pero me da pereza levantarme e ir a por un diccionario. Cuando usted lea esto habrán pasado ya varios meses desde que lo escribí, pero si observa bien, todavía sigue diciendo lo mismo, con lo cual puede hacer una idea de cuánta pereza me da...

Espero, de todos modos que la idea quede clara: Como parte del proceso de carga de la aplicación traducida a IL, uno de los componentes del sistema de carga realiza esta traducción sobre la marcha, según vaya siendo necesario. Inicialmente se traduce un pequeño fragmento; digamos, por concretar, que se traduce el método *Main* por donde comienza la ejecución del programa. Si este método contiene una llamada a otro método *MeDaLoMismo*, la instrucción que ejecuta ese método se truca para que salte en realidad a una rutina artificial. ¿Cuál es el código de esa rutina artificial? Muy sencillo: se traduce el código intermedio de esa rutina y se arreglan

las cosas para que una posible segunda llamada a *MeDaLoMismo* salte de cabeza al código nativo real y definitivo.

- **Compilación por adelantado**

Alternativamente, se puede compilar la aplicación durante su instalación. Antes de la instalación no es recomendable hacerlo, porque el resultado de esta operación depende en gran medida de la versión exacta del sistema operativo, del tipo de CPU, e incluso puede que de la memoria disponible. Este proceso se activa desde una aplicación de línea de comandos incluida en la plataforma y llamada *ngen*. Los detalles, tanto sintácticos como varían ligeramente con la versión de la plataforma. En la versión 2.0, por ejemplo, se puede diferir la traducción para que sea ejecutada en algún momento apropiado desde un servicio del ordenador.

Además de estos dos algoritmos básicos de traducción, hay que tener en cuenta las particularidades del equipo donde se ejecutará la aplicación. Por ejemplo, el traductor para Windows CE tiene en cuenta la poca memoria dinámica que es característica de los dispositivos donde debe ejecutarse. En estas circunstancias, el sistema operativo puede eliminar de la memoria dinámica la traducción de determinadas rutinas (siempre las menos usadas) y es posible que haya que traducir a código nativo una misma rutina más de una vez.

Imagino que, al llegar aquí, aquellos que todavía no sientan mucho afecto por .NET, sonreirán pensando que han encontrado la panacea. Una dosis de *ngen* a todo lo que compilen, y todas las pesadillas asociadas al código intermedio se desvanecerán. ¡Cuidado! Puede que pasar todo a código nativo no sea siempre una buena idea:

- Si la aplicación funciona como servidor, más temprano que tarde todo el código intermedia habrá sido traducido a código nativo, y el factor traducción deja de importar. Es una paradoja que sea preferible ejecutar *ngen* sobre una aplicación GUI, para las que se supone que la velocidad de procesamiento no es tan crítica, que ejecutarlo sobre un servidor, que suele ser el cuello de botella en muchos casos. Pero hay un candidato incluso mejor: las aplicaciones de línea de comando, como el compilador que estoy desarrollando para el lenguaje Freya¹. Es comprensible: cada vez que se ejecuta el compilador, inmediatamente después se descarga de la memoria, y en la próxima ejecución es necesario rehacer todo el trabajo de traducción. Por este motivo, el compilador de Freya se traduce a código nativo al ser instalado.
- Sorprendentemente, el compilador JIT en tiempo de ejecución puede realizar mejor trabajo que el compilador *ngen*. Hay varios motivos para ello, y no todos son evidentes. En tiempo de ejecución, por ejemplo, el traductor se puede hacer una idea mejor de las condiciones en las que se ejecuta la aplicación. Incluso puede jugar con la relocalización de bloques de código para minimizar la paginación. Un factor menos evidente es que en tiempo de ejecución ya se sabe cuáles ensamblados necesita realmente la aplicación. Esto permite realizar operaciones más agresivas de *inlining*, o expansión de código en línea.

Ensamblados y módulos

.NET no ofrece nada similar al *enlace estático* de código de otras plataformas de desarrollo, incluyendo Windows en modo “nativo”. Tomemos como ejemplo una clase: digamos que se trata de *Complex*, para el manejo de números complejos. En lenguaje tradicional, como C++, el programador que crea esta clase la compila para obtener una biblioteca de clases. En Windows nativo, esta biblioteca sería un fichero con extensión *lib*. Si a continuación, usted escribe dos programas que hace uso de esta clase, el código nativo correspondiente a la misma *se copia* dentro de sus dos aplicaciones. Aunque esto no es malo, los verdaderos problemas comienzan cuando usted diseña y compila dos bibliotecas dinámicas, como las DLL de Windows nativo, que hacen uso de *Complex*. Cada biblioteca cargaría con su propia copia de la clase, y si una aplicación necesita usar ambas bibliotecas dinámicas, en el espacio del proceso ejecutable tendríamos también dos copias indepen-

¹ freyalang.blogspot.com

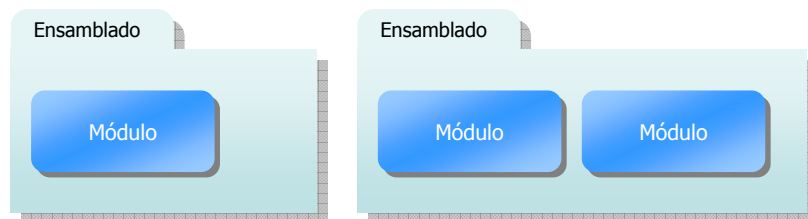
dientes de la clase de números complejos. Por supuesto, la solución evidente consiste en alojar la clase *Complex*, desde el primer momento, en una tercera biblioteca dinámica.

Los diseñadores de la plataforma .NET cortaron por lo sano: cada proyecto que usted compile generará, o un fichero ejecutable, o una DLL. ¿Quiere compartir código entre proyectos? Cree una biblioteca de clases, que siempre será de tipo dinámico. Con esta decisión no perdemos nada significativo. Es cierto que el sistema operativo tendrá un poco más de trabajo durante la carga de las aplicaciones, pero las ventajas que obtenemos como compensación son más que suficientes.

Existe, sin embargo, una máxima en Informática que, medio en broma, medio en serio, suele ser cierta:

Todo avance en Informática consiste en añadir otra capa de indirección.

En este caso, la capa de indirección consiste en que el programador no trabaja directamente con los ficheros físicos, sino que utiliza identificadores simbólicos para referirse a las bibliotecas de clases. Esto permite que varios ficheros físicos puedan combinarse de forma lógica en una misma biblioteca. A estos ficheros físicos se les conoce como *módulos*, y la unión de uno o más de estos módulos se llama *ensamblado*:



Un módulo sólo puede pertenecer a un ensamblado. Debo también precisar que los ficheros ejecutables también se consideran como un tipo especial de ensamblado. En realidad, esta es una explicación muy simplificada. Estoy dejando deliberadamente fuera el sistema de versiones, la firma de ensamblados para su autenticación, etc. Lo que ahora nos importa es añadir y gestionar referencias en un proyecto de Visual Studio.

Además de permitir la fragmentación de un ensamblado grande en trozos físicamente manejables, la principal utilidad de los módulos, al menos de los módulos compilados por usted y yo, es permitir que un ensamblado se cree a partir de proyectos desarrollados en lenguajes diferentes. Sin embargo, no es fácil trabajar con módulos, si nos atenemos a las plantillas predefinidas por Visual Studio. Para crear un módulo hay que pasar un parámetro especial al compilado de línea de comandos: como Visual Studio no trae plantillas para módulos, habría que indicar ese parámetro directamente en la configuración del proyecto. Para unir varios módulos en un ensamblado puede usarse también el correspondiente compilador de línea de comandos.

NOTA

He mencionado la existencia de módulos para explicar por qué hay ensamblados del sistema fragmentados entre varios ficheros físicos, y para no dejar espacios en blanco en la explicación de los ensamblados. No obstante, no volveremos a tratar con módulos en este curso, al menos de forma directa.

Clases y espacios de nombres

¿Qué contiene cada ensamblado en su interior? Lo principal: una lista de tipos de datos y sus implementaciones, cuando procede. Hay también referencias a otros ensamblados que necesita para su funcionamiento, y metadatos de todo tipo y nivel.

Cada tipo de datos tiene un nombre que consiste por lo general en una lista de identificadores separados por puntos. El punto, en realidad, es un carácter más dentro del nombre del tipo. De esta manera, tenemos tipos con nombres como:

- *System.String*
- *System.Data.SqlClient.SqlDataAdapter*

- *Microsoft.CSharp.CSharpCodeProvider*
- *IntSight.Proteus.Stargate.Entity*

Ahora bien, la mayoría de los lenguajes que soportan la plataforma interpretan este formato de nombres según un convenio recomendado por la plataforma; en realidad, no conozco lenguaje alguno que no siga el convenio mencionado. Este consiste en identificar el último identificador de la cadena como el nombre abreviado del tipo de datos. Los identificadores restantes se tratan como si fuesen nombres de *espacios de nombres*, o *namespaces*. Un espacio de nombre es simplemente un contenedor lógico que puede albergar otros espacios de nombre anidados, o tipos de datos. Por ejemplo, observe el siguiente nombre de clase:

System.Data.SqlClient.SqlDataAdapter

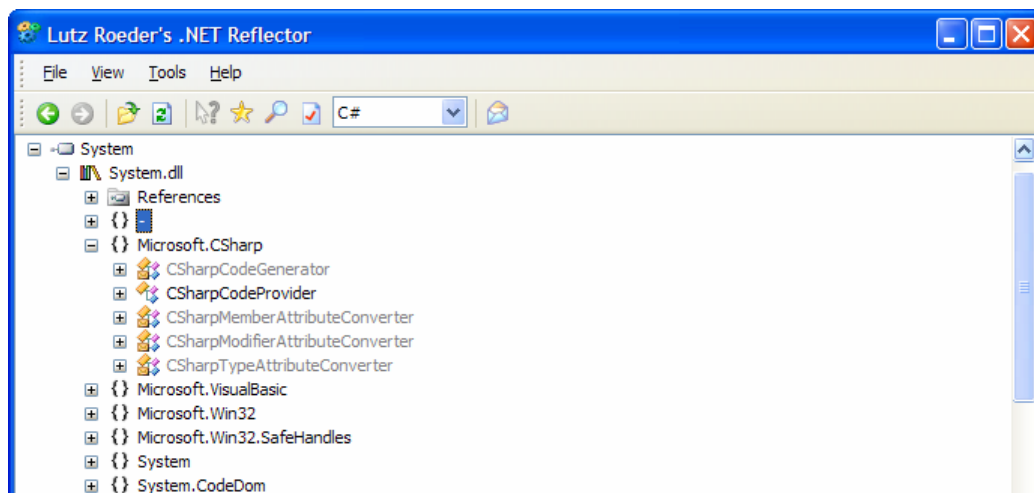
El nombre abreviado del tipo es *SqlDataAdapter*. El tipo se encuentra definido dentro de un espacio de nombres llamado *SqlClient*, definido dentro de *Data*, que finalmente se define dentro del espacio de nombres de primer nivel llamado *System*.

