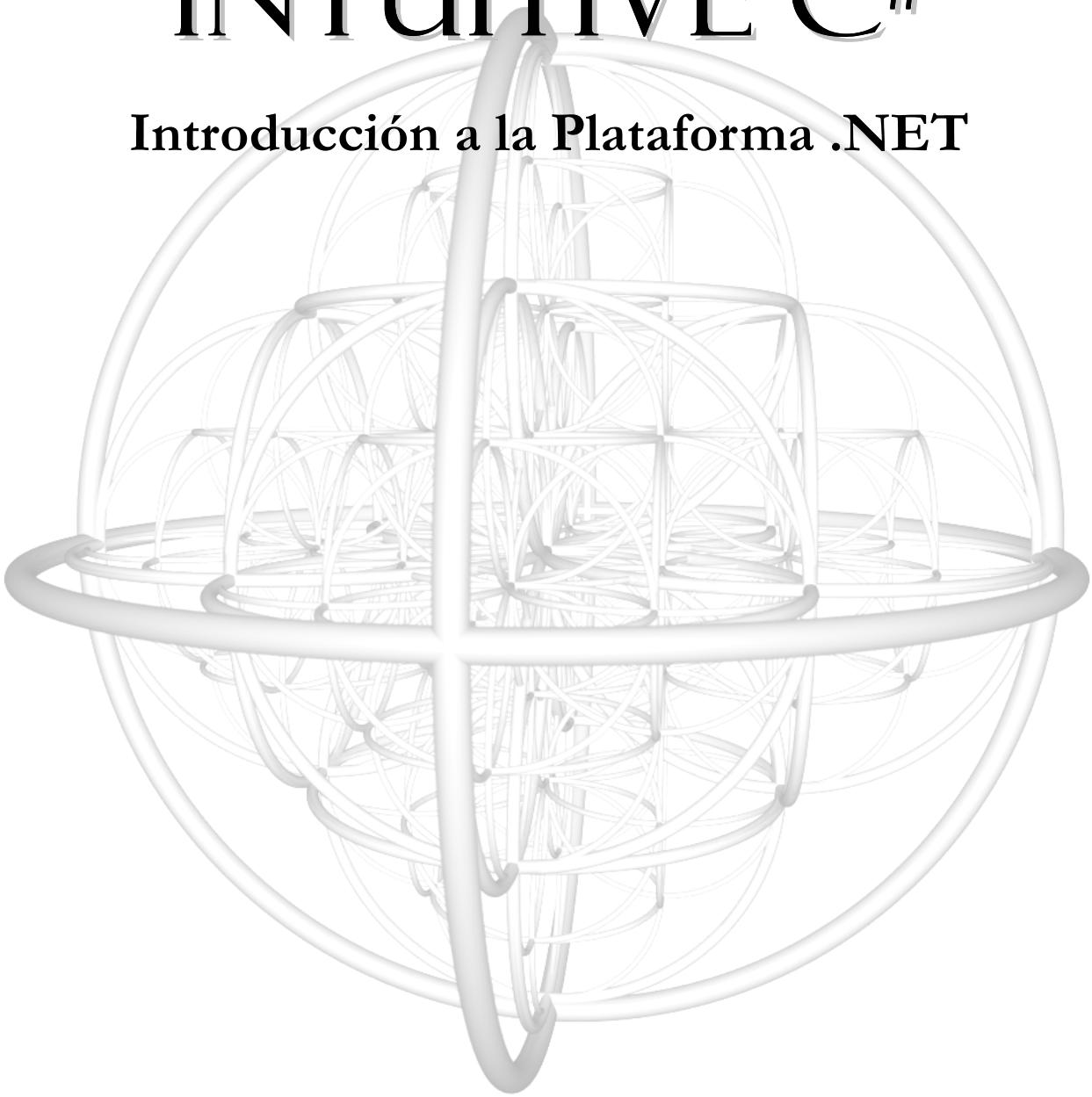
INTUITIVE C#

Introducción a la Plataforma .NET



Ian Marteens

2008

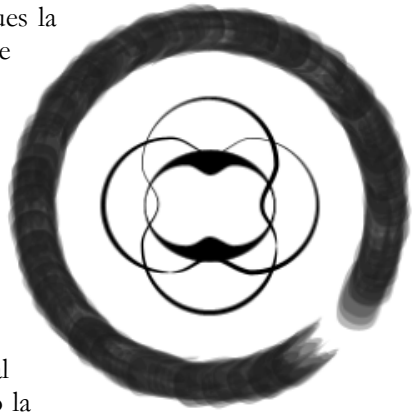
INDICE DE MATERIAS

PROLOGO

A dream is an answer to a question
We haven't learned how to ask.
Dana Scully

¿CUÁNTAS CONSTELACIONES PUEDE usted identificar en el cielo nocturno? Si intenta aprenderlas una a una, le costará mucho. Lo mejor es hilar una historia, e ir saltando de constelación en constelación. Por ejemplo, puede empezar por Orión, que es la más fácil de todas. A partir de Orión, localice la estrella Sirio, con lo que habrá dado con el Can Mayor; en la dirección opuesta estará esperándole el Toro, con las Pléyades en su frente.

Ouroburos



Aprender un lenguaje de programación no es tan diferente, pues la dificultad es similar: los temas están vinculados por multitud de enlaces circulares. Esto no es muy común. Resulta fácil, por ejemplo, escribir un libro o un curso sobre programación para bases de datos con ADO.NET. Es fácil localizar la hebra y tirar de ella, organizando la materia en una línea recta. Pero tratándose de aprender el propio lenguaje, te encuentras con que no puedes enseñar qué es un método virtual antes de mostrar qué es la herencia… y viceversa.

Creo que la solución consiste en renunciar a enseñarlo todo de golpe, y en hilar "historias" que recorran franjas del material como si se tratase de la franja del zodiaco, la línea ecuatorial o la mismísima Vía Láctea. Eso, al menos, es lo que he intentado hacer en este folleto.

Estas notas se han desprendido de una introducción a la plataforma .NET que inicialmente formaba parte de un curso sobre ADO.NET. Tratándose de un trabajo en marcha, que tendré que ampliar y corregir unas cuantas veces en el futuro cercano, no he intentado disimular el origen de la mayoría de las partes.

Estado actual del libro

Este no es un libro terminado y revisado según dictan las normas. Es un experimento, un libro cuya escritura tiene lugar a la vista de todos. Es inevitable que, mientras no termine su redacción, aparezcan saltos, inconsistencias e incluso algún que otro error en el contenido.

La presente versión (14/dic/2006) divide su contenido en capítulos por primera vez. Antes, el libro estaba organizado como un capítulo gigantesco. Esta es una lista de problemas y expectativas:

1. Habrá un capítulo entre el primero y el segundo, titulado *Programación orientada a objetos*.
2. El segundo capítulo de ahora, titulado *Clases y objetos*, cederá la mayor parte de su contenido a un nuevo capítulo titulado *Tipos de interfaz*.
3. Tengo dudas sobre mantener capítulos separados para *Programación orientada a objetos*, que contendría una explicación de las técnicas relacionadas con el clásico trío encapsulamiento/herencia/polimorfismo, y otro capítulo, Clases y objetos, para detalles concretos relacionados con C#. Ya veremos…
4. El material del capítulo *Estructuras* es nuevo casi por completo. Tengo que decidir dónde debe ir la sección sobre *clases parciales* (probablemente, mucho antes). Donde vayan a parar las clases parciales será donde se mencionen las clases estáticas de C# 2.0.
5. El material del capítulo *Reflexión* es también nuevo. Le hace falta una buena limpieza.
6. El capítulo *Iteración* es un poco extraño. Me he arriesgado a tratar ahí los métodos anónimos, y a mencionar de paso las expresiones lambda. No es una elección arbitraria: el otro sitio donde

podrían acomodarse los métodos anónimos sería junto a los delegados, pero me pareció prematura. Teniendo en cuenta que los métodos anónimos pueden verse como una extensión de las estructuras de control del lenguaje, al acercar el código al sitio desde dónde se utiliza, no creo que sea tan grave mi elección. Pero ya veremos…

7. Sobre todo, falta mucho contenido, respecto a mis planes. Le pido un poco de paciencia.

Creo que no necesito decir lo siguiente: por favor, cualquier error que detecte, ¡dígamelo! En esta etapa no es tan importante localizar las faltas de ortografía y redacción, pero avíseme sobre las que encuentre. Si no, se corre el riesgo de que queden ocultas en cuanto crezca el libro.