El problema con este convenio es que no existe una entidad “física” que represente al espacio de nombres. Esto, por regla general, es bueno: podemos definir clases “dentro” del espacio de nombres *System* en cualquier ensamblado. Eso mismo es, a la vez, un inconveniente. La única forma de saber qué espacios de nombres se utilizan dentro de un ensamblado es buscar todas las clases definidas dentro del ensamblado, eliminar los nombres abreviados de tipos y eliminar luego los duplicados. Este mismo problema es el que encontramos al definir las referencias de un proyecto: es fácil confundir un nombre de ensamblado con un espacio de nombres. En definitiva, ambos son listas de identificadores separados por puntos.

¿Cómo podemos saber qué contiene un ensamblado dado? Oficialmente, la plataforma ofrece una utilidad llamada *ildasm*, una aplicación de interfaz gráfica. Mi consejo, sin embargo, es que utilice otra herramienta para estos propósitos: el excelente y sin par *Reflector*, de Lutz Roeder, una aplicación gratuita que puede descargarse desde la siguiente página:

www.aisto.com/roeder/dotnet

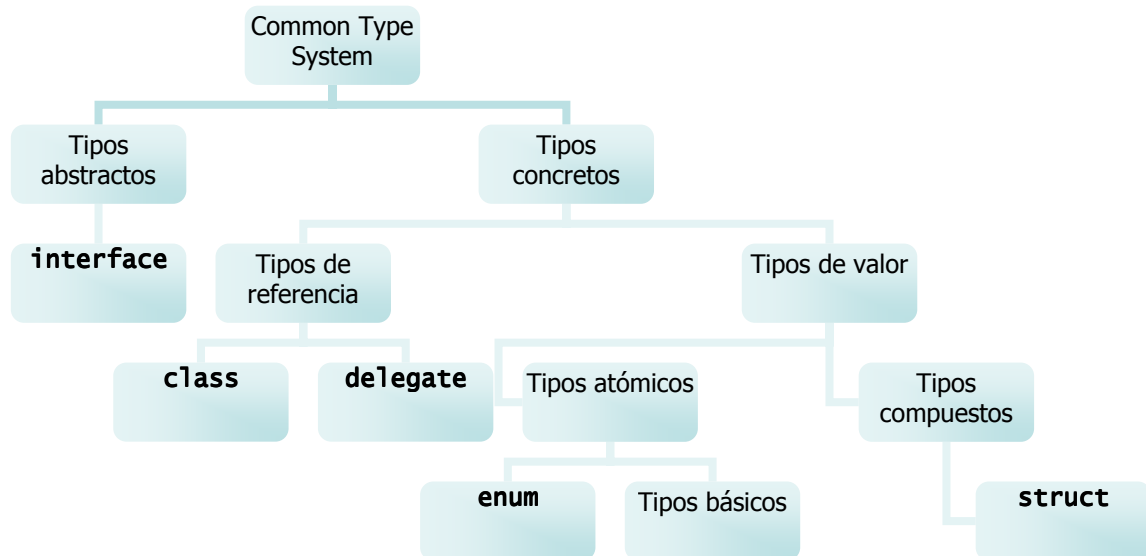
En el momento en que escribo estas líneas, la versión más reciente de *Reflector* es la 4.1.64.0, y funciona tanto con la versión 2.0 de la plataforma como con las versiones anteriores. Aquí tiene una imagen de esta aplicación:



El nodo superior del árbol corresponde al ensamblado *System*. Inmediatamente debajo aparece el módulo correspondiente al fichero *system.dll*. El módulo hace referencia a otros ensamblados, que no se muestran en la imagen, pero define una serie de clases que la aplicación reparte entre los correspondientes espacios de nombres. Por ejemplo, vemos un espacio *Microsoft.CSharp* que contiene varios tipos privados, en color gris, y el tipo *CSharpCodeProvider*, que al parecer es público y podemos usarlo en nuestras aplicaciones.

El sistema de tipos comunes

Para que todos los lenguajes soportados por .NET puedan comunicarse entre sí, deben compartir un mismo sistema de tipos y un mismo modelo de objetos. El siguiente diagrama muestra los distintos tipos de objetos definidos por el *Common Type System* (CTS), que es el nombre del sistema de tipos de .NET:



Este es un sistema de tipos más rico y expresivos que el soportado por Java. Este último lenguaje sólo incluiría, de los tipos anteriores, esta lista reducida:

- 1 Clases
- 2 Tipos de interfaz
- 3 Tipos básicos o atómicos

Java intenta utilizar el concepto de clase para modelar absolutamente todo, lo cual es correcto desde el punto de vista conceptual... y un verdadero incordio desde el punto de vista práctico. En .NET también se considera que la clase es el concepto fundamental, pero se toleran ciertas mutaciones del concepto para facilitar algunas técnicas de programación de uso frecuente.

Tipos básicos

Antes de complicarnos demasiado la vida, veamos qué es un tipo “básico” y cuáles de ellos nos ofrece la plataforma. No es una definición sencilla, entre otras cosas porque todos los lenguajes intentan disimular al máximo las posibles características especiales de estos tipos. Tomemos como ejemplo los enteros. Resulta que C# nos permite escribir expresiones como la siguiente:

```
999.ToString()
```

Claro, es una tontería escribir una expresión como la anterior... aunque no tanto, si la función *ToString* incluye una cadena de formato:

```
999.ToString("C")
```

¿Está seguro de que se trata de una expresión constante? En España imprime una cantidad en euros, claro... Lo que importa es que, a medida que los lenguajes orientados a objetos progresan, es cada vez más difícil distinguir entre tipos básicos y objetos. Esto es bueno para el programador, porque así debe recordar menos excepciones a las reglas, pero en el fondo se trata de un disfraz. Los objetos, ya sean instancias de clases o de estructuras, son tipos *compuestos* que, obviando la recursividad, deben definirse con la ayuda de tipos más simples.

A pesar de todo lo dicho, hay una forma inequívoca para diagnosticar si un tipo determinado se puede considerar “básico”: basta con mirar la documentación del *Common Language Runtime*, y averi-

guar si el tipo dado es reconocido como tal por la máquina virtual de ejecución. Por ejemplo, la máquina reconoce sin más los tipos enteros de 32 bits. Aunque no se puede trabajar directamente con enteros de 16 bits en la pila, existen operaciones para la conversión en las direcciones con los enteros de 32 bits. Hay otro indicio necesario, aunque no suficiente: todos los tipos básicos son *tipos de valor*, como veremos más adelante.

C# asigna una palabra clave para cada uno de los tipos básicos. Además, estos tipos tienen un nombre completo, como cualquier tipo de datos normal del *Common Type System*. Estos son los tipos básicos y sus nombres, tanto desde el punto de vista de C# como del *CTS*:

C#	CLR	Significado
bool	<i>System.Boolean</i>	Verdadero y falso, ¿no es así?
char	<i>System.Char</i>	Caracteres Unicode (¡16 bits cada uno!)
sbyte	<i>System.SByte</i>	Enteros de 8 bits con signo.
byte	<i>System.Byte</i>	Enteros de 8 bits sin signo.
short	<i>System.Int16</i>	Enteros de 16 bits con signo.
ushort	<i>System.UInt16</i>	Enteros de 16 bits sin signo.
int	<i>System.Int32</i>	Enteros de 32 bits con signo.
uint	<i>System.UInt32</i>	Enteros de 32 bits sin signo.
long	<i>System.Int64</i>	Enteros de 64 bits con signo.
ulong	<i>System.UInt64</i>	Enteros de 64 bits sin signo.
float	<i>System.Single</i>	Valores reales representados en 4 bytes.
double	<i>System.Double</i>	Valores reales representados en 8 bytes.

Hay autores, y entre ellos hay alguno famoso, que aconsejan no usar los nombre de tipos abreviados de C#. Aducen que así se facilita la conversión de código entre lenguajes soportados por la plataforma. De modo que, de hacer caso al consejo, para declarar una variable local de tipo entero tendríamos que escribir lo siguiente:

```
// ...
System.Int32 i = 0;
// ...
```

¿Se imagina un bucle **for**?

```
// ...
for (System.Int32 i = 0; i < 10; i++) {
    // ...
}
// ...
```

Me parece horrible. En primer lugar, porque no tengo ni la más mínima intención de programar con Visual Basic.NET... al menos, mientras pueda evitarlo. Es cierto que, en teoría, C# y este lenguaje son funcionalmente equivalentes; en la práctica, C# va siempre por delante. Pero nadie me paga para demostrar que soy un políglota de la programación. En segundo lugar, el que yo escriba **int** en mi código fuente no afecta al modo en que un hipotético programador en Visual Basic vería las declaraciones de mi ensamblado, ya que Visual Studio siempre se ocuparía de estos detalles sucios.

Hemos visto los tipos básicos, pero ¿no hay un tipo de datos para las cadenas de caracteres? Haylo, pero no es un tipo “básico”, sino una clase especial. Y lo mismo sucede con el tipo que se suele usar para representar dinero, pasta, parné, plata, guita²... o como quiera llamarlo. Lo notable respecto a estos dos señores es que tienen nombres especiales en C#:

C#	CLR	Significado
object	<i>System.Object</i>	¡La fuente de la vida! Al menos, teóricamente.
string	<i>System.String</i>	Cadenas de caracteres
decimal	<i>System.Decimal</i>	Valores decimales de alta precisión, con 128 bits de capacidad.

² Sólo las cosas *mu*y importantes tienen tantos nombres.

He añadido el tipo **object** a la tabla, porque se trata también de un tipo de clase con una palabra reservada a su disposición. La existencia de esta palabra clave se debe al mayor protagonismo de este tipo en las primeras versiones de la plataforma, en las que las operaciones de *boxing* y *unboxing* eran mucho más importantes que ahora, en la versión 2.0, donde los tipos genéricos las hacen innecesarias en la mayoría de los casos.

Finalmente, conviene presentar varios tipos relacionados con el acceso a datos:

CLR	Significado
<i>System.DateTime</i>	Fecha y hora.
<i>System.Guid</i>	Corresponde al tipo uniqueidentifier de SQL Server.
<i>System.DBNull</i>	Usado para representar valores nulos provenientes de una base de datos.

Estos tipos no tienen nombre propio en C#, y sólo los menciono porque los veremos con frecuencia en el curso. Sobre todo me interesa presentar la clase *DBNull*: cuando recuperamos el valor de una columna que contiene un valor nulo, recibimos un valor de tipo *DBNull*. Para ser exactos, recibimos el único valor que puede existir para esta clase. En su momento, veremos cómo averiguar si un valor leído con la ayuda de ADO.NET es nulo o no.

Qué hay dentro de una clase

En nuestro diagrama, las clases se clasifican como tipos concretos. Traducción: podemos crear “instancias”, es decir, objetos, a partir de una clase. Aclaremos que hay también clases marcadas a propósito como clases *abstractas*, pero en principio, estamos hablando de la capacidad de crear instancias, y no de si se aprovecha o no en un caso dado. Los tipos abstractos, de los que solo tenemos los tipos de interfaz en nuestro árbol, *nunca* pueden usarse para crear instancias.

Además, una clase es un tipo de *referencia*. Esto significa, en primer lugar, que una variable o campo perteneciente al tipo, almacena un puntero a una instancia del tipo, nunca directamente una de estas instancias. ¿Consecuencias? Unas cuantas, y entre ellas, éstas:

- Hace falta una instrucción especial para crear instancias de estos tipos de referencia. Esta operación tiene un coste asociado. Por sí mismo, es un coste pequeño, pero hay que añadirle el coste de la posterior devolución de la memoria, cuando el objeto ya no es necesario. Veremos luego cómo los tipos de estructuras pueden ayudar a evitar estos costes, en muchos casos.
- Dos variables diferentes pueden compartir un mismo objeto. Se puede modificar el estado de un mismo objetos por medio de variables, o campos, sin relación aparente.

Una clase es un tipo compuesto: sirve como contenedor y organizador para otras entidades definidas en su interior. Las clases en .NET pueden contener estos tipos de declaraciones:

- Campos: variables declaradas dentro de la clase. Es costumbre prohibir el acceso directo a los mismos desde fuera del objeto.
- Métodos: todo el código ejecutable en un lenguaje orientado a objetos debe estar encerrado dentro de *métodos*, es decir, funciones definidas dentro de una clase. Normalmente, estos métodos actúan sobre un objeto de la clase dentro de la cuál han sido declarados. Para llamar uno de estos métodos se debe indicar la instancia sobre la cuál queremos *aplicar* dicho método; si no especificamos tal instancia, se asume el uso de la instancia implícita del método donde se realiza la llamada. Esto implica, por supuesto, que tanto el método llamado como el método que contiene la llamada pertenecen a la misma clase... en un sentido relajado, porque hay que tener en cuenta la herencia de clases. Lo siento, pero es así de complicado. Para terminar de liarle la vida, veremos dentro de nada que existen también métodos y campos estáticos, que obedecen a un conjunto de reglas ligeramente distintas.
- Constructores: parecen métodos, porque contienen código ejecutable, pero sólo pueden usarse dentro de una expresión **new**, o usando una sintaxis especial, antes de que empiece a ejecutarse el código de otro constructor. No es necesario advertir que los constructores *construyen* instancias de objetos a partir de clases... y bueno, de estructuras. Ah, y si tiene alguna experiencia con otros lenguajes orientados a objetos, sepa que C# soporta algo que se parece mucho a un des-

tructor (¡incluso se llaman destructores!) pero que no se parecen en nada a un destructor tradicional.

- **Propiedades:** campos con superpoderes. Simulan la lectura del valor de un campo por medio de una función, y la escritura mediante otra. Esto permite añadir efectos secundarios a las lecturas y escrituras de miembros de la clase que aparentan ser campos, y es uno de los pilares de la programación mediante componentes.
- **Eventos:** es el mecanismo preferido para el envío de notificaciones entre componentes. Un evento utiliza casi siempre un campo oculto para mantener una cadena de punteros a métodos. Cuando la clase que define el evento lo considera apropiado, activa esa cadena de funciones mediante el “disparo” del evento. Microsoft prefiere que se hable de la “elevación” (*raising*) del evento... pero da igual.

Cuando declaramos un campo, lo normal es que se reserve memoria para éste en cada una de las instancias de la clase. Si le parece anormal la palabra “normal” para hablar de estos campos, le permito que también los llame campos de instancia. Podemos también usar el modificador **static** en la declaración de campos, propiedades, eventos, métodos e incluso constructores. Por ejemplo:

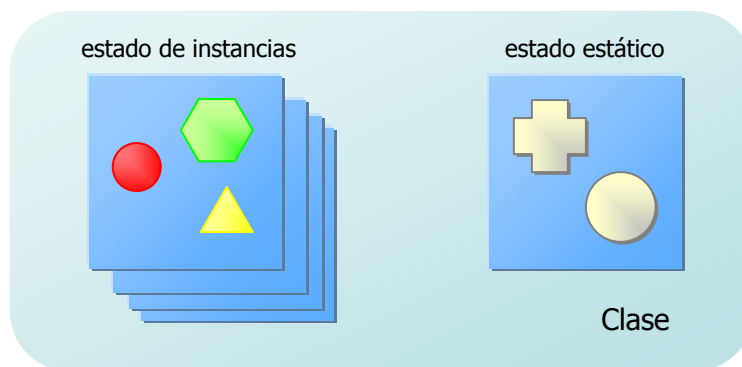
```
// Un campo de instancia
public string nombre;

// Un campo "estático"
public static string prefijo;
```

La particularidad de los campos estáticos es que sólo se reserve memoria para cada declaración una vez por clase. Siguiendo las declaraciones anteriores, tendremos a nuestra disposición un campo *nombre* por cada instancia que creemos de la clase donde ha sido declarado. Sin embargo, sólo tendremos un campo *prefijo*, sin importar el número de instancias creadas. De hecho, ¡tendremos un campo *prefijo* incluso antes de que creemos instancia alguna!

Un método “normal”, quiero decir, de instancia, actúa sobre el estado de los campos y propiedades de una instancia que debemos suministrarle, ya sea de forma explícita o implícita. Cuando declaramos un método estático, éste no tiene acceso a los campos de instancia, como era de esperar, sino solamente a los campos estáticos declarados dentro de la clase. Y tampoco necesitamos pasarle una instancia al método: nos basta con indicar, implícita o explícitamente, el nombre de la clase donde ha sido declarado el método.

Esto hace que podamos decir que, al declarar una clase con miembros estáticos, es como si estuviésemos declarando dos clases “normales”, sin miembros estáticos:



Una de estas clases imaginarias sólo contendría las declaraciones de instancias. La otra clase imaginaria implícita contendría todas las declaraciones que utilizan el modificador **static**, con la peculiaridad de que de esta segunda clase imaginaria sólo tendríamos una instancia a nuestra disposición, que siempre estaría creada... más o menos. No es una metáfora para ser tomada muy en serio, pero si el CLR no ofreciese soporte explícito para miembros estáticos, un compilador de un lenguaje como C# podría simularlos mediante este truco de las dos clases, de forma oculta para el programador.

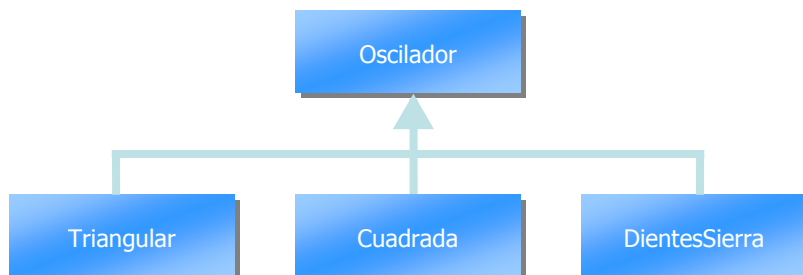
Herencia e interfaces

Puede que el truco más popular de la chistera de la Programación Orientada a Objetos sea la *herencia de clases*. Cuando declaramos una clase e indicamos que descende de otra, hacemos que nuestra clase “herede” todas las declaraciones de la clase ancestro, con la posibilidad de añadir nuevos miembros y modificar algunos de los heredados. Se dice en estos casos que heredamos tanto la estructura como el comportamiento.