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 complejos 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 …
    }
}
```

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 usado negritas para las palabras reservadas, 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.

Quiero agradecer, sobre todo, la fe y la paciencia de todas las personas que han adquirido las distintas series del curso de ADO.NET por adelantado. Sólo espero no defraudarlas, y que estas notas les sean de algún provecho.

Ian Marteens
www.marteens.com
www.commanet.net
Madrid, diciembre de 2006

1

COMMON LANGUAGE RUNTIME

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…!

¿Qué es la plataforma .NET?

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.

SEÑALADOR

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 pro-

grama. 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 Freya[1]. 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

[1] freyalang.blogspot.com

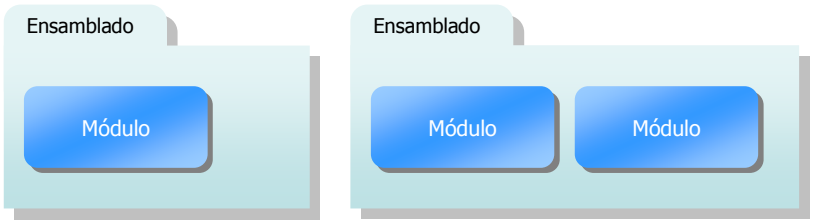
compila dos bibliotecas dinámicas, como las DLL de Windows nativo, que hacen uso de *Complex*. Cada biblioteca cargaría con su propia copia de la clase, y si una aplicación necesita usar ambas bibliotecas dinámicas, en el espacio del proceso ejecutable tendríamos también dos copias independientes de 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 autentificación, 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

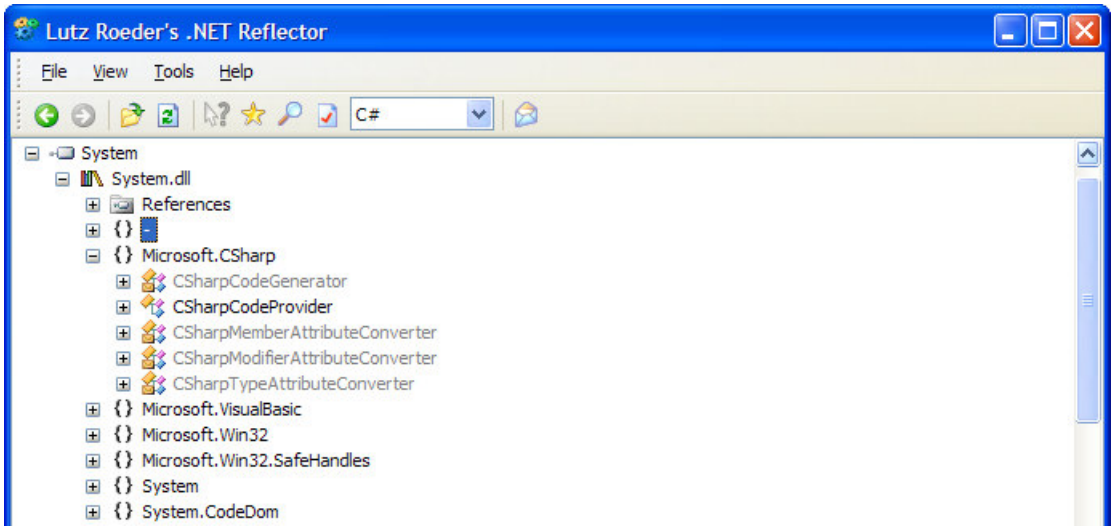
El nombre abreviado del tipo es *SqlDataAdapter*. El tipo se encuentra definido dentro de un espacio de nombres llamado *SqlClient*, definido dentro de *Data*, que finalmente se define dentro del espacio de nombres de primer nivel llamado *System*.

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 4.2.48.0, y funciona tanto con la versión 2.0 de la plataforma como con las versiones anteriores. Aquí tiene una imagen de esta aplicación:



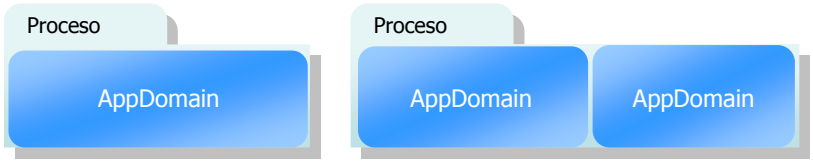
El nodo superior del árbol corresponde al ensamblado *System*. Inmediatamente debajo aparece el módulo correspondiente al fichero *system.dll*. El módulo hace referencia a otros ensamblados, que no se muestran en la imagen, pero define una serie de clases que la aplicación reparte entre los

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.

Procesos y dominios de aplicación

El modelo de ejecución tradicional de Windows consiste en procesos mutuamente aislados, con un conjunto de uno o más hilos (*threads*) por proceso. Cada hilo pertenece a un único proceso desde que se crea hasta que se destruye. Observe que he aclarado que se trata del modelo de Windows: otros sistemas operativos tienen modelos diferentes. El de Windows es bastante bueno y eficiente: la memoria se asocia al proceso, no al hilo, y es fácil compartir información entre hilos de un mismo proceso, pues todos tienen acceso a la memoria del proceso. Por supuesto, tendremos que preocuparnos por la sincronización del acceso a dicha memoria, pero la sincronización es inevitable más tarde o temprano, con independencia del sistema operativo.

No obstante, cuando ejecutamos .NET sobre Windows, manejamos un nuevo concepto: el de *dominio de aplicación*, o *application domain* (*AppDomain*):



Como sugiere el diagrama, cada proceso puede contener uno o más dominios de aplicación. ¿Se ha dado cuenta de que no he incluido hilos en el diagrama? Por una parte, procesos y dominios de aplicación tienen algo importante en común: identifican una zona de memoria. La memoria de cada proceso se reparte entre sus dominios de aplicación, y cada zona es inaccesible para las demás.

No existe, sin embargo, una relación tan clara entre hilos y dominios de aplicación. Los hilos de cada proceso pueden ejecutarse dentro de cualquiera de los dominios, aunque en cada momento, un hilo sólo puede residir dentro de un único dominio. ¿El motivo? Pues que los dominios puedan compartir hilos de una caché común. Cada proceso en .NET dispone de una caché propia de hilos, que es utilizada por varios sistemas del CLR, como la ejecución asíncrona mediante delegados.

Los dominios de aplicación son, a la vez, una oportunidad y una necesidad. Lo de "oportunidad" tiene que ver con la curiosa forma en que se implementan:

- Las aplicaciones y bibliotecas en .NET deben aprobar una *verificación* antes de poder ser cargados y ejecutados. Esta verificación garantiza que el código que la aprueba nunca invadirá memoria ajena. Por lo tanto, para que dos dominios de aplicación mantengan su independencia hay que hacer… nada, o casi nada.

.NET tiene buenas razones para echar mano de este recurso:

- La principal razón se llama ASP.NET. Un servidor Web necesita montones de "procesos" funcionando en paralelo para atender a las distintas páginas y aplicaciones. Si hablamos de "procesos" verdaderos, el coste en recursos sería enorme, pero si se trata de dominios de aplicación, todo encaja.
- Un motivo relacionado: ASP.NET necesita descargar módulos y volver a cargarlos si detecta cambios en el código fuente asociado. Una vez que se carga una DLL en un proceso clásico, es muy difícil descargarla, por las referencias implícitas al código que pueden estar ocultas por aquí y por allá. Sin embargo, si la DLL se ha cargado dentro de un dominio de aplicación, es muy fácil descargarla… descargando todo el dominio de aplicación. Este es el mecanismo oficial en .NET para la descarga dinámica de bibliotecas de funciones.

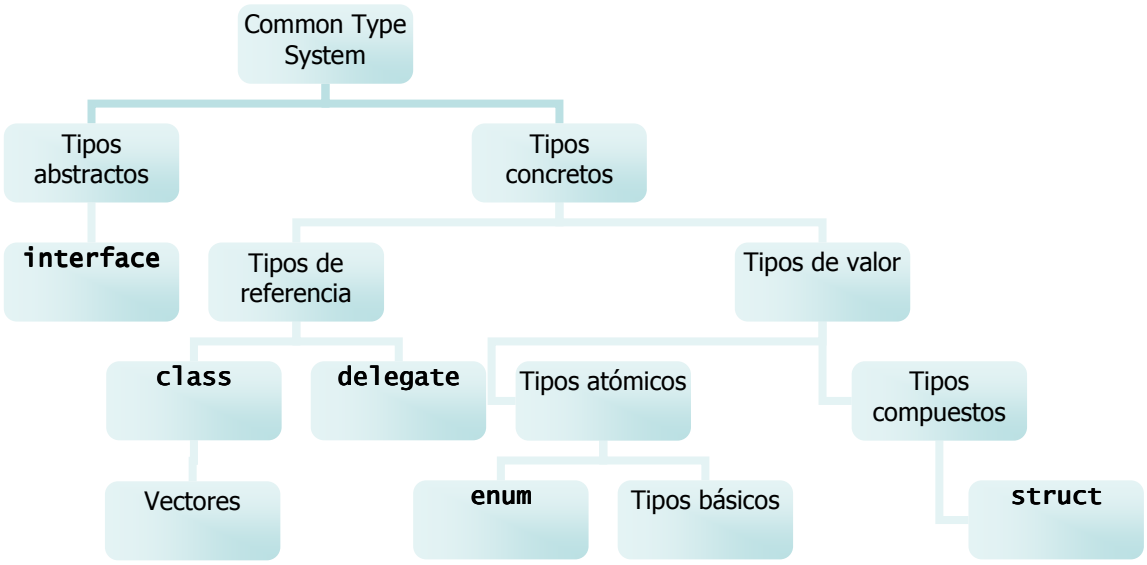
Incluso si no va a usar directamente ASP.NET, le interesa conocer el uso de este recurso. Las aplicaciones de gestión, por ejemplo, necesitan un grado de dinamismo que no puede, o no debe lograrse, mediante el clásico ciclo de desarrollo, compilación e implantación. Si un comercio decide usar una

nueva forma de pago, un sistema de transporte con tarifas especiales o una promoción con descuentos a la medida, la solución no debe ser volver a sentarse frente al ordenador y reprogramar los módulos afectados. En muchos casos, se utiliza algún tipo de intérprete o de generador de código para dar respuesta a este tipo de necesidades. Con .NET, es muy sencillo y eficiente generar código intermedio sobre la marcha. Normalmente, no es necesario hacer nada especial para descargar el código compilado, pero en determinados casos sí será un requisito. En tales casos, lo más indicado es cargar el código compilado en dominios de aplicación dentro del mismo proceso, para poder descargar luego el código de manera indolora.

> **NOTA** No obstante, como los dominios no pueden comunicarse directamente entre sí, hay que usar algún tipo de comunicación remota para poder manejar a distancia los objetos creados en otro dominio.

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 averiguar 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	*System.Boolean*	Verdadero y falso, ¿no es así?
char	*System.Char*	Caracteres Unicode (¡16 bits cada uno!)
sbyte	*System.SByte*	Enteros de 8 bits con signo.
byte	*System.Byte*	Enteros de 8 bits sin signo.
short	*System.Int16*	Enteros de 16 bits con signo.
ushort	*System.UInt16*	Enteros de 16 bits sin signo.
int	*System.Int32*	Enteros de 32 bits con signo.
uint	*System.UInt32*	Enteros de 32 bits sin signo.
long	*System.Int64*	Enteros de 64 bits con signo.
ulong	*System.UInt64*	Enteros de 64 bits sin signo.
float	*System.Single*	Valores reales representados en 4 bytes.
double	*System.Double*	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, guita[2]… o como quiera llamarlo. Lo notable respecto a estos dos señores es que tienen nombres especiales en C#:

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

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. A partir de la versión 2.0, los tipos genéricos hacen innecesarias la mayoría de las aplicaciones del *boxing*.

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

CLR	Significado
System.DateTime	Fecha y hora.
System.Guid	Corresponde al tipo **uniqueidentifier** de SQL Server.
System.DBNull	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.

Vectores

Un lenguaje, como cualquier teoría, es más elegante mientras menos casos especiales presente. Esta es una fuerza centrípeta, que tiende a reducir el tamaño y complejidad del lenguaje. Se equilibra con la correspondiente fuerza centrífuga: la simplificación no debe afectar negativamente ni a la facilidad de uso, ni a la eficiencia de la implementación.

El caso de los *vectores* (*arrays*) es un ejemplo clásico del conflicto entre ambas tendencias. Nos gustaría que un vector fuese una clase más… y hasta cierto punto se logra. Pero una característica de ellos nos amarga la golosina: no existen vectores a secas. Más bien, existen vectores de enteros, vectores de cadenas, vectores de personas y vectores de abogados. Técnicamente hablando, tenemos una operación de construcción de tipos, que a partir de un tipo *T*, nos ofrece un nuevo tipo *T'*, o si utilizamos la notación de C#, el nuevo tipo *T*`[]`.

NOTA En C# 1.0, este "constructor de tipos" es único en su género. Sin embargo, con la llegada de los tipos genéricos en la versión 2.0, los vectores podrían considerarse como un caso especial de tipo genérico. Lamentablemente, ni C# ni la propia plataforma los tratan de esta manera: los vectores siguen siendo tipos especiales. En Freya, en contraste, los vectores sí se consideran un caso más de tipo genérico, aunque la implementación luego no los trate así.

Si comparamos C# con C/C++, encontraremos una diferencia importante en la forma en que se declara un vector. Tome nota:

```
// Esto sería en C++
int vector[];

// Esto es C#
int[] vector;
```

Los autores de C defendían una extraña teoría sobre las declaraciones de variables: la declaración debía prefigurar la forma en que luego se usaría la variable. Como resultado, cuando se declara una

[2] Sólo las cosas *muy* importantes tienen tantos nombres.

variable de vector en C/C++, el tipo base del vector queda a la izquierda de la variable y los corchetes a la izquierda: no se trata al vector como a un tipo de datos más. En C#, y antes en Java, las declaraciones de vectores son más sensatas, y los corchetes se asocian al tipo, no al identificador que se está declarando. Ahondando en esta diferencia, la siguiente declaración en C++ no sería correcta en C# (excepto en el modo de programación no seguro, en C# 2.0):

```
// Esto es C++. En C# no está permitido.
int vector[7];
```

Para declarar e inicializar un vector en C# valen estas alternativas:

```
int[] vector1 = new int[7] { 6, 5, 4, 3, 2, 1, 0 };
int[] vector2 = new int[] { 11, 22, 33, 44, 55, 66, 77 };
int[] vector3 = { 111, 222, 333, 444, 555, 666, 777 };
```

A pesar de la sintaxis especial, las instancias de vectores son objetos, a todos los efectos. Aparte de obtener sus elementos individuales con la ayuda de corchetes y una expresión numérica, podemos aplicar métodos y propiedades sobre la instancia. Observe cómo se utiliza la propiedad *Length* en el siguiente ejemplo:

```
string[] dias = { "Lunes", "Miércoles", "Viernes" };
for (int i = 0; i < dias.Length; i++)
    dias[i] = dias[i].ToLower();
```

En el bucle anterior, recorremos y modificamos el vector, pero si sólo necesitamos recorrer sus elementos, podemos usar la instrucción **foreach**:

```
foreach (string s in dias)
    Console.WriteLine(s);
```

Internamente, el *Common Language Runtime* considera que una instancia de vector pertenece a una clase especial derivada de la clase *System.Array*. Es importante saberlo porque podemos usar los miembros definidos para la clase *Array* con vectores del tipo que sean. Muchos de los métodos que ofrece *Array*, sin embargo, son métodos estáticos. El método estático *Resize*, por ejemplo, nos permite cambiar la cantidad de elementos en una instancia de vector:

```
string[] dias = { "Lunes", "Miércoles", "Viernes" };
Array.Resize(ref dias, dias.Length + 1);
dias[dias.Length - 1] = "Sábado";
```

La palabra clave **ref** que aparece en el primer parámetro de la llamada a *Resize*, indica que se trata de un parámetro pasado por referencia, y que el método puede cambiar el contenido de la variable.

2

PROGRAMACIÓN ORIENTADA A OBJETOS

LOS LENGUAJES DE PROGRAMACIÓN ORIENTADOS a objetos llevan ya mucho tiempo con nosotros. Cuesta imaginar que exista todavía algún programador que no tenga unas nociones mínimas sobre qué significa la famosa "orientación a objetos". Pero qué tenga plena consciencia sobre qué es lo que hace buena esta metodología es otro asunto. Esto hace posible que, de cuando en cuando, algún iluminado se disfrace de profeta y afirme haber superado la tontería de los objetos… e incluso la necesidad de la verificación estática de tipos.

La Programación Orientada a Objetos tiene sus limitaciones y problemas, por supuesto. Existen propuestas que la complementan, o que corrigen algunos de sus problemas. Es también posible que un día descubramos que existe una forma mejor de programar, y que la metodología se diferenciaba tanto de la P.O.O. que era casi imposible dar con ella mediante pequeñas desviaciones de lo que ahora se considera correcto. Pero hay que ser prudentes. También llegará un día, quizás no muy lejano, en el que la Teoría de la Relatividad de Einstein sea sustituida por una teoría mejor. La nueva teoría, sin embargo, tendrá que dar respuesta a todo lo que la Relatividad ya explica. Y en todo caso, es poco probable que sea el cajero del supermercado quien dé con ella. Aunque es posible…

NOTA

Este capítulo es *opcional*. Si lleva usted tiempo programando, no sólo le advierto que puede saltárselo; además, ¡se lo aconsejo!, al menos, en una primera lectura, pues todo lo que narro aquí ya lo habrá leído en más de una ocasión. Si decide seguir, tengo para usted una segunda advertencia: lo que explico aquí no es exactamente una historia de las ideas, sino una posible explicación de las relaciones entre las mismas… porque lo que intento es hacer más asequibles y comprensibles esas ideas. Me ha interesado más, por ejemplo, contar cómo una pequeña mutación separa las unidades de Delphi y los paquetes de Ada que relatar la historia de Jean Ichbiah y el diseño de Ada.

Abstracción Orientada a Objetos

Antes de la aparición de los lenguajes orientados a objetos, los lenguajes imperativos tradicionales se enfrentaban a una elección difícil por culpa de los sistemas de tipos de aquel entonces. Un lenguaje podía implementar la verificación estática de tipos con todas sus consecuencias. Lo malo es que, entre esas consecuencias, estaba la pérdida de expresividad y flexibilidad. Por el contrario, podía relajar las verificaciones aquí y allá, y convertir al lenguaje en un coladero de errores de programación, o incluso renunciar a la verificación estática, sustituyéndola por verificaciones en tiempo de ejecución. ¿Resultado? Lentitud y falta de seguridad en la programación.

El otro problema importante anterior a la Programación Orientada a Objetos era la dificultad para reutilizar código ya existente. Los módulos reutilizables de por entonces eran simples bibliotecas de funciones. ¿Cómo se puede montar un sistema a partir de un conjunto de funciones? Si éstas son lo suficientemente flexibles, podemos agrupar varias de ellas… siempre que escribamos el código a la medida que actúe como pegamento.

Digamos, por concretar, que queremos programar un *ray tracer*. Si no existiese la Programación Orientada a Objetos, nuestro punto de partida, con la suerte a nuestro lado, sería una biblioteca de funciones diseñada para trabajar con vectores, matrices y colores. Y poco más. Es cierto que existen descripciones muy precisas de los algoritmos utilizados por un *ray tracer*. Pero el tipo de abstracción necesaria para materializar estos algoritmos en código reutilizable se expresaba con dificultad en aquellos lejanos tiempos.

Para explicarlo, usaré un ejemplo más sencillo: imagine que necesitamos una rutina que ordene un vector con elementos de tipo arbitrario. Si elegimos el algoritmo *quicksort*, la rutina debe comportarse de esta manera:

1. Debemos elegir un elemento arbitrario del vector.
2. Movemos los elementos del vector de manera que todos los elementos iguales o menores que el elemento antes elegido queden a la izquierda de los elementos mayores. Para ello:
   - Usamos un par de punteros, inicialmente al principio y al final del vector a ordenar.
   - **Comparamos** los dos elementos referidos por los punteros mencionados y…

… un momento, ¿qué quiere decir *comparar*? Puede significar diferentes algoritmos, por lo que deberíamos buscar un modo de pasar la formar de comparación como un parámetro más de la rutina. ¿Se puede pasar un algoritmo como parámetro en la programación estructurada? ¡Claro que se puede! Eso es lo que se ha conseguido desde siempre con los punteros a funciones. ¿Por qué he dicho entonces lo de *"se expresaba con dificultad"*? La respuesta es sencilla: si nos ponemos quisquillosos, ¡no hay algoritmo que no pueda programarse en lenguaje ensamblador!

El que tal programa sea comprensible y fácil de mantener es un asunto muy diferente.

Encapsulamiento: datos

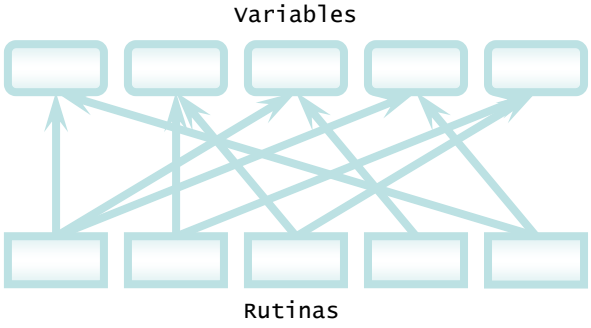
En programación, hay que tener mucho cuidado con la frase *"esto no vale"*. Hay muchos criterios para valorar la calidad del código, y no existe hasta el momento un procedimiento mecánico al que le suministremos código fuente y responda *"esto es una porquería"* o *"usted es un genio, amigo"*. Quizás sea bueno que así sea. ¿Imagina la cantidad de personas que podían quedarse sin empleo?

¿Por qué es difícil de mantener un programa que pasa punteros a funciones a diestra y siniestra? La culpa no es, en su mayor parte, de que los punteros a funciones sean tipos muy especiales, aunque también influya esta singularidad. La culpa, en realidad, es la falta de *abstracciones* adecuadas en el estilo de programación antes esbozado. La rutina de ordenación es muy simple como ejemplo, por lo que utilizaremos algo más complicado.

Una de mis primeras aplicaciones "grandes", cuando estudiante, fue un editor de texto. Tenía dos compiladores a mi disposición: el Turbo Pascal 3.0 y el compilador de C de Microsoft. Como ve, nada de programación orientada a objetos. Como todavía no se habían puesto de moda los sistemas de ventanas, mi editor era una aplicación que editaba un único fichero por vez. Las variables que controlaban el estado del editor eran, precisamente, *variables*: un buen puñado de variables globales al alcance de cada una de las rutinas. No conservo el código fuente de aquel desastre, pero podemos hacernos una idea:

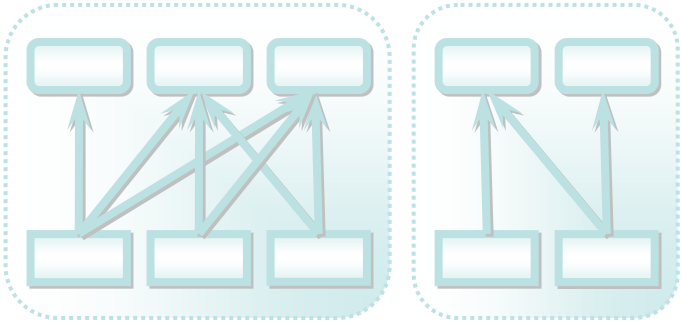
```
var
    DesplHorizontal, DesplVertical: Integer;
    CursorX, CursorY: Integer;
    NombreFichero: String;
    GuardarCopia: Boolean;
```

Me va a perdonar que omita las correspondientes rutinas que utilizaban todas estas variables, pero creo que este diagrama le permitirá hacerse una idea:



He intentado dibujar el equivalente de una orgía dionisíaca entre rutinas y variables. Es imposible saber de quién es cada mano o cada pie… y es muy probable que alguna rutina termine tocando algo que no le interesaba tocar. Si hay M variables y N rutinas, el total de dependencias entre rutinas y variables es, naturalmente, *MxN* enlaces. Y la realidad suele ser aún peor, pues no he hablado de las dependencias entre las propias rutinas. Como resultado, cada vez que haya que hacer un cambio en este sistema, habrá que revisar todas las rutinas para averiguar cuáles variables y rutinas se verían afectadas.

Pero esta es una película que han pasado muchas veces por televisión. A no ser que esté usted empezando en este negocio, ya tendrá usted la respuesta: para evitar esta maraña de dependencias, se utiliza la *descomposición modular*. Se trata de dividir el sistema en partes más pequeñas, de manera que entre estas partes exista la menor comunicación posible. Es decir: debe tratarse de partes lo más independientes que sea posible. Incluso si sólo logramos dividir el sistema en dos, ya habremos conseguido una importante ventaja:



Como cada parte tiene ahora la mitad de elementos que antes, el número total de flechas se reduce a la mitad (¡compruébelo!). Recuerde, no obstante, que la reducción de complejidad se consigue cuando no existe prácticamente comunicación entre los módulos en los que hayamos dividido el sistema. Y para ello hacen falta recursos en el lenguaje que no tenían los primitivos Pascal y C. Por ejemplo, una manera de organizar un poco el código de mi editor consistía en agrupar las variables globales dentro de *records*, en Pascal, o *structs*, en C.

```
type
    Posicion = record
        X, Y: Integer;
    end;

var
    Desplazamiento: Posicion;
    Cursor: Posicion;
```

Esto sólo nos sirve para hacernos una idea sobre cuáles grupos de variables pueden modificarse en equipo, pero no nos ofrece más garantías. Sigue siendo una orgía jamonera, pero ahora hay marcas de pintura en el pavimento, como en un aparcamiento.

Unidades, paquetes y otros triunfos parciales

Naturalmente, el mundo no contuvo el aliento hasta el momento en que Bjarne Stroustrup desenterró el mapa del tesoro que unos noruegos habían escondido a finales de los 60s en un fiordo escandinavo: nos hubiésemos asfixiado de intentarlo. Estas ideas sobre descomposición modular son casi axiomas del sentido común, y estaban dando vueltas en el imaginario colectivo desde los mismísimos orígenes de esta disciplina. Ahí están, como demostración, antiguas modas como las unidades de Pascal, los módulos de Modula y los paquetes de Ada.

De estos tres sistemas, el más sencillo y fácil de explicar son las unidades de Pascal, que aparecieron por primera vez en UCSD Pascal, y que inspiraron las unidades que Borland añadiría un poco más tarde a su Turbo Pascal. La idea es extremadamente sencilla: cada fichero compilable se divide en dos zonas. La primera, encabezada por la palabra clave **interface**, contiene declaraciones públicas.

Todas las declaraciones de la segunda sección, encabezada por **implementation**, son internas al fichero o unidad.

Para ser honestos, casi todos los lenguajes que permitían *compilación por separado*, es decir, todos los lenguajes *decentes* de aquella época, ofrecían un mecanismo similar… aunque por cortesía del enlazador, casi siempre. En C, por ejemplo, a no ser que una rutina o una variable se declarase explícitamente con el modificador **extern**, se consideraba un recurso privado, invisible fuera del fichero compilado. ¿Por qué no he mencionado el sistema de C para ocultar información? La respuesta es que C y su sistema de compilación por separado es una de las tres grandes plagas que ha sufrido la Informática[3]. Aunque con disciplina se puede domar la bestia, es muy fácil hacer un uso equivocado, e incluso peligroso, de la técnica utilizada por C.

La multiplicación de los panes y los peces

Mi editor tenía otro problema, un poco más sutil, pero que al final sería su condena: sólo podía editar un fichero por vez. Tratándose de la época en que el prácticamente el único sistema operativo disponible para un PC era MSDOS, se trataba de una limitación importante.

Observe que hay dos causas relacionadas, pero diferentes, que provocan este problema:

1. Si el estado de un editor se mantiene en un conjunto grande de variables globales, para mantener dos editores con sus correspondientes estados, es necesario duplicar este conjunto de variables.
2. Pero incluso si hubiese la suficiente voluntad para declarar dos juegos de variables, o tres, o un número variable de ellas, nos quedaría el problema de que la mayoría de las rutinas utilizaban directamente las variables globales.

En este sentido, los paquetes (*packages*) de Ada ayudaban más que las unidades de Pascal. Estos paquetes se inspiraban en la teoría de los *tipos de datos abstractos*, o *ADTs*. Un paquete, en sí, no era muy diferente de una unidad Pascal, pero allanaba el camino hacia la implementación de tipos de datos opacos. ¿Cuál es la diferencia? En Pascal podías agrupar en un registro las variables que controlaban el estado del editor, y programar, por separado, una lista de operaciones que actuasen sobre este tipo de registros. Pero si el registro se declaraba, como era necesario, en la sección de interfaz de la unidad, ¡todo el contenido del registro seguía estando al alcance de cualquiera! No había forma de evitar que alguien puentease las operaciones programadas y metiese las zarpas directamente en los campos del registro.

Con un tipo opaco, por el contrario, se prohibía el uso directo, fuera del paquete, de los campos del registro. Se limitaba así el efecto de un cambio en estos campos al interior del paquete: si la semántica, es decir, la forma de usar las operaciones aprobadas, no se modificaba, quien quisiese usar el editor no tenía que estar pendiente de los cambios en su implementación.

Observe que las ventajas que acabo de enumerar beneficiaban al programador que quería usar mi editor. Pero también había ventajas para mi editor: podía descomponer su implementación en bloques, e implementar estos con la misma tecnología de tipos opacos.

> **NOTA** En el caso particular de un editor de texto, para tener dos instancias del mismo ejecutándose en pantalla no basta con declarar dos variables de tipo *Editor*. La explicación más rápida es que todas las instancias tendrían que compartir un único recurso: la pantalla.

Encapsulamiento: comportamiento

Ada es un lenguaje expresivo, potente y seguro, y pudo haber dominado durante mucho más tiempo la vida intelectual en las facultades de Informática. De hecho, una de las características más notables

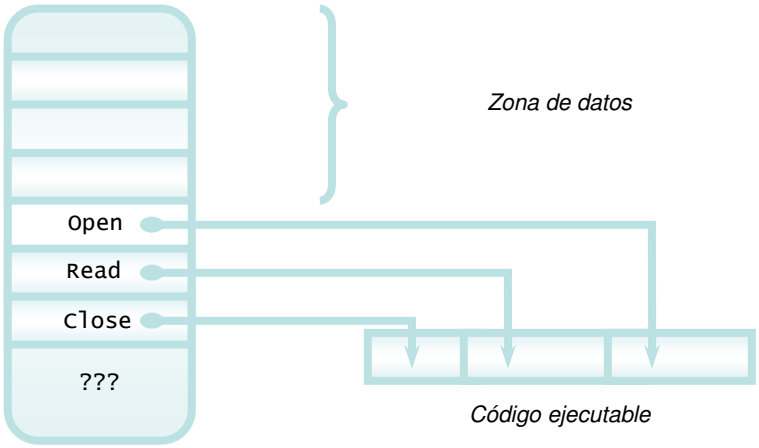
[3] Las otras dos plagas fueron el sistema MS-DOS, por su falta de soporte para la concurrencia, y el perverso formato OBJ de Intel.

de Ada, la *genericidad*, ha tardado bastante en migrar de manera adecuada a otros lenguajes. Sí, existe algo parecido en C++… pero nuevamente, su diseño e implementación han sufrido la influencia negativa de su vinculación al enlazador del correspondiente sistema operativo. Pero, en todo caso, fue la ola de la Programación Orientada a Objetos la que barrió con Ada.

Si aquí se tratase del reconocimiento a los investigadores, habría que retroceder hasta el año 69 y el lenguaje Simula. Pero como se trata más bien de explicar las ideas principales, voy a mostrarle cómo hubiese podido surgir la idea si el año 69 hubiese sido borrado de la línea del tiempo. Y como mi primer lenguaje de programación fue el Pascal, el ejemplo lo tomaré de Turbo Pascal, antes de que le añadiesen las extensiones orientadas a objetos.

La joya se esconde entre el barro del trabajo con ficheros de textos del Turbo Pascal primigenio. La frase *fichero de texto* es engañosa: en los tiempos en que UNIX era el ejemplo a imitar, podía referirse a un fichero "de verdad", pero también a la pantalla, o a la impresora, o si se trataba solamente de lecturas, podíamos estar hablando del mismísimo teclado. En pocas palabras, un fichero de texto, o *dispositivo*, en la jerga de entonces, era cualquier artefacto capaz de escupir o tragar una ristra de caracteres. Hoy hablaríamos de *streams*, eso que traducimos como *flujo de datos*.

En aquellos oscuros tiempos, Turbo Pascal ofrecía un mismo tipo *Text* para todas estas tareas, y este tipo se declaraba como un *record*, o registro. No era un vulgar registro, sin embargo. El siguiente diagrama muestra lo que contenían sus campos:



La parte superior del registro contenía, como muestra el diagrama, los tipos campos que esperamos de una estructura de este tipo: está abierta y funcionando o no, en qué posición del fichero se encuentra, qué tamaño total tiene el mismo, etcétera, etcétera. Tras esta zona, venía lo bueno: los tres campos que he etiquetado en el diagrama como *Open*, *Read* y *Close*[4]. Eran campos especiales porque contenían punteros a las funciones que debían ejecutar estas operaciones, y que variaban en dependencia del tipo de dispositivo representado por el registro. ¿Recuerda los punteros a funciones de la explicación del *quicksort*? Acabamos de cerrar el círculo.

Naturalmente, estos punteros a procedimientos debían ser cuidadosamente inicializados. De ello se ocupaban rutinas como *Assign*, para el acceso a ficheros en disco, o *AssignPrn*, en el caso de escritura en la impresora. Observe que, si tenemos varios registros de este tipo de manera simultánea, hay cierto derroche de memoria. Más adelante, veremos cómo este pequeño problema se resolvió en la Programación Orientada a Objetos.

Lo que más nos interesa ahora es que estamos ante una forma de encapsulación diferente a que por entonces se consideraba "tradicional": en un mismo tipo se encapsulaban *datos*, al modo clásico, pero también *comportamiento*. Dos registros del mismo tipo podían comportarse de manera muy diferente, gracias a que sus datos incluían punteros a las funciones adecuadas. De esta idea a la primera ola de la Programación Orientada a Objetos sólo había un pequeño paso.

[4] No recuerdo exactamente la cantidad y nombres de estos campos, pero es irrelevante para la explicación.

Un pequeño cambio en la sintaxis

Ese pequeño paso era de naturaleza sintáctica, y la aparente irrelevancia del mismo fue lo que despistó a programadores y teóricos durante un tiempo. Supongamos que a un programador de aquella época le pidiesen un conjunto de rutinas para tratar con vectores geométricos. En Pascal se escribiría algo parecido a esto:

```
type
    Vector = record X, Y, Z: Double; end;
```

Luego, y enfatizo lo de "luego", se declararían las rutinas que trabajarían con el nuevo tipo:

```
procedure InitializeVector(var v: Vector);
procedure SumVectors(var result: Vector; const v1, v2: Vector);
function VectorLength(const v: Vector): Double;
```

Imagine ahora que le toca montar un sistema complejo que requerirá el uso de matrices, vectores, cuaterniones y otras criaturas de raro pelaje. Alguien ha querido simplificar su trabajo, y ha mezclado los ficheros de los distintos subsistemas. Pero ese alguien tiene un curioso sentido del orden, y ha agrupado las declaraciones de tipos en un fichero, escrupulosamente ordenadas alfabéticamente, y las rutinas en otro fichero, también respetando el mismo orden. Sí, se trata de una de mis pesadillas favoritas, y siempre despierto lanzando alaridos.

Si en vez de Pascal, el programador hubiese utilizado Ada, con sus paquetes y tipos opacos, la probabilidad de revolver el código tan desconsideradamente habría sido mucho menor… pero seguiría siendo posible.

La idea que mencionaba consistió en convertir estas rutinas en parte de la declaración del propio tipo. ¿No estábamos hablando de encapsular *comportamiento*? Es lógico, entonces, que las rutinas que definen ese comportamiento se consideren parte integrante del tipo de datos. En la notación de Turbo Pascal 5.5, tendríamos entonces:

```
type
    Vector = object
    public
        X, Y, Z: Double;

        constructor Create;
        function Length: Double;
        procedure Add(const V: Vector);
    end;
```

Tomemos nota de las transformaciones:

1. La primera es evidente: las tres rutinas originales son ahora parte de la definición del tipo, que ha dejado de ser un **record** para convertirse en un **object** (una pifia del amigo Hejlsberg, por cierto, pero una pifia feliz, pues dejó libre la palabra **class** para ser usada más tarde en Delphi).
2. Todas las rutinas han perdido un parámetro de tipo *Vector*. En realidad, se ha convertido en un parámetro implícito.
3. También ha desaparecido parte del nombre de cada rutina: *VectorLength*, por ejemplo, ahora se llama *Length* a secas. Esta es una feliz consecuencia del movimiento de las rutinas al ámbito de nombres particular del tipo de datos.
4. La rutina que presumiblemente se encargaba de la "inicialización" del vector, se ha convertido en un *constructor*.
5. La suma de vectores, en cambio, ha perdido no uno, sino dos parámetros. Sobre esto hablaremos enseguida.

Vamos a pasar por alto cómo se implementaba un tipo como *Vector*, en primer lugar, porque no se parece mucho a la técnica de C#, y en segundo lugar, porque es un estilo de declaración que ya es obsoleto incluso para Delphi, el sucesor de Turbo Pascal. Pero sí me interesa mostrar cómo se utilizaba el tipo *Vector*:

```
var
    V1, V2: Vector;
    Len: Double;
begin
    V1.Create;          // En vez de InitializeVector(V1)
    V1.X := 1;
    Len := V1.Length;   // En vez de VectorLength(V1)
    V2.Create;
    V2.Y := 1;
    V1.Add(V2);
    // … etcétera …
end;
```

Hemos transformado uno de los parámetros en cada una de las rutinas en un parámetro implícito, y ahora muestro cómo se pasa ese parámetro implícito: se escribe antes del nombre la rutina, que cambia su nombre a *método*. Observe que el parámetro implícito, en este caso, debe pasarse por referencia. De no ser así, la llamada a *Create* no podría modificar directamente el vector sino, en todo caso, una copia del mismo.

El cambio más significativo, sin embargo, es el que ha sufrido la rutina *SumVectors*. Esta era su declaración original:

```
procedure SumVectors(var result: Vector; const v1, v2: Vector);
```

Se reciben dos vectores y se sintetiza uno nuevo, que representa la suma de los otros dos. En realidad, tanto a usted como a mí nos habría gustado más este otro prototipo:

```
function SumVectors(const v1, v2: Vector): Vector;
```

Con esta variante, recibimos dos vectores y devolvemos el resultado como un tercer vector temporal, que podemos almacenar en una variable o utilizar directamente como operando en otra operación. Así imitamos el comportamiento de las expresiones matemáticas con las que nos familiarizamos desde niños. No existen efectos secundarios… o al menos, no hay necesidad de introducirlos, con lo que se simplifica enormemente el análisis y comprensión del código. Sobre todo, es importante que podamos utilizar esta rutina para *componer* expresiones más complejas. Es fácil, por ejemplo, sumar tres vectores:

```
SumVectors(SumVectors(v1, v2), v3)
```

Con la declaración original sería más complicado, pues necesitaríamos al menos una variable temporal:

```
SumVectors(v_temp1, v1, v2);
SumVectors(v_temp2, v_temp1, v3);
```

EJERCICIO PROPUESTO

¿Podríamos haber reutilizado la primera variable temporal en la segunda llamada? ¿Por qué? ¿En qué condiciones sería seguro hacerlo?

Uno de los problemas de este paradigma de computación es que está muy alejado del hardware de nuestros ordenadores, y suele resultar muy ineficiente. Por supuesto, la ineficiencia siempre es relativa, y puede aceptarse si se obtienen los suficientes beneficios a cambio. Pero el estilo funcional, desligado de otras ventajas de la programación funcional, propiamente hablando, no ofrece lo necesario a cambio de la ralentización en la ejecución.

El prototipo final de la suma de vectores refleja mejor el paradigma de la orientación a objetos:

```
procedure Add(const V: Vector);
```

Tenemos un vector o, más bien, ¡somos un vector!, y nos pasan un segundo vector. Lo agarramos por las piernas, lo levantamos en peso y zarandeamos hasta vaciar el contenido de sus bolsillos. Luego arrojamos la víctima y nos llevamos el botín… Vale, he mentido un poco: el segundo vector no sufre cambios. Pero ahora no se genera un nuevo vector, sino que modificamos uno ya existente. En la mayoría de los casos, esta es la técnica más eficiente, aunque sea la que más nos obligue a

escribir para combinar operaciones. ¿Es éste un buen síntoma? Depende. Si usted quiere programar un *ray tracer*, es lo mejor que le podía pasar: usted necesita velocidad, sobre cualquier otra consideración.

De hecho, con la aparición de la Programación Orientada a Objetos (aquella historia de SIMULA no cuenta a efectos prácticos), ciertos tipos de sistemas recibieron un buen empujón, en el buen sentido de la palabra. Estos sistemas se caracterizaban por la reducción de complejidad tras la encapsulación del estado… vale, he soltado la frase para hacerme el inteligente, pero se puede explicar con facilidad. Por simple agotamiento de las posibilidades, podemos crear nuestras aplicaciones con módulos dotados de memoria propia o no. ¿Qué cree usted que es mejor?

Aunque parezca lo contrario, la respuesta no es trivial. Un servidor de Internet que debe atender miles de respuestas por segundo funciona mejor si no tiene que memorizar, o técnicamente, recordar el estado de cada una de sus peticiones. Pero existen muchos más sistemas en los que recompensa mantener una memoria privada. Un ejercicio mental interesante es revisar los tipos de aplicaciones que se han beneficiado de la Programación Orientada a Objetos. Para empezar, tendríamos los sistemas guiados por eventos (*event driven*, en la jerga sajona). También es interesante comprobar qué tipo de aplicaciones no notaron tanto el cambio. En mi humilde opinión, es más fácil escribir un compilador en C# que en el antiguo C… pero no estoy seguro de que C# supere en este tipo de aplicaciones a un lenguaje funcional clásico.

¿Me permite una imagen como resumen? El mensaje de moda sonaba ahora como un grito desesperado de autoafirmación en una reunión de Programadores Anónimos:

– ¡Sí, soy adicto a la arquitectura Von Neumann! ¿Y qué?

Durante décadas, los teóricos de la Informática habían intentado atenuar, o al menos disfrazar, los problemas de la llamada "arquitectura Von Neumann". Ahora se decían unos a otros: "sí, somos sucios, unos verdaderos cochinos; pero si vamos a ser puercos, al menos hagámoslo bien".

Herencia

Pero la orientación a objetos nos trajo un segundo regalo que todavía provoca encendidas polémicas: la *herencia*. La idea, sin embargo, no era nueva. Los pujos ontológicos de las primeras olas de la Inteligencia Artificial habían popularizado la idea. Esto se debe principalmente a un gremio de gente a quienes les gusta llamarse *filósofos*. No son tan nocivos como los abogados, pero pueden ser utilizados como munición de catapulta por los ejércitos de ideologías totalitarias. En todo caso, lo que distingue a un filósofo del resto de los mortales es su pretensión de poder ayudar a resolver cualquier problema preguntándose que hubiera hecho Aristóteles, Santo Tomás de Aquino o Wittgenstein (todo depende de la tribu originaria del filósofo) de hallarse en su caso.

El caso es que a los filósofos les encantan las clasificaciones y jerarquías: Pinky no es una simple gata, sino un ejemplar de *felis catus*, del género *felis*, del género *felidae*, etcétera, etcétera. Sin embargo, en la vida real usted no se encuentra un *carnívoro* caminando por la calle: se encontrará con un gato, un chucho o, si tiene muy mala suerte, uno de esos tigres borgianos que te dicen *¡che, qué puto mundo!* antes de zamparte de un bocado. Lo que quiero decir es que las jerarquías no forman parte de la razón del mundo, sino del mundo de la razón, y en muchos casos confunden más que ayudan. En particular, la vieja clasificación de Lineo se ha convertido en una curiosidad cultural una vez que conocemos un poco más sobre genes, cromosomas y esas cosas.

Una vez expresadas las correspondientes protestas, veamos qué nos ofrece la herencia de la Programación Orientada a Objetos:

1. La primera promesa es que podremos comernos el mundo bocado a bocado.

   No estoy exagerando. Los informáticos siempre han soñado con una especie de Biblioteca de Borges que traiga respuestas enlatadas para todas las preguntas posibles… y que funcione sobre cualquier hardware con cualquier sistema operativo.

2 La programación orientada a objetos nos permite utilizar software que aún no se ha escrito.

Dicho así, parece sorprendente, pero luego veremos que la explicación es sencilla, y que ya hemos visto la mayor parte de ella.

Lo que ocurre con la herencia es que, con la misma facilidad, se puede emplear bien o mal. Por ejemplo, ¿debería la clase *Punto3D* derivarse por herencia de *Punto2D*? Estructuralmente, sí: un punto tridimensional sólo añade un campo más a un punto en dos dimensiones. ¡Y tiene las mismas operaciones, más algunas nuevas! Sin embargo, no creo que exista algún programador loco que implemente un punto tridimensional en esta manera. ¿Por qué no? Hay explicaciones para todos los gustos. Las favoritas del estimable público son las que entran en delicadezas ontológicas. Personalmente, prefiero señalar que pocas aplicaciones necesitan tratar en plano de igualdad (sí, es un juego de palabras) puntos en el plano y en el espacio. Si nunca vamos a mezclar ambas clases, es una frivolidad complicar, aunque sea de manera mínima, sus implementaciones. Tenga presente que la propiedad que calcularía la distancia al origen de coordenadas de un punto en 3D se tendría que escribir así en C#:

```
public override double Length
{
    get
    {
        double l2d = base.Length;
        return Math.Sqrt(l2d * l2d + this.z * this.z);
    }
}
```

Polimorfismo

La segunda gran ventaja que trae consigo la herencia se conoce como *polimorfismo*. En el contexto de la programación orientada a objetos, se refiere a la posibilidad de que el tipo *real* de una instancia sea diferente del tipo de la variable que hace referencia a ella. Suponga que tenemos dos clases sencillas:

```
public class A
{
    // … campos definidos por A y heredados por B …

    public A() { } // Un constructor sencillo para A.
}

public class B : A
{
    // … campos introducidos por B …

    public B() { } // Un constructor sencillo para B.
}
```

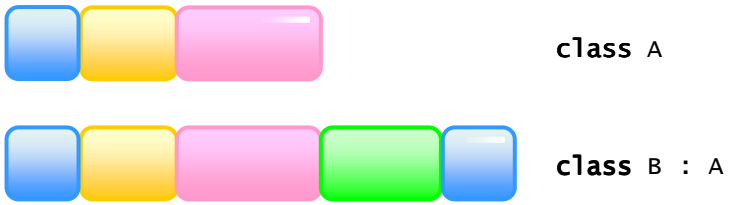
Si un método determinado acepta como parámetro una instancia de la clase *A*, dicho método funcionará sin problemas si en vez de la instancia de *A* le pasamos una instancia de la clase *B*. Para simplificar, supondremos la existencia de un método estático declarado en una tercera clase:

```
public class C
{
    public static void Metodo(A parametro)
    {
        // …
    }
}
```

En estas condiciones, las dos llamadas del siguiente ejemplo son correctas:

```
C.Metodo(new A());
C.Metodo(new B());
```

El siguiente diagrama muestra la razón de esta compatibilidad:



Como puede ver, las instancias de las clases *A* y *B* tienen el mismo formato en su parte inicial. Si el método *Metodo* lee y modifica el campo amarillo de las instancias de *A*, cuando le pasemos una instancia de *B* encontrará también un campo amarillo en la misma posición relativa al inicio de la instancia.

Observe, sin embargo, que esta compatibilidad funciona en una sola dirección. Imagine ahora un segundo método definido en la clase *C*:

```
public class C
{
    public static void OtroMetodo(B parametro)
    {
        // …
    }
}
```

En estas circunstancias, la segunda llamada sería incorrecta, y el compilador mismo se encargaría de señalar el error:

```
C.OtroMetodo(new B());
C.OtroMetodo(new A());  // ¡Error de compilación!
```

OtroMetodo, en efecto, espera recibir una instancia de la clase *B*, y es posible que necesite acceder al campo verde de la clase *B*. Si pudiéramos pasarle una instancia de *A*, ¿qué ocurriría cuando el código intentase acceder al campo verde?

Hemos utilizado métodos de una tercera clase para que el ejemplo fuese lo suficientemente claro. Ahora daremos una vuelta adicional a la tuerca, observando lo que ocurriría con los métodos declarados en la clase *A*:

```
public class A
{
    // … campos definidos por A y heredados por B …

    public A() { } // Un constructor sencillo para A.

    public void Metodo()
    {
        // Manipulamos el campo amarillo.
    }
}

public class B : A
{
    // … campos introducidos por B …

    public B() { } // Un constructor sencillo para B.
}
```

Sabemos, a ciencia cierta, que podemos *aplicar*, usando la jerga de la programación orientada a objetos, el método *Metodo* a cualquier instancia de la clase *A*:

```
A a = new A();
a.Metodo();
```

La propia definición de herencia nos asegura que la clase *B*, por descender de *A*, debe heredar todos los campos de esta clase, y también sus métodos. Esto *tiene* que ser posible:

```
B b = new B();
b.Metodo();
```

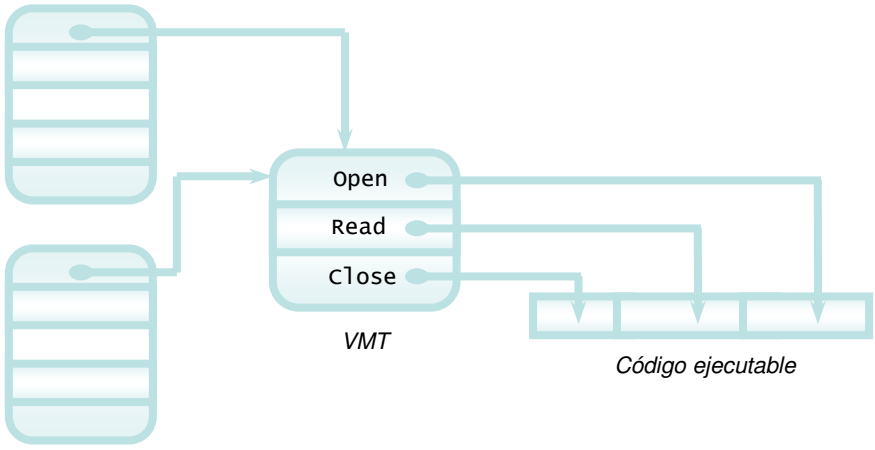
Y efectivamente, lo es. Sólo que ahora nos es más fácil explicar por qué esta llamada es "segura": el código de *Metodo*, programado en la clase *A*, sólo maneja campos definidos en dicha clase, e ignora cualquier campo adicional introducido por sus descendientes. Si le pasamos a *Metodo* una instancia de *B*, el código se encontrará con los mismos campos, en las mismas direcciones relativas al origen de la instancia.

La regla de asignación polimórfica

Métodos virtuales

La Tabla de Métodos Virtuales

¿Recuerda aquella historia sobre los ficheros de texto en el viejo Turbo Pascal?



Gracias a este formato, podríamos averiguar si dos objetos pertenecen a la misma clase comparando sus punteros a sus respectivas tablas de métodos virtuales.

Los límites de la tecnología

En el fondo del problema está el propio leitmotiv de la descomposición modular. En nuestro afán de aislar al componente de su entorno, hacemos que éste desperdicie toda la rica información disponible sobre el sitio donde le ha tocado funcionar.

La Singularidad

Un buen día, un programador, que puede ser usted o uno de sus hijos, creará una aplicación que nos dará respuestas útiles sobre temas más o menos arbitrarios. Ahora podemos pedirle a una calculadora que nos diga cuál es la raíz cuadrada de π, pero a este programa le podremos preguntar cómo trasladar un lobo, una oveja y una col a la otra orilla si en el bote sólo cabemos yo y uno de estos tres objetos. Oh, sí, ahora también… pero necesitamos antes programarlo.

No hará falta que el programa sea perfecto. Tampoco lo somos ni usted ni yo, y nos pagan por nuestro trabajo. Bastará con que las respuestas sean útiles. Normalmente, en épocas anteriores, un avance de este tipo provocaría una pequeña tormenta de nuevas ideas, y la disciplina recibiría un buen empujón que duraría unos cuantos años. Esta vez, sin embargo, ocurrirá algo muy diferente: la aplicación caerá en manos de los expertos en el genoma humano.

Nuestro ADN es uno de los programas más complejos que existe: ha sido escrito por los diez mil y un hermanos de Hanuman acoplados a un bucle de retroalimentación positiva. No sólo por la complejidad de la propia codificación, sino por la propia máquina que debe "ejecutar" dicho código. Los genes egoístas controlan los mandos de una máquina proteínica, a la cual no le puedes pedir despreocupadamente "tráeme un batido de chocolate"… porque te puede traer el batido, sí, pero también una tendencia a padecer migrañas o una incómoda mutación en salva sea la parte. De modo que el pequeño programa de mi historia será recibido como regalo del cielo de los monos por los bioingenieros de por entonces.

Uno de estos investigadores encontrará la forma de mejorar proteínicamente el rendimiento intelectual. No bromeo. La productividad del trabajo en los Estados Unidos es mucho mayor que la europea, y la diferencia crece con los años. Existen hipótesis que atribuyen la diferencia al auge de los psicofármacos, y en especial, al Prozac. Es complicado demostrarlo, pero la hipótesis tiene su lógica: un trabajador o un empresario deprimido aprovecha peor su jornada. ¿Y si fuese posible llegar mucho más lejos con unos pequeños retoques en nuestra química cerebral? El investigador de marras logrará precisamente eso… e iniciará un período de cambios explosivos debido a la inevitable retroalimentación positiva: programadores con cerebros "mejorados" producirán aplicaciones más inteligentes, que serán utilizadas por ingenieros genéticos cada vez más inteligentes para mejorar el funcionamiento de los programadores y de ellos mismos.

En un breve plazo de diez años, y tras algunos sonoros fracasos, la Humanidad dispondrá de una generación de autómatas moleculares programados mediante un protocolo estándar que se utilizarán en un procedimiento médico llamado La Mejora. Será un protocolo estándar porque los gobiernos sentirán pavor ante la idea de perder el control sobre la mejora genética de la especie. Será también un proceso al que cada persona podrá someterse voluntariamente una sola vez en su vida, pues no sólo reparará los daños celulares ya existentes, sino que reprogramará mediante ciertas rutinas genómicas estandarizadas el propio ADN del anfitrión, por lo que los hijos de éste ya nacerán "mejorados".

Las consecuencias de una mejora serán espectaculares: salud, longevidad extraordinaria, una mente más eficiente, sin efectos secundarios desagradables… e incluso una superior belleza física. Si usted o yo viésemos un mejorado, probablemente lo confundiríamos con un ángel. Nuestra especie habrá ascendido entonces, sin sangre ni dolor, a un peldaño superior.

… pero no toda nuestra especie. Siempre existirán personas que, por uno u otro motivo, se negarán a ser "mejorados". Entre ellos se llamarán "naturales", pues los mejorados se apropiarán del término "humanos". Vivirán en pequeñas reservas, aislados del resto de la sociedad, en condiciones cada vez peor, y condenados a la extinción por el goteo constante de disidentes que querrán transformarse en "humanos". Y entonces…

3

CLASES Y OBJETOS

AUNQUE C# OFRECE UNA RICA paleta de tipos de datos al programador, no hay duda de que el color de las *clases* es el favorito: tanto por ser los tipos más usados, como porque las restantes variedades de tipos de datos son, estrictamente hablando, variaciones sobre el concepto de *clase*. Hay incluso lenguajes, como Java, que sólo ofrecen clases en su repertorio; no es un ejemplo a seguir, pero debemos tenerlo presente.

Identidad

Las clases de la plataforma .NET son tipos con *semántica de asignación por 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. 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 objeto por medio de variables, o campos, sin relación aparente.

NOTA

Aunque sería correcto llamar *tipos de referencia* a las clases, esta terminología podría inducir a error en .NET. Para el *Common Language Runtime*, un tipo de referencia, hablando con propiedad, es el equivalente de un puntero a otro tipo en lenguajes tradicionales. Aunque estos tipos no afloran explícitamente en la especificación de C#, son indispensables para implementar técnicas como el traspaso de parámetros por referencia.

Qué hay dentro de una clase

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 eje-

cutarse 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 destructor (¡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.

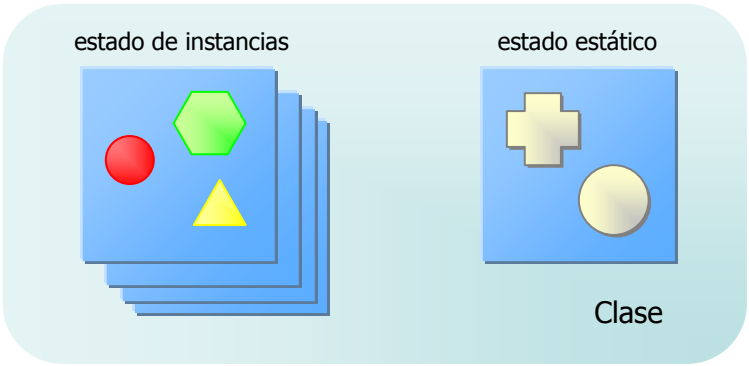
Cuando declaramos un campo, lo normal es que se reserve memoria para éste en cada una de las instancias de la clase. Si le parece anormal la palabra "normal" para hablar de estos campos, le permito que también los llame *campos de instancia*. Podemos también usar el modificador **static** en la declaración 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 modificar **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.

Visibilidad

Cada miembro declarado dentro de una clase 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, 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* (p.72).

Constructores

Aunque sea usted un experto en C++, Java o cualquier otro lenguaje orientado a objetos, no le vendrá mal echar un vistazo a cómo funciona la inicialización de instancias en C#, pues existen en este lenguaje algunas particularidades dignas de mención. Aquí nos ocuparemos solamente de los tipos con semántica de asignación de referencia; en otra palabra, de las clases. Las estructuras tienen otro conjunto bastante embrollado de reglas que examinaremos a su debido momento.

No hay grandes cambios en la sintaxis, comparando sobre todo con Java. Los constructores se declaran usando el mismo nombre de la clase, y no tienen un tipo de retorno asociado. La exigencia sobre la coincidencia del nombre trae como consecuencia que dos constructores diferentes tengan que distinguirse obligatoriamente por el número o tipo de sus parámetros. Podríamos llamar a esta la primera regla sobre constructores, y va acompañada de polémica, porque refleja una limitación impuesta al propio *Common Language Runtime*. Los lenguajes como C#, C++ y Java no tienen problemas, pero otros lenguajes, como Delphi o Eiffel, permiten definir constructores con nombres diferentes y tienen que recurrir a trucos sucios para implementarlos.

NOTA

¿Es tan importante poder dar nombres a los constructores? Tomemos como ejemplo una clase que implemente números complejos. Hay una forma evidente de crear un número complejo: pasar al constructor su parte real y su parte imaginaria. Pero también podríamos desear un constructor que recibiese los componentes del complejo en coordenadas polares. Ambos constructores tendrían entonces idénticos prototipos. En C#, cuando se da este problema, la única solución consiste en implementar uno de los constructores como un método estático.

Segunda regla: toda clase debe tener al menos un constructor. Si un programador no define constructores para una clase, entonces el compilador sintetiza uno por su cuenta. Ese constructor sintético no tiene parámetros.

Tercera regla: los constructores no se heredan. Bueno, no exactamente: usted los hereda, pero no puede mostrárselos a las visitas. Solamente puede usar un constructor heredado para invocarlo antes del código fuente de sus propios constructores. Si una clase no declara sus propios constructores, entonces no tiene constructores, a pesar de todos los que haya podido heredar de su clase base. Por lo tanto, el compilador crearía un constructor sintético, de acuerdo a la segunda regla.

Cuarta regla: todo constructor, antes de comenzar a ejecutar *sus* instrucciones, debe ejecutar una llamada a un constructor heredado o a otro de los constructores definidos en la misma clase. La llamada al constructor heredado puede ser explícita, o implícita. Si es implícita, se asume que se está llamando a un constructor heredado que no tiene parámetros. Si ese constructor heredado no existe, entonces tenemos un error. Suponga que hemos definido una clase como la siguiente:

```
public class A
{
    public A(int i) { … }
    ⋮
}
```

¿Ve alguna llamada al constructor heredado? Se supone entonces que el constructor anterior es equivalente al siguiente:

```
    public A(int i) : base() { … }
```

En este caso, no hay problemas: la clase ancestro de *A* es *System.Object*, y esta tiene un constructor sin parámetros. Observe la sintaxis que se utiliza para ejecutar el constructor heredado, a imitación de Java. Sobre todo, tome nota de que la llamada al constructor heredado se escribe antes del bloque de instrucciones del propio constructor.

La clase que sí tendría problemas sería la siguiente:

```
public class B : A
{
    public B() { … }
```

```
    public B(int i): base(i) { … }
}
```

El error se produciría en el primer constructor, porque la clase ancestro no tiene constructores sin parámetros. Observe, sin embargo, que esto sí sería aceptado:

```
public B() : this(0) { … }
```

En este caso, **this** se ha utilizado para ejecutar el constructor "hermano", declarado en la misma clase *B*. La sintaxis es la misma que hemos visto para las llamadas mediante base: sólo cambia la palabra clave, y por supuesto, su significado. Existen restricciones en las expresiones que se pueden pasar como parámetros a un constructor heredado o hermano. No podemos, por ejemplo, utilizar campos de instancia en estas llamadas. Se supone que, en este punto de la ejecución del constructor, dichos campos aún no están correctamente inicializados.

Inicialización de campos

A pesar de la aparente sencillez de la construcción de instancias, hay que tener en cuenta dos hechos al programar un constructor:

1. Tratándose de clases, antes de la llamada al constructor, se produce la reserva de memoria dinámica para la instancia. Como parte de esta reserva de memoria, todos los campos de la instancia se inicializan automáticamente con ceros.

Esto es bueno. Incluso si no hacemos nada en un constructor, podemos contar con que todos los campos de la instancia estarán inicializados con ceros.

2. En la declaración de un campo, se puede añadir una expresión de inicialización opcional. Antes de que comience a ejecutarse la llamada implícita o explícita al constructor heredado, se ejecutan todas las asignaciones debidas a expresiones de inicialización de campos de instancia.

Este comportamiento suele pillar por sorpresa incluso a viejos lobos de mar. Antes de seguir, observe que lo anterior no se aplica a los constructores que llaman a un constructor hermano. De todos modos, en alguna parte de la cadena de llamada a constructores, habrá uno que llamará a un constructor heredado, y entonces tendrá lugar la mencionada paradoja.

Veamos un ejemplo:

```
public class A
{
    public A()
    {
        Console.WriteLine("A");
    }
}

public class B : A
{
    private int i = 1234;

    public B() : base()
    {
        Console.WriteLine("B");
    }
}
```

Imagine que creamos una instancia de la clase *B*. ¿En qué orden cree usted que se ejecutan las dos escrituras en la consola y la asignación al campo entero definido en *B*? Véalo con sus propios ojos:

```
.method public hidebysig specialname rtspecialname instance void .ctor()
      cil managed
{
      .maxstack 8
      L_0000: ldarg.0
      L_0001: ldc.i4 1234
```

```
        L_0006: stfld int32 ConsoleApplication1.B::i
        L_000b: ldarg.0
        L_000c: call instance void ConsoleApplication1.A::.ctor()
        L_0011: ldstr "B"
        L_0016: call void [mscorlib]System.Console::WriteLine(string)
        L_001b: ret
}
```

Este es el código IL generado para el constructor de *B*, obtenido mediante *Reflector*. Como puede ver, primero se asigna la variable entera, luego se llama al constructor heredado y finalmente se ejecutan las instrucciones que van en el bloque del constructor de *B*.

"No tiene la menor importancia", puedo escucharle desde aquí. "En definitiva, no se puede acceder al contenido del campo *i* desde el constructor de la clase *A*". Bien, si alguien dice lo anterior, apuesto que es un programador muy versado en C++ aunque conoce algo menos de Java y Delphi. Efectivamente, si esto fuese C++, sería imposible que la clase *A* estirase tanto su brazo. Sin embargo, tanto C#, como Java y Delphi, lo permiten. Observe:

```
public class A
{
    public A()
    {
        Console.WriteLine("A: i = " + LecturaFutura());
    }

    protected virtual int LecturaFutura() { return 0; }
}

public class B : A
{
    private int i = 1234;

    public B() : base()
    {
        Console.WriteLine("B");
    }

    protected override int LecturaFutura() { return i; }
}
```

¿Qué aparecerá escrito en la pared si creamos una instancia de *B*? ¿Acaso *half dollar, half dollar, one penny and two bits*?

- Si hacemos el experimento con C++, con los debidos cambios sintácticos, la llamada a *LecturaFutura* desde el constructor de *A* devolverá cero.
- En Delphi, Java y C#, *LecturaFutura* siempre devolverá 1234.

C++, a diferencia del resto de la banda, no limpia con ceros sus instancias antes de ejecutar un constructor. Si C++ permitiese que un constructor ejecutase una función virtual redefinida en un descendiente remoto, se encontraría con campos de valor impredecible. *LecturaFutura* ha devuelto 1234 en C# porque el campo se ha inicializado antes de la llamada al constructor, pero incluso si no existiese la expresión de inicialización, *LecturaFutura* devolvería un predecible cero, gracias a que el campo *i* se habría inicializado automáticamente con ceros al reservar memoria para la instancia.

Para evitar la situación antes descrita, C++ modifica la semántica de las llamadas a métodos virtuales que se realizan dentro de un constructor: mientras se ejecuta el código del constructor definido en *A*, la instancia se comporta como si "sólo" fuese una instancia de la clase *A*. En concreto, los métodos virtuales que se ejecuten desde el constructor de *A* utilizarán la versión más cercana que no sea posterior a la clase *A*.

La estrategia de C++ es robusta, coherente y, por lo tanto, correcta. Pero restringe bastante lo que podemos hacer dentro de un constructor, y es complicado predecir, con una jerarquía compleja de clases, el comportamiento de subsistemas de la clase basados en métodos virtuales. La ruta seguida

por Delphi, Java y C# es quizás menos elegante desde el punto de vista teórico, pero es correcta gracias a la inicialización automática de instancias, y al final, resulta ser más potente que la de C++.

Volvamos a las inicializaciones de campos: está claro que es el compilador quien se encarga de recopilar todas las asignaciones a campos de instancias para crear el código que se ejecuta antes de la inevitable llamada al constructor heredado que corresponda. En el caso en que no existen constructores en una clase, el compilador sintetiza uno recopilando las asignaciones a campos de instancia y añadiendo al final una llamada al constructor heredado que no tiene parámetros. Esta recopilación de expresiones de inicialización de campos no tiene lugar para los constructores que llaman a constructores hermanos: el constructor hermano se encargará, directa o indirectamente, de la inicialización de los campos.

Para demostrarle que los campos se inicializan antes de la llamada al constructor heredado, preferí presentarle el código intermedio extraído mediante *Reflector*. Pero resulta que existe otra forma de demostración, y que la técnica necesaria puede serle útil en algún otro caso. Observe:

```
public class A
{
    public A()
    {
        Console.WriteLine("A");
    }
}

public class B : A
{
    private int i = Devuelve1234();

    public B() : base()
    {
        Console.WriteLine("B");
    }

    private static int Devuelve1234()
    {
        Console.WriteLine("Inicializando campos");
        return 1234;
    }
}
```

En este ejemplo, para inicializar un campo de instancia, hemos utilizado un método estático. C# nos prohíbe usar métodos de instancia para inicializar campos, pero no pone objeciones a los métodos estáticos. Y un método estático puede provocar efectos secundarios interesantes… aunque nunca sobre la instancia que se está construyendo. Por ejemplo, se nos podría ocurrir la malvada idea de añadir un parámetro de tipo *B* al método estático, con la finalidad de zarandear un poco la instancia de *B*:

```
private static int Devuelve1234(B instancia)
{
    ⋮
}
```

Pero el compilador no permitiría entonces pasar **this** a *Devuelve1234* en la expresión de inicialización, con lo el peligro desaparece.

El constructor estático

Como ya sabe, cualquier clase encierra dos almas en su interior: aquella que se traduce en instancias y eso que podríamos llamar "el alma estática". Podía haberme ahorrado el misticismo, pero dicho así, suena más entretenido. Lo que quiero decir es que podemos definir campos estáticos, métodos estáticos, propiedades estáticas e incluso eventos estáticos. ¿Podremos entonces definir constructores estáticos? Claro que podemos, aunque estos constructores estáticos tendrán sus fobias y manías.

La responsabilidad de un constructor estático sería inicializar la "parte estática" de la clase; en otras palabras, los campos estáticos de la misma. La diferencia respecto a un constructor de instancias es que el constructor estático nunca es ejecutado explícitamente. Una consecuencia inmediata de esto es que el constructor estático no puede tener parámetros, y una consecuencia indirecta es que sólo puede declararse un constructor estático, como máximo, para cada clase.

Otra peculiaridad del constructor estático es la forma en que se declara:

```
public class Ejemplo
{
    public static double Sqrt2;

    // ¡Este es un constructor estático!
    static Ejemplo()
    {
        Sqrt2 = Math.Sqrt(2.0);
    }
}
```

Como era de esperar, el constructor estático se declara con la palabra clave **static**, pero además, no admite que se le asocie explícitamente un modificador de nivel de acceso.

Al igual que ocurre con los campos de instancia, los campos estáticos puede también inicializarse in situ, sin esperar al constructor estático, y estas inicializaciones se ejecutan antes del código del constructor:

```
public class Ejemplo
{
    public static double Sqrt3 = Math.Sqrt(3.0);
    public static double Sqrt2;

    static Ejemplo()
    {
        Sqrt2 = Math.Sqrt(2.0);
    }
}
```

Aquí, sin embargo, debo advertirle con toda seriedad:

- ¡No declare constructores estáticos, al menos mientras pueda evitarlo! Utilice, dentro de lo posible, expresiones de inicialización para los campos estáticos.

Para explicar lo que hay en juego, hay que echar mano nuevamente de *Reflector*. Esta es la cabecera, en lenguaje intermedio, de la declaración de una clase que no tiene constructor estático:

```
.class public auto ansi beforefieldinit A extends object
```

Si añadimos un constructor estático, la declaración cambia:

```
.class public auto ansi A extends object
```

Ha desaparecido cierto misterioso modificador *beforefieldinit*… y el acceso a la clase se ha ralentizado al mismo tiempo. El modificador controla el momento en que se inicializan los campos estáticos de la clase. Mientras no haya un constructor estático *explícito*, el entorno de ejecución se siente con libertad de inicializar esas variables en cualquier momento, como por ejemplo, al arrancar la aplicación. Destaco lo de "explícito", porque si existen inicializadores de campos estáticos, el compilador sintetizará un constructor estático, pero tendrá la misma libertad de elegir el momento en que tiene lugar la inicialización. Eso es lo que significa *beforefieldinit*: que la plataforma decidirá cuando le viene bien inicializar la parte estática de la clase.

Si por el contrario, hay un constructor estático explícito, la especificación formal de C# deja muy claro en qué momento debe ejecutarse. Debe hacerlo justo antes de que ocurra la primera de estas dos acciones:

- Cuando se cree la primera instancia de la clase.

- Cuando alguien intente acceder a campos estáticos de la clase.

La única forma de implementar este comportamiento es comprobar si la clase ha sido inicializada cada vez que se accede a un campo estático de la clase y cada vez que se ejecutan sus constructores de instancias. ¿Resultado? Menos velocidad al trabajar con estos recursos de la clase.

4

TIPOS DE INTERFAZ

En la misma época en la que C++ afianzaba su dominio en la programación y liquidaba los últimos focos de resistencia, tenía lugar una callada revolución subterránea que, con el transcurso del tiempo, significaría el fin de la hegemonía de C++.

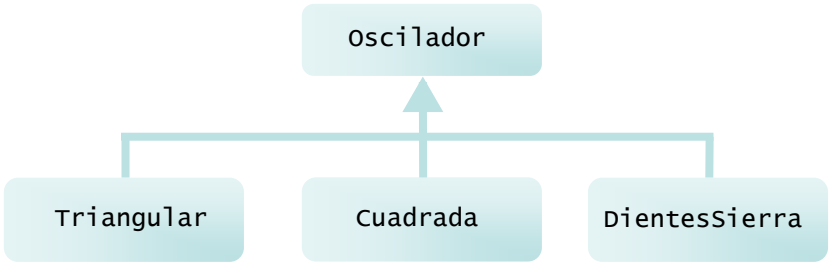
Herencia e interfaces

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 desciende 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 desciende 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.

Implementación de tipos de interfaz

Hay dos técnicas para implementar un tipo de interfaz en una clase. La primera, llamada implementación implícita, consiste en declarar métodos y propiedades en la clase cuyos prototipos coincidan con los de los métodos y propiedades del tipo de 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. Al método *Voltaje* se le exige que sea público. Internamente, el compilador añade los modificadores **virtual** y **sealed**: se trata de un requisito exigido por el CLR.

La otra forma de implementación se conoce como *implementación explícita*:

```
double IOscilador.Voltaje(double tiempo)
{
    // Implementación de la onda triangular …
}
```

Dos detalles han cambiado en la declaración del método:

1. El nombre del método, que ahora aparece con el nombre del tipo de interfaz como prefijo.
2. No se ha mencionado el modificador de acceso al declarar el método.

De esta manera, el método no puede ser ejecutado directamente a través de una variable de tipo *Triangular*. En primer lugar, porque el nombre del método contiene un punto, que no se permite en un identificador, y en segundo lugar, porque el método se genera con nivel de acceso privado.

La principal ventaja de la implementación explícita es que permite resolver potenciales conflictos de nombres provocados por interfaces diferentes que declaran métodos con el mismo nombre. Supongamos que hay dos tipos de interfaz llamados *I1* y *I2*, y que ambos exigen la presencia de un método al que llamaremos *M*. Digamos entonces que la clase *C* implementa ambas interfaces. Si la implementación es implícita, el método *M* declarado en *C* implementaría a la vez los métodos *M* de los dos tipos de interfaz. Puede que esto sea lo que deseemos. Si no es así, entonces debemos recurrir a la implementación explícita, declarando métodos *I1.M* e *I2.M* en la clase *C*.

En determinadas situaciones, nos puede interesar la implementación explícita, por razones más sutiles, de tipo metodológico. Es más arriesgado depender del prototipo de una clase que de un tipo de interfaz: con la clase, se nos pueden escapar dependencias difíciles de detectar. Con frecuencia, se diseñan sistemas alrededor de un tipo de interfaz que es implementado por toda una lista de clases. Si estas clases implementan la interfaz de manera explícita, no existe el riesgo de que, por descuido, se nos escape una llamada a uno de los métodos de la interfaz… pero realizada a través de una referencia a una clase. Sólo tenemos que tener en cuenta que, con las implementaciones explícitas, no podemos tampoco llamar a los métodos así implementados desde la propia clase:

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

    public void Test()
    {
        // ¿Cómo ejecutar el método anterior aquí?
    }
}
```

En estos casos, la única solución consiste en realizar una conversión de tipos:

```
public void Test()
{
    double d = ((IOsciladorthis).Voltaje(0.0);
    Console.WriteLine(d);
}
```

Convertimos una referencia a la clase *Triangular* en una referencia a la interfaz *IOscilador*. Menciono esta posibilidad porque he comprobado que el compilador de C# realiza esta conversión eficientemente. De hecho, ¡no genera código alguno para la conversión! Esto ocurre porque el CLR permite directamente la llamada a *Voltaje* sobre una referencia a *Triangular*.

Interfaces y polimorfismo

Los tipos de interfaz aumentan la complejidad y potencia de las reglas de asignación polimórfica. Suponga que tenemos la siguiente asignación.

```
x = y;
```

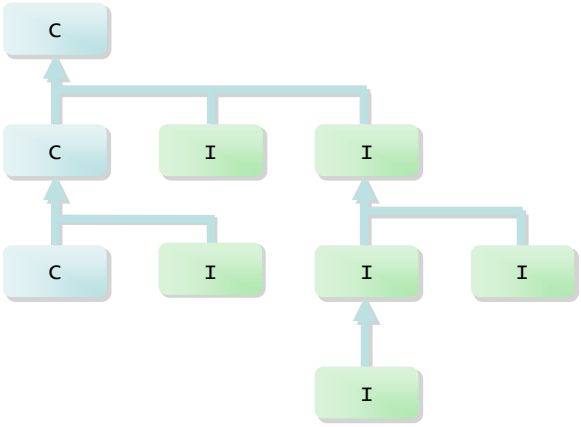
Ahora tenemos que contemplar estas dos posibilidades adicionales:

1. Que el tipo de *x* sea una interfaz, y que el tipo de *y* sea una clase que implemente dicha interfaz.
2. Que el tipo de *x* sea una interfaz, y que el tipo de *y* sea un tipo de interfaz derivado del mismo.

En principio, parece fácil decidir a simple vista si se cumple uno de estos casos, pero en la práctica puede resultar complicado… si el compilador no nos echase una mano, claro. El criterio se complica porque *implementar* y *derivar* son aceptadas tanto en sus variantes directas como indirectas. Por ejemplo, podemos tener una expresión cuyo tipo sea la clase *A*, para la cual se cumple:

- La interfaz *IDisposable* se utilizó para derivar la interfaz *IVentana*, que fue implementada por la clase *Ventana*, que engendró a *VentanaRedonda*, que engendró a *Claraboya*.

Y claro, podemos asignar una claraboya en una variable de tipo *IDisposable* sin que salten las alarmas. El siguiente esquema muestra un ejemplo no muy retorcido de cómo pueden complicarse las relaciones entre clases e interfaces:



Descifrar esta cábala es un poco más sencillo que leer el futuro en las hojas de té. Como clases e interfaces son diferentes especies, he dibujado de azul las primeras, y las segundas de verde. Así se nota a simple vista que las clases se agrupan en una sola rama, como consecuencia de la herencia simple. La clase que analizamos es la de la esquina superior derecha: no se deje engañar por el sentido de las flechas, pues se trata sólo de un convenio de los diagramas de clases. Nuestra clase está a dos pasos de distancia de *System.Object*, la clase base común en .NET, e implementa directamente dos tipos de interfaz. Su clase base implementa una sola interfaz. La mayor complicación viene dada por una de las interfaces, que combina dos interfaces bases. Además, aunque no se puede ver en el diagrama, algunas de estas interfaces pueden aparecer en más de un nodo. Cuando un compilador desea averiguar si dos tipos son compatibles para la asignación, debe explorar un árbol como el que acabo de mostrar.

Métodos de extensión

Antes de explicar qué es un *método de extensión* (*extension method*), debo aclararle que este recurso no está necesariamente vinculado a los tipos de interfaz. Lo que vamos a hacer en esta sección gracias a dichos métodos es posible repetirlo con clases y estructuras. No obstante, he decidido explicar los métodos de extensión en el capítulo dedicado a las interfaces porque es al usarlo con estos tipos que los métodos de extensión cobran más sentido… y presentan menos problemas.

Imagine que acabamos de declarar un tipo de interfaz como éste:

```
public interface IStack
{
    bool IsEmpty { get; }
    bool Top { get; }
    void Push(object value);
    void Pop();
}
```

Quien desee soportar este contrato, tendrá que programar dos métodos y dos propiedades con los prototipos apropiados, ya sea en una nueva clase o en una estructura. A partir de este núcleo mínimo y consistente de funcionalidad, sin embargo, podríamos definir automáticamente otros métodos y propiedades potencialmente útiles. Por ejemplo, ¿qué tal si el usuario de *IStack* pudiese usar un método que eliminase todos los elementos de una pila, o que insertase varios objetos con una sola llamada? Si incluyésemos estos métodos en la declaración de *IStack*, estaríamos jugándole una mala pasada al futuro implementador de la interfaz, obligándole a implementar dos métodos adicionales sin ayuda alguna por parte nuestra. Lo peor de todo es que cada implementación de *IStack* tendría su propia versión de estos métodos adicionales, aún cuando hagan exactamente lo mismo y de la misma manera.

La solución clásica consiste en implementar estos métodos en una clase o estructura auxiliar, como la siguiente:

```
public static class StackExt
{
    public static void PushList(IStack stack, params object[] list)
    {
        foreach (object item in list)
            stack.Push(item);
    }

    public static void Clear(IStack stack)
    {
        while (!stack.IsEmpty)
            stack.Pop();
    }
}
```

De momento, lo de declarar la clase con **static** es un aparente tecnicismo: los métodos van a modificar una pila, y no necesitan información de instancia adicional. Esta decisión nos obliga a declarar también los dos métodos como estáticos, ¡pero eso es lo que deseábamos!

Supongamos ahora que ya tenemos una clase *StackImpl* que implementa la interfaz *IStack*. Con la ayuda de la clase auxiliar, podemos ya ejecutar instrucciones como las siguientes:

```
IStack stack = new StackImpl();
StackExt.PushList(stack, "all", "mimsy", "were", "the", "borogoves");
// …
StackExt.Clear(stack);
```

¿No le deja cierto poso de insatisfacción el listado anterior? Estamos obligando al usuario de la interfaz a recordar un nombre de tipo adicional, *StackExt*, que resulta ser bastante feo, para rematar. La sintaxis de la llamada es también más larga y engorrosa, y dificulta la escritura de *object paths*, o cadenas de llamadas, de cierta complejidad. ¿Qué tal si pudiéramos reescribir el código anterior de la siguiente manera?

```
IStack stack = new StackImpl();
stack.PushList("and", "the", "mome", "raths", "outgrabe");
// …
stack.Clear();
```

Esto es posible en C# 3.0 mediante un truco bastante feo, para mi gusto. Hay que declarar los métodos adicionales dentro de una clase estática, como métodos también estáticos, y tenemos que "decorar" el primer parámetro de cada uno de ellos mediante el modificador **this**:

```
public static class StackExt
{
    public static void PushList(this IStack stack, params object[] list)
    {
        foreach (object item in list)
            stack.Push(item);
    }

    public static void Clear(this IStack stack)
    {
        while (!stack.IsEmpty)
            stack.Pop();
    }
}
```

Lo bueno del truco es que, aunque en mi ejemplo original se aplicó sobre el tipo de interfaz, también podríamos aplicarlo sobre cualquier clase que implementase dicha interfaz:

```
StackImpl stack = new StackImpl();
stack.PushList("and", "the", "mome", "raths", "outgrabe");
// …
stack.Clear();
```

Ahora parece que también existe un *PushList* y un *Clear* en la clase *StackImpl*. Lo que ocurre, en realidad, es que la regla de asignación polimórfica permite usar un *StackImpl* donde quiera que se pueda usar un *IStack*.