En un momento dado, los teóricos de la programación pensaron que era deseable que una clase pudiese heredar de más de una clase base. Pero en la práctica, los lenguajes que soportaban herencia múltiple, como el popular C++, demostraron que:

- Muchas de las técnicas necesarias para evitar conflictos de nombres eran, o demasiado complejas, o demasiado limitantes.
- Como consecuencia, muchos programadores ni siquiera se aventuraban a usarla.
- Una buena parte de los osados cometía errores de difícil diagnóstico.
- Era un verdadero dolor en las vértebras implementar un compilador con soporte para herencia múltiple.
- Da la casualidad que el principal caso práctico en el que merece la pena utilizar herencia múltiple se puede resolver con otra técnica que enseguida veremos.

El misterioso caso tiene que ver con las llamadas *clases abstractas*. Una clase puede declarar que uno o más de sus métodos es abstracto, para que sea implementado en alguna clase derivada. Esta situación es relativamente frecuente cuando se diseñan jerarquías de herencia. Por ejemplo, para programar un sintetizador de sonidos por software (están de moda, por cierto), se puede partir de una clase base llamada *Oscilador*, de la que derivaremos luego varias clases que proporcionen el comportamiento de distintas formas de onda. En un diagrama, representaríamos tal situación de esta manera:



El triángulo de la imagen no es cuestión estética: se utiliza en UML para representar las relaciones de herencia. En código, tendríamos esto, si nos ahorramos detalles innecesarios:

```
public abstract class Oscilador
{
    public abstract double Voltaje(double tiempo);

    // Otros miembros, no necesariamente abstractos...
}

public class Triangular
{
    public override double Voltaje(double tiempo)
    {
        // Implementación de la onda triangular ...
    }

    // Otros miembros ...
}
```

Podemos llegar al extremo y declarar alguna que otra clase solamente con métodos, propiedades y eventos abstractos; no tiene sentido hablar de campos abstractos, y .NET no soporta constructores

abstractos. El caso especial al que antes me refería consiste en declarar una clase que herede de una sola clase concreta y de una o más clases abstractas. ¿Qué hay de especial en este caso?

- La estructura física se hereda de la única clase concreta. Esto simplifica mucho la implementación de compiladores.
- Los conflictos y ambigüedades, aunque no desaparecen del todo, se reducen al mínimo posible.
- Desde el punto de vista del sistema de tipos, la nueva clase sigue siendo compatible para la asignación respecto a todas sus clases bases.

Para evitar la herencia múltiple y seguir soportando este útil caso especial, los lenguajes como Java y C# permiten el uso de *tipos de interfaz*. Estos tipos son muy similares a las clases, pero sólo permiten declaraciones de métodos, propiedades y eventos. Y muy importante: para ninguno de estos miembros se puede suministrar una implementación. Por ejemplo, en vez de declarar una clase abstracta para definir las habilidades de un oscilador digital, podríamos declarar una interfaz:

```
public interface IOscilador
{
    double Voltaje(double tiempo);
}
```

Los tipos de interfaz se interpretan como especificación de una funcionalidad determinada. La interfaz *IOscilador* es muy sencilla: describe elementos que pueden suministrar un voltaje que varía con el tiempo. Ahora tenemos que establecer las reglas que deben cumplir C# y la mayoría de los lenguajes .NET:

- 1 Toda clase debe *heredar* exactamente de una sola clase.
- 2 La excepción: la clase *System.Object*; en C#, **object** a secas.
- 3 Una clase puede omitir la clase base en su declaración, y en tal caso se asume que descende de *System.Object*.
- 4 Por el contrario, una clase puede *implementar* cero o más tipos de interfaz.

He resaltado *heredar* e *implementar* porque esa es la terminología oficial. *Implementar* significa que una clase menciona el tipo de interfaz en su declaración, con lo cual se obliga a implementar los métodos, propiedades y eventos declarados en la interfaz:

```
public class Triangular : Chip, IOscilador, IConfigurable
{
    public double Voltaje(double tiempo)
    {
        // Implementación de la onda triangular ...
    }

    // Otros miembros ...
}
```

El primer tipo de la lista de herencia debe ser una clase. En este caso, me he inventado una clase a la que he llamado *Chip*, para mostrar que ahora tenemos más libertad al elegir una clase base. A continuación se menciona *IOscilador*, y efectivamente: la clase *Triangular* implementa su método *Voltaje*. Observe que he mencionado otro tipo de interfaz, *IConfigurable*: esto podría indicar que nuestra clase puede leer y guardar su configuración desde algún tipo de medio persistente, como un fichero XML, el registro de Windows o incluso una base de datos.

Encapsulamiento y control de acceso

Cada miembro declarado dentro de una clase o estructura en C# debe ir acompañado de un modificador que indique su *nivel de acceso*: desde qué zonas de un proyecto puede ser utilizado dicho miembro. El objetivo de este control de acceso es ocultar la mayor cantidad de detalles de la vista de los potenciales usuarios de la clase. No se trata de esconder ideas o secretos técnicos, sino de simplificar la forma de usar la clase y evitar dependencias innecesarias. En realidad, para un usuario con los permisos necesarios no hay secretos dentro de un ensamblado .NET. Si esto fuese un problema, debería utilizar técnicas adicionales como el ofuscamiento del código intermedio.

El modelo de control de acceso en C# está determinado en su mayor parte por la implementación del CLR y ofrece cinco niveles. Tres de ellos tienen mucho en común con el modelo de acceso de lenguajes orientados a objetos clásicos como C++:

- **public**
Los miembros declarados **public** pueden ser utilizados desde cualquier parte del proyecto, ya sea en el mismo ensamblado donde se está declarando la clase, como desde un ensamblado diferente.
- **private**
Por el contrario, los miembros **private** sólo pueden ser utilizados por la propia clase que los declara.
- **protected**
Los miembros **protected** pueden ser utilizados por la propia clase que los declara, y además, por cualquier clase que descienda de la clase original, no importa en qué ensamblado se encuentre.

Hay otros dos niveles de acceso que tienen que ver con los ensamblados:

- **internal**
Un miembro declarado **internal** sólo puede ser utilizado por clases ubicadas dentro del mismo ensamblado que la clase original. Es una versión menos estricta de **private**.
- **protected internal**
Un miembro declarado **protected internal** puede ser utilizado desde cualquier clase ubicada dentro del mismo ensamblado. Fuera de éste, sólo tendrán acceso al miembro las clases derivadas de la clase original.

Tenga presente que estos cinco niveles se refieren a miembros declarados dentro de una clase o estructura, incluyendo otros tipos anidados. En cambio, para los tipos declarados directamente dentro de un espacio de nombre, es decir, para aquellos tipos que no son tipos anidados, sólo hay dos niveles aplicables:

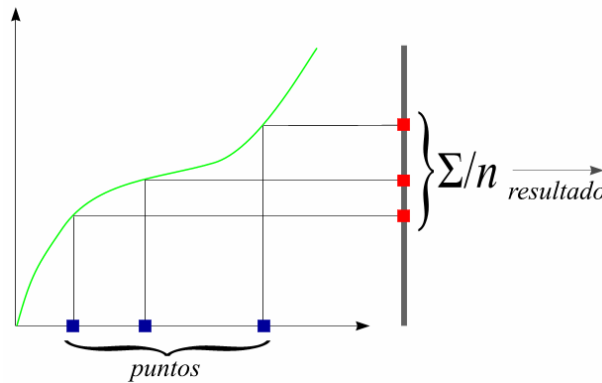
- **internal**
El tipo sólo puede ser usado dentro del ensamblado.
- **public**
¡Barra libre!

Se asume que todos los miembros declarados dentro de un tipo de interfaz son públicos, por lo que no se debe usar un modificador de acceso en la declaración de estos. Debe saber también que algunos miembros especiales son considerados automáticamente como privados. En esta categoría se encuentran los *constructores de tipos*, también llamados *constructores estáticos*, los métodos usados para la implementación explícita de tipos de interfaz y los *finalizadores* (página 22).

Delegados

¿Me permite una pregunta tonta? Asumo que sí, de modo que ahí va: ¿por qué llamamos *variables* a las *variables*? Lo malo de hacer estas preguntas es que termina por contestarlas uno mismo. Las llamamos variables porque pueden contener un valor que *varía*. Si es una variable entera, puede que contenga un cero, un seiscientos sesenta y seis o un menos uno. Si es una variable lógica, sólo puede contener verdadero o falso. Gracias a este hecho tan elemental, querido Watson, podemos desarrollar algoritmos que actúen sobre un valor entero arbitrario, o sobre un valor lógico arbitrario, etcétera.

¿Podemos tener *variables de funciones*? Si fuese posible, podríamos desarrollar algoritmos que actuaran sobre una función arbitraria. Imagine, por ejemplo, un algoritmo que calcule la media de los valores de una función real evaluada en una lista de puntos.



Para concretar, suponga que tomamos la función raíz cuadrada, y queremos promediar el resultado de esta función en una lista de puntos pasada como parámetro:

```
public static double MediaRaiz(double[] puntos)
{
    // Asumiremos que el vector puntos no está vacío.
    double total = 0.0;
    for (int i = 0; i < puntos.Length; i++)
        total += Math.Sqrt(puntos[i]);
    return total / puntos.Length;
}
```

¿Y si queremos usar la función exponencial? Tendríamos que modificar el bucle de esta forma:

```
for (int i = 0; i < puntos.Length; i++)
    total += Math.Exp(puntos[i]);
```

Viendo que se trata de modificaciones mínimas, ¿por qué no programar la función *Media* de forma tal que acepte *cualquier* función real que le suministremos? En los dos ejemplos anteriores, he resaltado en rojo la función utilizada. Para lograr nuestro propósito, tendríamos que sustituir la llamada a una función concreta por una llamada a una función... que puede variar. He aquí que necesitamos una *variable de función*: una variable, o parámetro, que contenga una “referencia” a una función. Está claro que no puede ser una función cualquiera: las funciones que debe aceptar dicha variable deben admitir un parámetro de tipo real y retornar un valor también real. Necesitamos poder programar lo siguiente:

```
public static double MediaRaiz(double[] puntos,
    TipoEspecialAunPorDeterminar funcion)
{
    double total = 0.0;
    for (int i = 0; i < puntos.Length; i++)
        total += funcion(puntos[i]);
    return total / puntos.Length;
}
```

Parece convincente... pero si le cuenta esta historia a su profesor de Java, verá como frunce el ceño y, como mínimo, le llama “psicópata pervertido que nunca comprenderá la belleza de la OOP”.

NOTA

A Microsoft, modificar la JVM para permitir estas cosas le costó un pleito de aúpa con Sun, que además perdió. Ya es triste que el J++ mutante fuese más rápido y eficiente que el trilobites antediluviano de Jaime Gansito y compañía. Más triste aún, si cabe, es ver que quienes aplauden el veredicto de ese juicio son luego quienes se oponen más radicalmente a las patentes de software. Había una vez una hermosa dama llamada Coherencia, alabada por todos pero amada por nadie...

Me ataré una mano a la espalda para mostrar dos formas de bordear el problema al estilo Java, aunque usaré C#. La primera consiste en encapsular el algoritmo de la media dentro de una clase:

```
public abstract class Promediador
{
    protected abstract double Funcion(double valor);
}
```

```

public double Calcular(double[] puntos)
{
    double total = 0.0;
    for (int i = 0; i < puntos.Length; i++)
        total += Funcion(puntos[i]);
    return total / puntos.Length;
}

```

Imagino que ya estará viendo el disparate. Cada vez que quiera usar el algoritmo con una nueva función, tendrá que crear una clase derivada:

```

public class PromedioRaiz : Promediador
{
    protected override double Funcion(double valor)
    {
        return Math.Sqrt(valor);
    }
}

```

¡Una clase para poder usar una función! Le aseguro, además, que esto hace más lento el algoritmo, porque para evaluar la raíz en cada punto se realiza primero una llamada virtual, y luego viene la llamada a la verdadera función. Para poder ejecutar el algoritmo, necesitaremos crear también una instancia de la nueva clase:

```

double[] puntos = new double[]{ 0.0, 1.0, 2.0 };
Promediador prom = new PromedioRaiz();
Console.WriteLine(prom.Calcular(puntos));

```

En realidad, casi nadie seguiría este camino de perdición: un javanés tomado al azar que posea un mínimo de sensatez usaría un tipo de interfaz:

```

public interface IFuncion
{
    double Funcion(double valor);
}

```

Podríamos programar ahora una función similar a la presentada al iniciar la sección, usando el tipo de interfaz en el papel del *TipoEspecialAunPorDeterminar*:

```

public static double MediaRaiz(double[] puntos, IFuncion funcion)
{
    double total = 0.0;
    for (int i = 0; i < puntos.Length; i++)
        total += funcion.Funcion(puntos[i]);
    return total / puntos.Length;
}

```

¿Le parece interesante? Si responde que sí, es que no se ha dado cuenta todavía de que estamos repitiendo casi al pie de la letra la tontería de la clase abstracta. Para que esto funcione, seguimos necesitando una nueva clase para cada función que se nos ocurra promediar. Lo único que ha cambiado es que, donde antes había herencia de clases, ahora tenemos la implementación de un tipo de interfaz:

```

public class PromedioRaiz : IFuncion
{
    public double Funcion(double valor)
    {
        return Math.Sqrt(valor);
    }
}

```

La forma de ejecutar el algoritmo tampoco ha cambiado mucho:

```

double[] puntos = new double[]{ 0.0, 1.0, 2.0 };
IFuncion raizCuadrada = new PromedioRaiz();
Console.WriteLine(MediaRaiz(puntos, raizCuadrada));

```

¿Listo para la solución “de verdad”?

```
public delegate double Funcion(double valor);

public static double MediaRaiz(double[] puntos, IFuncion funcion)
{
    double total = 0.0;
    for (int i = 0; i < puntos.Length; i++)
        total += funcion(puntos[i]);
    return total / puntos.Length;
}
```

La primera línea es una declaración de tipo: declaramos un tipo *Funcion* que representa aquellas funciones o métodos que reciben un valor de doble precisión y devuelven un valor del mismo tipo. Veamos ahora cómo usar el algoritmo:

```
double[] puntos = new double[]{ 0.0, 1.0, 2.0 };
Console.WriteLine(prom.Calcular(puntos, new Funcion(Math.Sqrt)));
```

Puede que me esté llamando tramposo. ¿Por qué he omitido la variable temporal que sí he utilizado con las soluciones tipo Java? Quizás debería haber escrito esto:

```
double[] puntos = new double[]{ 0.0, 1.0, 2.0 };
Funcion raizCuadrada = new Funcion(Math.Sqrt);
Console.WriteLine(MediaRaiz(puntos, raizCuadrada));
```

Pero es que en C# 2.0 se puede usar una sintaxis abreviada:

```
double[] puntos = new double[]{ 0.0, 1.0, 2.0 };
Console.WriteLine(prom.Calcular(puntos, Math.Sqrt));
```

No es la brevedad lo que más me gusta de esta solución: repase los ejemplos anteriores, para que vea que estas dos líneas son las que mejor se comprenden, las que mejor expresan el problema planteado y su solución. El tipo *Funcion* nos permite declarar variables, campos y parámetros para representar las *variables de función* por las que suspirábamos al principio. Y gracias a la nueva sintaxis de C# 2.0, podemos usar el nombre de una función concreta, como *Math.Sqrt*, ¡como si se tratase de una *constante de función*! Compare:

```
// A una variable entera le asignamos una constante entera
int i = 0;

// A una variable de función le asignamos una constante de función
Funcion f = Math.Sqrt;
```

Observe que, en la segunda instrucción no se está pidiendo la ejecución de la función que calcula la raíz cuadrada. La diferencia es pequeña y merece ser destacada:

```
// A una variable real le asignamos el resultado de una función
Funcion f = Math.Sqrt(3.0);
```

Es la ausencia de paréntesis, a continuación del nombre de la función, la que indica que queremos una referencia a la función, no el resultado de su evaluación.

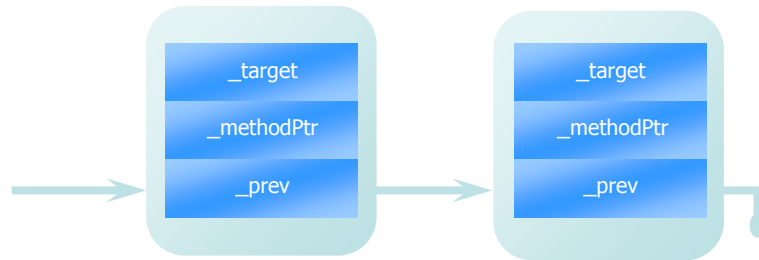
Cadenas de delegados

He simplificado un poco las cosas en la explicación anterior, de modo que los tipos delegados de .NET se pareciesen lo más posible a los punteros a funciones de la programación más tradicional. Los punteros a funciones han estado en C desde siempre, y fueron añadidos a Pascal poco después de su creación. Era más complicado añadir este recurso a Pascal principalmente por culpa de los procedimientos anidados de este lenguaje, que ya de por sí añaden complejidad a la implementación de compiladores. La solución más popular ha sido permitir que estos punteros sólo puedan hacer referencia a funciones y procedimientos no anidados, tal como ocurrió en Turbo Pascal 4.

De todos modos, hay una pequeña diferencia entre los punteros a funciones tradicionales y los delegados: en un lenguaje orientado a objetos puro, los métodos no existen separados de las clases,

como entidades independientes. Cuando guardamos una referencia a un método en un delegado, el diseñador del lenguaje debe tomar una decisión: ¿debe el delegado guardar también la referencia a un objeto en particular, o no? C++, que no es un lenguaje “puro”, opta por la vía negativa. Los lenguajes .NET siguen la vía contraria: un delegado, en realidad, almacena una referencia opcional a un objeto, además de la referencia al método en sí. La referencia al objeto es opcional para que los delegados se puedan usar también con métodos estáticos, que como sabemos, no necesitan una instancia para ser ejecutados.

Y hay más complicaciones: los valores delegados pueden formar cadenas de referencias a distintos métodos. El siguiente diagrama muestra, al fin, la estructura interna de una instancia de un tipo delegado y el mecanismo usado para formar las cadenas mencionadas:



En primer lugar, y como sugiere el diagrama, los delegados son tipos de referencia: .NET los implementa como un tipo muy particular de clases. Ese es el motivo por el que había que utilizar el operador **new** en C# v1.0 para obtener un delegado:

```
Funcion raizCuadrada = new Funcion(Math.Sqrt);
```

Esto sigue siendo cierto en la versión 2.0, pero el compilador ahora puede añadir por su cuenta la llamada al constructor:

```
Funcion raizCuadrada = Math.Sqrt;
```

Se asume que todos los tipos delegados descienden de la clase especial *MulticastDelegate*, que a su vez deriva de *Delegate*, la cual desciende directamente de *Object*. Al parecer, Microsoft consideró en algún momento la posibilidad de ofrecer delegados simples y combinables, pero en la versión final sólo sobrevivieron los delegados combinables que ahora conocemos.

El diagrama nos muestra también la existencia de tres campos ocultos dentro de una instancia de delegado. El primero de ellos, bautizado *_target* en la ilustración, almacena una referencia a un objeto; en el caso de que el delegado se refiera a un método estático, se guarda un puntero nulo en este campo. El campo *_methodPtr* contiene una referencia al método concreto al que el delegado apunta. Sospecho que este valor debe interpretarse como una dirección dentro del segmento de código de la aplicación. Finalmente, el campo *_prev* apunta a otro delegado del mismo tipo, y es el truco que se utiliza para implementar cadenas de delegados.

¿Qué sentido tiene combinar delegados en una cadena? Si se trata de funciones matemáticas, como la raíz cuadrada o la exponencial, no tiene ningún sentido. Si activamos el delegado, se ejecutarán cada una de las funciones de la cadena, pero sólo podremos usar el valor de retorno de la última función que se ejecute. Algo muy diferente ocurre cuando trabajamos con métodos que provocan efectos secundarios. Veamos un ejemplo sencillo:

```
public delegate void Mensaje();
```

Este delegado se puede conectar a métodos sin parámetros y sin valor de retorno. Definamos dos métodos compatibles con este tipo. Para simplificar, usaremos métodos estáticos:

```
public static void BuenosDias()
{
    Console.WriteLine("Buenos días");
}
```

```
public static void Adios()
{
    Console.WriteLine("Adiós");
}
```

Declaremos una variable y construyamos una cadena de delegados:

```
Mensaje cadena = new Mensaje(BuenosDias) + new Mensaje(Adios);
```

Como ve, podemos utilizar el operador de adición para componer cadenas de delegados. En el fondo, el operador ejecuta el método estático *Combine* de la clase *Delegate*. Ejecutemos ahora los métodos de la cadena:

```
cadena();
```

Hemos usado la variable como si fuese el nombre de una función, y le hemos pasado los parámetros necesarios. En nuestro caso, no hay parámetros, por supuesto.

Eventos

¿Por qué Microsoft se tomó tantas molestias para permitir la combinación de delegados en cadenas? La respuesta es que podemos usar los tipos delegados para declarar *eventos* dentro de una clase. Ya he presentado los eventos cuando veíamos qué tipo de miembros puede contener una declaración de clase. Muchos programadores confunden al principio los dos conceptos, el de evento y el de tipo delegado, pero basta con recordar un par de hechos para aclarar las ideas:

- Un delegado es un tipo de datos.
- Un evento es un tipo de miembro que es admitido por las clases, estructuras y tipos de interfaz. El evento se declara utilizando un tipo delegado.

Antes dije también que los eventos servían para implementar notificaciones. En efecto: una clase como *Button* puede declarar un evento llamado *Click*. Los usuarios de la clase pueden enganchar delegados a la cadena de delegados que puede colgar de este evento. Este acto se interpreta como una *suscripción* al evento. Cuando la clase *Button* detecta que alguien ha pulsado el control, activa los métodos enganchados a la cadena definida por *Click*.

Un evento, en una primera aproximación, se parece mucho a una propiedad de tipo delegado... pero sería incorrecto. Para ver la diferencia, usaremos el siguiente tipo delegado, definido en *System*:

```
public delegate void EventHandler(object sender, EventArgs e);
```

No nos interesa ahora el papel de estos parámetros, pero debe saber que, aunque potencialmente podríamos declarar eventos con tipos arbitrarios de delegados, .NET establece el convenio de que un tipo delegado asociado a un evento debe tener siempre dos parámetros. El primero de ellos debe ser siempre de tipo **object**, y el segundo debe pertenecer a la clase *EventArgs*, o a una clase derivada de la misma.

Ahora declaremos una propiedad y un evento dentro de una clase, usando el tipo *EventHandler*:

```
public class Clase01
{
    private EventHandler campo;

    public EventHandler Propiedad
    {
        get { return campo; }
        set { campo = value; }
    }

    public event EventHandler Evento;
}
```

La primera diferencia es que una propiedad exige una implementación, mientras que sólo declaramos el evento y dejamos que el compilador proporcione una implementación. Esta no es una

diferencia fundamental, sin embargo; entre otras razones porque podemos darle una implementación explícita al evento. La diferencia realmente importante se manifiesta durante el uso de la propiedad y el evento. Para concretar, suponga que estamos dentro de una segunda clase:

```
public class Clase02
{
    private void ManejadorCompatible(object sender, EventArgs e)
    {
        System.Console.WriteLine("yoo-hoo!");
    }

    public void Prueba(Clase01 cls01)
    {
        // Usemos la propiedad
        cls01.Propiedad = new EventHandler(ManejadorCompatible);
        // Usemos el evento
        cls01.Evento += new EventHandler(ManejadorCompatible);
        // ¡Esto produce un error de compilación!
        cls01.Evento = new EventHandler(ManejadorCompatible);
    }
}
```

Podemos asignar un valor a una propiedad, siempre que ésta permita escrituras. Pero no podemos asignar directamente un valor a un evento. No obstante, parece que podemos añadir, con el operador de asignación y suma, y le aseguro que también podemos “quitar” de un evento, con el imaginable operador compuesto de asignación y resta:

```
// ¡Sería también correcto!
cls01.Evento -= new EventHandler(ManejadorCompatible);
```

La clave está en las cadenas de delegados. Cuando exponemos una propiedad de tipo delegado, estamos permitiendo a que un usuario de la clase que la contiene añada manejadores o quite manejadores de la cadena de delegados a cuyo inicio apunta la propiedad. Pero también nos arriesgamos a que un programador despistado asigne sin más su propio manejador a la propiedad, y se cargue de este modo una hipotética cadena ya existente. Eso es precisamente lo que se evita al declarar un evento: el evento no puede ser asignado desde fuera de la clase donde se define, y sólo podemos usar los dos operadores mencionados para su manipulación.

Estructuras

Las *estructuras* (*structs*) son otra clara ventaja de .NET frente a Java. Mientras que las instancias de clases siempre utilizan memoria dinámica, las instancias de estructuras residen normalmente en el segmento de datos estáticos, como campos estáticos, o “aplanadas” dentro de la memoria reservada para una instancia de clase, o directamente en la memoria de pila local de un método.

Para entender por qué necesitamos estructuras, considere lo que ocurre en una típica aplicación de interfaz gráfica: en la capa de software que está detrás de los controles, hay un uso intensivo de las primitivas de dibujo soportadas por el entorno de ejecución. La inmensa mayoría de estas funciones necesitarán puntos para funcionar. Podríamos representar estos puntos mediante una clase, pero ¿se imagina la cantidad de puntos de usar y tirar que necesitaríamos cada vez que fuese necesario dibujar *cualquier tontería*? En un sistema con recolección automática de basura, el algoritmo de recogida se estaría activando cada dos por tres. Este razonamiento puede aplicarse a casi cualquier dominio de programación: siempre habrán tipos sencillos como los puntos, de los que necesitaremos cantidades industriales, que se convertirán en una amenaza para el eficiente funcionamiento de las aplicaciones. Donde Java se resigna a lo aparentemente inevitable, .NET ofrece los tipos de estructuras como solución efectiva.

Las ventajas de los tipos de estructuras se pagan mediante algunas restricciones que padecen. Por ejemplo:

- No podemos definir un constructor sin parámetros para una estructura. Estamos obligados a convivir con el constructor sin parámetros que sintetiza el sistema.

Este constructor implícito, además, inicializa con ceros todos los campos de la estructura. Suponga que definimos un tipo para representar vectores tridimensionales:

```
public struct Vector
{
    private double x, y, z;

    public double X { get { return x; } set { x = value; } }
    public double Y { get { return y; } set { y = value; } }
    public double Z { get { return z; } set { z = value; } }
}
```

El constructor implícito asignaría cero a las tres componentes del vector.

- Los campos de una estructura no pueden tener una expresión de inicialización.

Esta es una consecuencia inmediata de la restricción anterior. Si pudiésemos adjuntar una expresión de inicialización a un campo de una estructura, estaríamos interfiriendo en el efecto del constructor implícito.

- Si definimos un constructor con parámetros, estamos obligados a inicializar *todos* los campos de la estructura antes de abandonarlo. En particular, esto también se exige para poder llamar métodos de la propia estructura dentro del código del constructor.

Esta regla existe para facilitar el trabajo del verificador de .NET. La siguiente declaración contiene un error que es detectado por el compilador:

```
// Esta estructura contiene errores de compilación
public struct Contador
{
    private int valor;

    public Contador(int v)
    {
        Limpiar(); // Esta llamada genera un error de compilación
        valor += v;
    }

    public void Limpiar()
    {
        valor = 0;
    }
    :
}
```

El conflicto está en el código del constructor: la llamada a *Limpiar* se ejecuta antes de que hayamos podido inicializar el campo *valor*. Es cierto que *Limpiar* se encarga de ello... pero el compilador no lo sabe, porque para verificar si se cumple la regla mencionada, sólo examina el código definido dentro del propio constructor.

- No se puede heredar de una estructura.

Dada la implementación actual del CLR, para soportar la herencia de estructuras habría que trabajar con punteros a estructuras en varias operaciones, y eso complicaría tanto la implementación en sí como la verificación del código.

NOTA

Eiffel ofrece un concepto más elegante para la declaración de tipos con semántica de valor: los tipos incrustados, o *embedded types*. Sospecho que la CLR complicará la implementación de este modelo, aunque no puedo jurarlo sin volver a echar un vistazo a la especificación de Eiffel.

Para terminar con las estructuras, aunque sólo de momento, debo mencionarle la existencia de una operación llamada *boxing*, y su inversa, *unboxing*; no me atrevo a traducirlas. El *boxing* consiste en crear una instancia en memoria dinámica para copiar en su interior un valor de una estructura, con la peculiaridad de que no se pierda la información del tipo de estructura original con la operación.

El *boxing* ocurre con frecuencia cuando asignamos una estructura a una variable o parámetro declarada de tipo **object**. Por ejemplo:

```
object obj = new Point(0, 0);
```

El *unboxing* deshace el hechizo sacando el conejo de la chistera. Después de la asignación anterior, podríamos hacer esto:

```
Point pt = (Point)obj;
```

Como ve, esta última operación es un caso particular de una vulgar conversión de tipo. Es en el *boxing* donde reside toda la magia.

¿Por qué demonios había que inventar una historia tan retorcida como esto de meter estructuras en cajitas para sacarlas después? Por una parte, se trata de un requisito lógico que simplifica extraordinariamente el diseño e implementación del lenguaje. Todos los tipos del *Common Type System*, ya sean tipos de referencia o con semántica de asignación por valor, se consideran derivados directos o indirectos de la clase *Object*. Se dice, por ejemplo, que todos los tipos de estructuras desciende de la clase especial *ValueType*. La herencia implica la compatibilidad para la asignación, y con el *boxing* tenemos la posibilidad de asignar cualquier tipo de valor a una variable de tipo **object**.

Un segundo motivo: el tipo **object** sustituye al tipo *Variant* que utilizaban lenguajes como Delphi y Visual Basic “nativo”. A primera vista, y también a segunda y tercera, el tipo variante parece una chapuza... pero lo cierto es que simplifica mucho la programación para bases de datos, y en particular, la capa de acceso genérico sobre la que se pueden construir capas más seguras desde el punto de vista del sistema de tipos.

Finalmente, están los problemas planteados por las estructuras de datos, en especial, los contenedores: listas dinámicas, pilas, colas, etc. En la versión 2.0, los tipos genéricos ofrecen la solución más elegante, segura y eficiente, pero estos tipos no estaban disponibles en las versiones anteriores. Para poder reutilizar una lista diseñada para almacenar “objetos”, en general, era necesario poder convertir tipos con semántica de valor en referencias.

Recogiendo la basura

Una característica común a Java y .NET es la existencia de un recolector de basura (*garbage collector*), con el propósito de automatizar la liberación de memoria. Los algoritmos básicos de este tipo existen desde hace muchísimo tiempo. En la década de los 80s se produjeron algunos avances importantes, como la técnica llamada *generation scavenging*, o barrido generacional, que mejoraron los tiempos de ejecución del algoritmo y lo hicieron más atractivo fuera de las áreas especializadas donde se utilizaba. A finales de los 80s y principios de los 90s, el lenguaje Eiffel popularizó la idea. Se trataba de un lenguaje orientado a objetos muy completo y elegante que incorporaba el manejo automático de la memoria con un recolector de basura. Tanto Java como los lenguajes de la plataforma .NET aprovechan las experiencias acumuladas en el diseño y uso de Eiffel.

Para ser sinceros, siempre he sentido recelos hacia este tipo de algoritmos... y sigo desconfiando. No obstante, es cierto que la recolección automática de basura es una jugada obligada en el diseño de .NET, y la justificación principal es la necesidad de contar con técnicas seguras y eficientes para el manejo de memoria en entornos distribuidos. Puede parecer extraña esta justificación, pero es completamente cierta. De modo que hay que acostumbrarse a que la basura se evapore misteriosamente ante nuestras narices, aunque a algunos no nos entusiasme excesivamente la idea.

Debo reconocer que la técnica de recolección de basura tiene cierto regusto Zen. Puede incluso que al obispo Berkeley le hubiese agradado. La idea se resume así:

Si nadie te ve, no existes

O más exactamente: si no existen referencias a cierto objeto desde un objeto vivo, el objeto en cuestión está muerto, y su memoria puede ser liberada. De vez en cuando, se disparará un hilo de ejecución que se llevará todas estas almas muertas a Neverland... ¿o era al Walhalla? Como puede

ver, todo se basa en definir qué es lo que está vivo y qué es lo que está muerto, y al parecer, la definición es recursiva. De modo que pongámonos formales:

- 1 Definamos un conjunto de objetos *vivos* iniciales, a los que llamaremos *raíces*.
 - Todos los objetos a los que se pueda acceder a través de un campo estático de una clase, se consideran *raíces*.
 - En un momento dado, congelemos la pila de llamadas de la aplicación junto con sus correspondientes zonas de variables locales. En ese momento, cualquier objeto apuntado desde cualquier variable local activa, se considera también una *raíz*.
- 2 Tomemos todas las referencias a otros objetos desde un objeto vivo: todos esos objetos referenciados estarán también *vivos*.

¿Es eficiente la recolección de basura de .NET? ¿Se nota cuando tiene lugar? No se preocupe: ya no estamos en los tiempos de los antiguos intérpretes de LISP y Prolog, y sobre todo, nuestros ordenadores no son ya los de aquella lejana época. He estado trabajando mucho tiempo en el desarrollo de un compilador de línea de comandos para un lenguaje compatible con .NET. Durante la ejecución del compilador se crean multitud de pequeños objetos estrechamente relacionados entre sí. La compilación de un proyecto de unos pocos miles de línea no necesita ni siquiera un segundo, y la mayor parte de este tiempo se consume en leer información sobre los ensamblados que se van a usar como referencia. En una etapa posterior del proyecto intentaremos disminuir ese tiempo probando un par de técnicas y optimizaciones.

Pues bien, durante ese par de décimas de segundos, el entorno de ejecución puede llegar a activar dos o tres veces la recolección de basura. Como entenderá, el tiempo de ejecución de este algoritmo no es notable, ni siquiera en este caso extremo.

Finalización y destrucción determinista

No obstante, la recolección automática de basura tiene un lado muy oscuro: nadie sabe cuándo va a tener lugar, y esto fastidia el uso de una de las técnicas más atractivas de la programación orientada a objetos. En palabras de Bjarne Stroustrup, el creador de C++:

Object creation is resource allocation

Traducido al romance: la creación de objetos es equivalente a la reserva de recursos. O con mayor claridad: podemos encapsular “recursos” dentro de clases, de modo que la creación de una instancia de estas clases implique la obtención de una instancia de estos “recursos”. En este contexto, “recurso” significa cualquier objeto o entidad potencialmente costosa, que exija de nosotros una operación explícita de petición pareada con una operación explícita de devolución. Si cuesta, debemos devolverlo, ¿no? Dentro de esta categoría entran entidades tales como las transacciones en un sistema de bases de datos, ficheros abiertos, semáforos del sistema operativo, la aprobación de uso de uno de estos semáforos, la mayoría de los objetos que se utilizan para dibujar en pantalla... y la lista podría estirarse. Como curiosidad, observe que la mayoría de estas entidades son en realidad instancias de objetos definidos en un nivel más bajo del sistema.

El problema nunca se presenta al pedir uno de estos recursos, sino a la hora de devolverlos. Sobre todo, porque debemos garantizar la devolución de los mismos. Si no cerramos los ficheros que abrimos, podemos perder la capacidad de abrir más ficheros. Si no terminamos correctamente las transacciones que iniciamos, podemos perder los cambios que hayamos hecho en el contexto de estas, además de que estaríamos interfiriendo en el trabajo de los restantes usuarios de la base de datos. Si seguimos los pasos de Stroustrup y asociamos la concesión de un recurso al estado de una instancia, es normal que asociemos la devolución del recurso a la destrucción de la instancia...

... y es aquí donde empezamos a tirarnos de los pelos, porque en un sistema con recolección automática de basura no podemos predecir cuándo tendrá lugar la siguiente limpieza de la memoria: será cuando el sistema lo estime oportuno. Puede que nunca. En cualquier caso, C# nos permite escribir código para que se ejecute cuando un objeto se convierta en basura y pase a mejor vida. La sintaxis necesaria recuerda el uso de destructores en C++:

```

public class ClaseConRecurso
{
    private Recurso recurso;

    ClaseConRecurso()
    {
        // Este es el constructor
    }

    ~ClaseConRecurso()
    {
        // Parece un destructor, pero es un «finalizador»
        if (recurso != null)
            recurso.Liberar();
    }
}

```

No se apresure a sacar conclusiones a partir de este fragmento de código; para empezar, hay que tener mucho cuidado con la clase *Recurso*. Si, por ejemplo, *Recurso* fuese una clase CLR como *FileStream*, ¡sería incorrecto, e incluso peligroso, usar la finalización! Luego explicaré por qué. De momento, tenemos dos problemas con la finalización. Ya hemos mencionado el primero de ellos: nunca sabremos en qué momento se va a ejecutar. El segundo es casi tan grave: cuando el colector de basura encuentra una instancia con código de finalización, la mueve a una cola de finalización, y utiliza un hilo separado para ejecutar el código de finalización de cada miembro de la cola. Esto puede llegar a ser muy costoso.

A la vista de todas estas dificultades, muchos optan por no aplicar la máxima de Stroustrup, y utilizar métodos “normales” para reservar recursos y liberarlos posteriormente. Pero se trata de una cobarde rendición: es muy fácil olvidar la llamada al método que devuelve el recurso. En todo caso, hemos complicado las cosas añadiendo el sistema de recolección de basura para automatizar la devolución de uno de los recursos más baratos: la memoria libre. Sin embargo, seguimos indefensos frente al problema de la gestión automática del resto de los tipos de recursos. Los demonios aparentemente exorcizados vuelven a asomar sus feos rostros.

No hay forma de evadirse de estos problemas. La plataforma .NET ofrece un remedio parcial: un patrón de uso conocido como destrucción determinista. En vez de dejar que el autor de cada clase se invente un nombre diferente para el método de liberación, .NET nos propone que lo llamemos siempre *Dispose*. Más aún, nos aconseja que avisemos que nuestra clase tiene recursos que deben ser liberados haciendo que la clase implemente el tipo de interfaz *IDisposable*:

```

public interface IDisposable
{
    void Dispose();
}

```

De modo que si encontramos una clase que implementa *IDisposable*, sabremos con certeza total que debemos garantizar la ejecución de su método *Dispose*. Ahora bien, incluso este requisito es demasiado para algunos programadores. El patrón de uso de una clase con destrucción determinista debería parecerse a esto:

```

ClaseConRecurso objeto = new ClaseConRecurso();
try
{
    // Aquí usamos la instancia creada
}
finally
{
    IDisposable d = objeto as IDisposable;
    if (d != null) d.Dispose();
}

```

La instrucción **try/finally** vigila la generación de excepciones; si se produce alguna mientras usamos la instancia, nos garantiza que se ejecute el código incluido en la cláusula **finally**. Por desgracia, hay muchos libros en los que el autor olvida esta simple regla de higiene. Estos pecados no reciben

su merecido castigo de forma inmediata. Tan solo sucede que terminamos con una aplicación poco robusta entre manos, a la espera de que una pequeño fallo se amplifique hasta convertirse en una verdadera catástrofe.

Para ahorrarnos líneas de programa, mejorar la legibilidad y hacer evidentes nuestras intenciones, los diseñadores de C# idearon la instrucción **using**:

```
using (ClaseConRecurso objeto = new ClaseConRecurso())
{
    // Aquí usamos la instancia creada
}
```

Estas cuatro líneas hacen el mismo trabajo que las diez líneas originales. Para no complicarme demasiado, he usado una instrucción **new** en la cabecera de la instrucción, pero es posible usar cualquier instrucción que devuelva un objeto perteneciente a una clase que soporte *IDisposable*:

```
using (SqlDataReader rd = sqlCommand.ExecuteReader())
    while (rd.Read())
        Console.WriteLine(rd[0]);
```

Como puede ver en estos sencillos ejemplos, una de las ventajas adicionales de la instrucción **using** es que hace coincidir el tiempo de vida del objeto asignado a la variable con el intervalo en el que la clase encapsula el recurso presuntamente valioso. Después de ejecutar *Dispose*, el objeto se convierte en un *zombie*, en un cascarón vacío sin alma. Sería arriesgado poder seguir usando la instancia después de que sus recursos se hayan liberado.

NOTA

El ejemplo que utiliza la clase *SqlDataReader* muestra una técnica de uso frecuente en el acceso a datos, especialmente en ASP.NET v1.1. A pesar de ello, los ejemplos equivalentes en la documentación oficial del SDK no utilizan una instrucción **using** para garantizar la liberación de recursos. Ni siquiera se utiliza **try/finally** o se advierte del problema que puede ocasionar una excepción fuera de control.

Precauciones en un mundo feliz

Conviene que resuma mi opinión sobre la recolección automática de basura:

- 1 Un sistema orientado a objetos que soporte objetos distribuidos debe implementar algún tipo de gestión automática de la memoria. COM/COM+ usaba contadores de referencia, mientras que .NET utiliza recolección de basura.
- 2 Es muy probable que algunos de los diseñadores de .NET estuviesen sinceramente entusiasmados con la perspectiva de usar *garbage collection*.
- 3 Creo, sin embargo, que esta técnica es un mal menor. Es cierto que resuelve el problema de la liberación de la memoria ocupada por instancias dinámicas, pero deja sin resolver el problema mucho más grave de la liberación de otros tipos de recursos valiosos.
- 4 Es incluso posible que las primeras aplicaciones para .NET que salgan al mercado vean resentida su estabilidad por los errores a los que puede inducir la falsa confianza que proporciona la recolección de basura.

Por supuesto, mi última afirmación necesita ser demostrada: la recolección automática de basura puede provocar graves meteduras de pata. Volvamos por un momento a uno de los ejemplos ya mostrados:

```
public class ClaseConRecurso
{
    private Recurso recurso;

    ClaseConRecurso()
    {
        // Este es el constructor
        // ...
    }
}
```

```

~ClaseConRecurso()
{
    // Este es el finalizador
    // ...
}

```

Cuando presenté esta clase, evité cuidadosamente dar demasiados detalles sobre el tipo *Recurso*. Identifiquemos temporalmente la misteriosa clase con una clase cualquiera del CLR, como *FileStream*.

```
private FileStream recurso;
```

Pregunta capciosa: ¿es correcto hacer lo que muestra el siguiente finalizador?

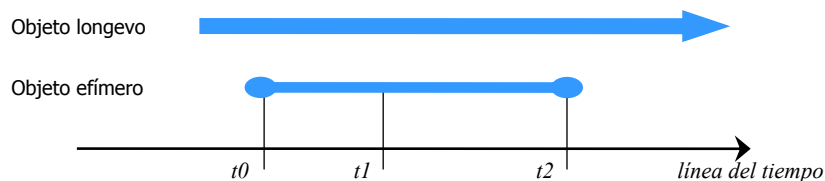
```

~ClaseConRecurso()
{
    if (recurso != null)
        recurso.Dispose();
}

```

Respuesta: ¡es un peligroso disparate! Razonemos: ¿por qué se está ejecutando el finalizador? Resulta que el objeto de tipo *ClaseConRecurso* ha muerto porque nadie lo miraba. Si nadie lo miraba, ¿habría alguien mirando al objeto enganchado a *recurso*? Lo más probable es que no, porque se trata de un campo oculto dentro de la clase. Entonces, el único objeto que mira al objeto enganchado en *recurso*, ¡es un cadáver! (sí, es tétrico). Para lo que viene a continuación es importante el hecho de que *FileStream* es una clase perteneciente a .NET. Esto significa que *FileStream* sigue los convenios de la plataforma, y tiene un finalizador para cuando alguien olvide cerrar el fichero antes de perderlo de vista. Por lo tanto, el fichero *recurso* está también en la cola de finalización del colector de basura. ¿Cuál objeto exprimirá primero el colector, al de tipo *ClaseConRecurso* o al fichero? ¡Es imposible saberlo! Es más, es probable que la referencia al fichero siga siendo válida, pero esto es algo que depende de la implementación concreta del colector de basura: después de una pasada del colector, se compacta la memoria dinámica. ¡Ni siquiera se nos garantiza que el fichero siga ahí!

Es cierto que, descontando este problema con los finalizadores, la recolección de basura hace imposible que utilicemos objetos ya liberados. Como compensación, el nuevo problema más frecuente consiste en que objetos innecesarios sean mantenidos vivos de forma artificial, y sin que seamos conscientes de ello. Una forma muy sencilla de provocar este error guarda relación con los manejadores de eventos. Para provocar el problema necesitamos un objeto de larga duración, o incluso una clase con estado estático como *Application*. Este objeto lanza ciertas notificaciones mediante un evento. En el caso de *Application*, este evento podría ser *Idle* o incluso *ThreadException*. Un objeto con tiempo de vida más corto se interesa por estas notificaciones y añade un manejador a la correspondiente cadena de delegados.



El diagrama muestra el solapamiento de los tiempos de vida de los dos objetos involucrados. Hay tres puntos marcados en la línea del tiempo. En el primero, se crea el objeto efímero. En el segundo instante, el objeto efímero se suscribe a un evento disparado por el objeto longevo. Por último, el programador cumple con sus deberes y ejecuta, directa o indirectamente, el método *Dispose* del objeto temporal, para que muera. Pero ha olvidado desconectar el manejador de eventos. Como el delegado contiene un puntero oculto al objeto manejador, el objeto que debía haber muerto sigue estando en el directorio telefónico del objeto de larga duración. Si, por desgracia, este dispara el evento, tendremos problemas con casi total seguridad, porque al llamar a *Dispose* el objeto se ha convertido en un cascarón vacío.

¿Le sabe a poco?

Comencé esta introducción hablando de elefantes, y voy a terminar también con elefantes. Le confesaré algo: me siento como un rottweiler hambriento al que le sirven un elefante asado. Mi instinto asesino me impele a morder con los ojos cerrados... pero correría el riesgo de abrir un túnel en el asado y que me cayese encima todo el peso del paquidermo. La foto de la izquierda, tomada en broma hace casi un año, muestra una pequeña porción de mi actual colección de libros sobre .NET (con algún intruso a la izquierda). Es inevitable que toda introducción se quede corta... pero esta no es la última oportunidad para, si aún no lo sabe, enterarse qué maldita utilidad tiene, digamos, una propiedad estática o un constructor de instancias privado.



A pesar de la pila de libros que muestra la foto, y la otra pila que ha crecido desde entonces, hay muy pocos libros en ella sobre el lenguaje “en sí”. Mi consejo, si quiere estudiar a fondo el lenguaje es que utilice primero la ayuda en línea que viene con el SDK de .NET. La ayuda contiene una referencia muy buena y bastante completa. Si al terminar con ella quiere “más”, puede echar un vistazo a estos libros:

Inside C# (second edition)

Tom Archer, Andrew Whitechapel
ISBN 0-7356-1648-5

Programming in the Key of C#

Charles Petzold
ISBN 0-7356-1800-3

The C# Programming Language

Anders Hejlsberg, Scott Wiltamuth, Peter Golde
ISBN 0-321-15491-6

The Common Language Infrastructure Annotated Standard

James S. Miller, Susann Ragsdale
ISBN 0-321-15493-2

Los dos primeros son libros de introducción al lenguaje. De ellos, prefiero el primero, porque encuentro un poco ñoño el segundo. Los dos últimos libros son en realidad referencias. No son libros para novatos, pero recuerde que nadie es novato toda la vida. El último de ellos trata sobre el entorno de ejecución, y contiene información que no encontrará en ningún otro lugar.