La cara oscura de los métodos de extensión

Los métodos de extensión simulan la existencia de métodos en un tipo arbitrario: métodos que no estaban en la mente del programador al diseñarlo. Es decir, no sólo *extienden* interfaces, sino también clases, estructuras e incluso tipos enumerativos. Por ejemplo, tomemos el siguiente tipo enumerativo:

```
public enum Estados
{
    Sólido, Líquido, Gaseoso, Plasma
}
```

Con un poco de maña podemos simular que *Estados* tiene un método que devuelve el "sucesor" de un estado, retornando al primero tras el último:

```
public static class EstadosExtender
{
    public static Estados Succ(this Estados estado)
    {
        return (Estados)(((int)estado + 1) % (int)Estados.Plasma);
    }
}
```

Antes dije que mi sentido de la estética se ve ofendido por estos métodos, y ahora me explico: tiene sentido "extender" una interfaz, pero ¿ocurre lo mismo con una clase? ¿Por qué no definimos directamente el método deseado dentro de la clase? ¿Porque es una clase creada por otros, cuyo código fuente no podemos o debemos modificar? En tal caso, hay cierta justificación en la comodidad que ofrece el recurso, pero enseguida veremos el peligro que se oculta tras tanta amabilidad y conveniencia.

Uno de los puntos negros de los métodos de extensión es la forma en la que se "activan". Supongamos que el ensamblado *A* define una clase *C*, y que el ensamblado *B* contiene una clase con métodos que extienden *C*, definidos dentro de un espacio de nombres *N*. Para poder usar estos métodos, hay que incluir una cláusula **using** que haga referencia al espacio de nombres donde reside la clase extensora, en este caso:

```
using N;
```

Este mecanismo tan absurdo hace que sea muy complicado detectar si determinado tipo está siendo extendido por alguna clase cuya existencia desconocemos.

Por otra parte, si la clase extendida tiene un método con el mismo nombre que un método de extensión, el método de instancia original tiene prioridad. Por ejemplo, no tiene sentido definir un método de extensión *ToString* porque, sea cual sea el tipo extendido, ya tendrá un *ToString* predefinido, que ocultará cualquier método de extensión con el mismo nombre.

Y es aquí donde el peligro acecha:

1. Usted tiene entre manos una clase a la que le "falta" un método llamado *Oops*.
2. Declara entonces una clase auxiliar con un método de extensión *Oops*, y llena su código fuente de llamadas disfrazadas a *Oops*.
3. Un buen día, el programador de la clase original descubre lo bien que estaría tener un método *Oops* en su clase, y lo añade.
4. De repente, todas las llamadas a *Oops* en su código fuente se convierten en llamadas al nuevo *Oops*. Si ambos hacen exactamente lo mismo, no es un problema grave. Pero supongamos lo peor.
5. Lo peor es que a usted le costará sangre averiguar qué demonios se ha roto en su aplicación.

En mi humilde opinión, hubiera sido preferible permitir directamente la declaración e implementación de métodos dentro del tipo de interfaz, y dejar que fuese el compilador quien separase los intrusos en una clase auxiliar aparte:

```
// ADVERTENCIA: ¡¡¡Esto no es C#!!!
public interface IStack
{
    bool IsEmpty { get; }
    bool Top { get; }
    void Push(object value);
    void Pop();

    void Push(params object[] values)
    {
        foreach (object value in values)
            Push(value);
    }

    void Clear()
    {
        while (!IsEmpty)
            Pop();
    }
}
```

Mi propuesta tiene una ventaja sobre la de Microsoft: permitiría definir e implementar propiedades, mientras que de momento no existen "propiedades de extensión". Pero el objetivo de Microsoft al idear este recurso no se reducía a los tipos de interfaz: los métodos de extensión forman parte importante de LINQ.

De todos modos, una vez reconocido el peligro, es verdad que podemos sacarle bastante partido a este recurso. El tipo de cadenas, *System.String*, es un tipo de referencia. Existen dos formas distintas de representar una cadena vacía: mediante un objeto de cadena con longitud cero… y en dependencia del método, puede que se acepte también un puntero nulo como cadena vacía. En consecuencia, para detectar una cadena vacía, la clase *String* nos ofrece el siguiente método:

```
public static bool IsNullOrEmpty(string s);
```

Como puede ver, se trata de un método estático. No podía programarse como método de instancias porque C# prohíbe explícitamente usar un puntero nulo para llamar un método de instancia:

```
TipoClase objeto = null;
// La siguiente instrucción provocará una excepción.
objeto.HazAlgo();
```

Sin embargo, esta restricción no existe para los métodos de extensión. Podemos declarar un método que extienda la clase *String*:

```
public static bool IsEmpty(this string s)
{
    return !String.IsNullOrEmpty(s);
}
```

Ahora podemos escribir código como el siguiente:

```
string cadena = null;
Console.WriteLine(cadena.IsEmpty());
```

Resumiendo: los métodos de extensión son un recurso muy potente, pero no están exentos de peligro. Úselos con prudencia, preferiblemente para "extender" tipos de interfaz. Más adelante, volveremos a tratar con ellos, al estudiar LINQ.

NOTA

Para comprobar que el objeto al que se le aplica un método no es una referencia nula, el compilador de C# traduce la llamada mediante la instrucción **callvirt** de IL aunque, teóricamente, los métodos no virtuales pueden ejecutarse mediante la instrucción **call**. Cuando un método no virtual se ejecuta mediante **callvirt**, el compilador JIT detecta que no existe una tabla de métodos virtuales y llama directamente al método deseado. No obstante, antes comprueba si el objeto activo del método es nulo. Cuando llamamos a un método estático o a un método de extensión, sin embargo, siempre se utiliza la instrucción **call**, que omite esta verificación. Es una ironía que este tipo de llamada sea ligeramente más eficiente que si hubiésemos declarado el método directamente en la clase deseada.

El Conocimiento Secreto

- Descomponga las interfaces con muchos miembros en interfaces más sencillas.

  Preste atención, por lo tanto, al tamaño estimado de las estructuras que defina. Si son lo suficientemente grandes, es preferible pasar la estructura como instancia que como parámetro.

- Utilice interfaces para jerarquías relativamente planas. Para jerarquías profundas, es mejor utilizar la herencia de clases tradicional.

- No utilice tipos de interfaz como sistema para marcar e identificar tipos concretos.

- Recuerde: si hay algo al otro extremo de una referencia de interfaz, tiene que tratarse de un "objeto", obligatoriamente. Por lo tanto, aunque la definición de la interfaz no lo establezca explícitamente, puede utilizar los métodos de *System.Object* con esta referencia.

5

ESTRUCTURAS

PARA DIBUJAR CUALQUIER TONTERÍA EN un monitor, necesitamos puntos, muchos puntos. Una línea comienza en un punto y termina en otro punto. Para dibujar el texto de una cadena de caracteres, hay que indicar dónde, y en ocasiones se utiliza otro punto para delimitar la zona de dibujo. ¿Rectángulos? Puntos. ¿Mapas de bits? Más puntos…

Guerra al despilfarro de memoria

Si todos estos puntos fuesen instancias de una clase, es decir, si estos puntos se construyesen con cargo a la memoria dinámica del proceso… bueno, sería posible que estuviésemos trabajando en Java, que dejando a un lado los tipos primitivos, sólo ofrece clases como tipos de datos.

Toda esta creación de objetos que inmediatamente son abandonados tiene un coste muy alto. No se trata del precio pagado por la construcción en sí, porque en un sistema con recolección automática de basura, la memoria dinámica suele mantenerse compactada. Gracias a ello, una petición de memoria sólo exige mover un puntero interno que indica el tope de la memoria disponible.

Pero no podemos vapulear la memoria impunemente: tanto objeto perdido aumenta la frecuencia con la que se dispara la recolección automática de basura. He podido comprobar esta regla en dos aplicaciones muy exigentes con el reloj, el compilador de Freya y el proyecto XSight Ray Tracer. En ambos casos, la mejora más espectacular consistió en habilitar una caché para los objetos más socorridos: nodos de la pila LALR en el compilador, y rayos en el *ray tracer*.

Implementar una caché de objetos reutilizables tampoco es tarea sencilla. Para empezar, debemos tener un modelo claro del tiempo de vida de los objetos implicados. Si la aplicación accede a estos objetos desde varios hilos paralelos, hay un riesgo elevado de que las cosas empeoren mucho antes de empezar a mejorar. Por estos motivos, antes de recurrir a una caché de instancia debemos comprobar si nuestro problema se puede resolver con la receta oficial de .NET para estos síntomas: las *estructuras*.

Creación eficiente, destrucción indolora

¿En qué se diferencia un tipo de estructura de un tipo de clase? Básicamente, en que al declarar una variable, un campo o un parámetro de una clase, lo que realmente hacemos es reservar memoria para un puntero. Más adelante, tendremos que asignar "algo" a dicho puntero, ya sea un nuevo objeto o un objeto creado en algún otro momento. Por el contrario, cuando declaramos una variable, un campo o un parámetro de un tipo de estructura, estamos reservando memoria directamente para el objeto, sin necesidad de usar una referencia como intermediario.

La idea se explica mejor gráficamente. Suponga que queremos utilizar puntos geométricos en el espacio tridimensional. Podemos utilizar una clase regular para representar puntos:

```
public class PuntoC
{
   public double X, Y, Z;

   ⋮
}
```

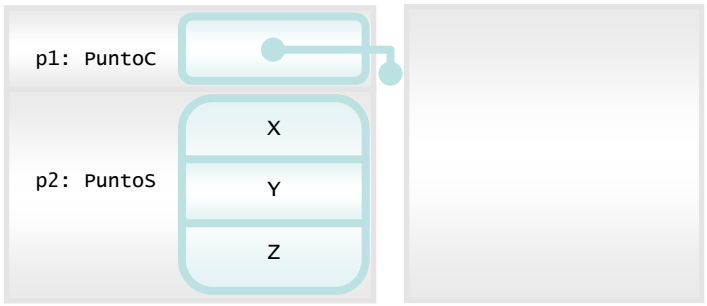
La alternativa es declarar los puntos como estructuras:

```
public struct PuntoS
{
   public double X, Y, Z;

   ⋮
}
```

Imagine ahora que usamos estos dos tipos en un método, para declarar variables locales:

```
public static void Main()
{
   PuntoC p1;
   PuntoS p2;
   ⋮
}
```

Cuando *Main* se ejecuta, reserva espacio automáticamente para las dos variables declaradas en su memoria local, o memoria de pila. Técnicamente hablando, no podemos usar ninguna de las dos variables, porque no han sido inicializadas. Pero ya sabemos con certeza el formato de la memoria local del método:

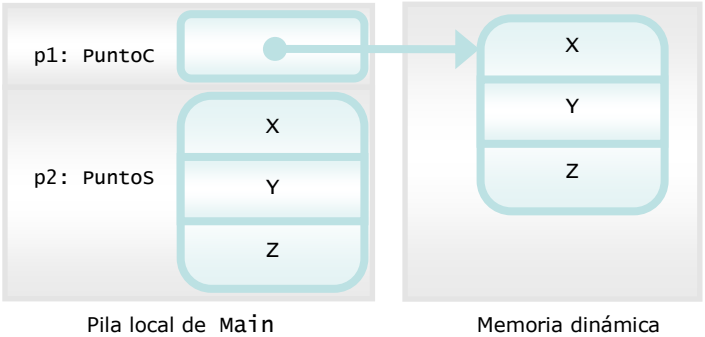


Para el punto representado por una clase, se ha reservado el espacio necesario para un puntero. Si estamos ejecutando la aplicación sobre un procesador de 32 bits, se trata entonces de una zona de cuatro bytes de tamaño. Por el contrario, para el punto representado mediante la estructura, se reservan 24 bytes: un campo de tipo **double** ocupa ocho bytes en esta misma arquitectura, y cada punto contiene tres campos de este tipo.

C# exige que inicialicemos ambas variables antes de utilizarlas:

```
public static void Main()
{
   PuntoC p1 = new PuntoC();
   PuntoS p2 = new PuntoS();
   ⋮
}
```

A pesar del parecido entre ambas asignaciones, ocurren cosas diferentes para cada variable:



El operador **new** aplicado a *PuntoC*, el tipo de clase, reserva memoria para un punto en la memoria dinámica, inicializa sus campos con ceros y devuelve el puntero al objeto recién creado, que finalmente es almacenado en la variable local *p1*. En cambio, la segunda operación es traducida por el operador de forma totalmente distinta. Las instrucciones generadas limpian con ceros la memoria correspondiente al punto en la zona reservada para ella. Para comprobarlo, podemos analizar el código IL generado por el compilador con Reflector:

```
.method private hidebysig static void Main(string[] args)
    cil managed
{
    .entrypoint
    .maxstack 1
    .locals init (
        Estructuras.PuntoC oc1,
        Estructuras.PuntoS os1)

     newobj instance void Estructuras.PuntoC::.ctor()
     stloc.0
     ldloca.s os1
     initobj Estructuras.PuntoS
}
```

La creación de la instancia de clase tiene lugar ejecutando la instrucción **newobj**, que deja la instancia recién construida en la pila. A continuación, esta referencia se almacena en la variable global mediante la instrucción **stloc**. Por el contrario, para construir una instancia de una estructura, primero se obtiene la dirección a la zona donde comienza la instancia mediante **ldloca** (*Load Local Address*). El puntero obtenido es utilizado de inmediato por la instrucción **initobj**. Más que hablar de *construcción*, tratándose de estructuras, deberíamos hablar de *inicialización*.

Debemos tener cuidado al intentar adivinar el funcionamiento real de una instrucción del lenguaje intermedio. Al ejecutarse la aplicación, el compilador JIT tiene mucha libertad para elegir la implementación más eficiente. Aunque no es recomendable obsesionarse con la traducción final a código nativo, creo que esta vez merece la pena echar un vistazo a lo que ocurre entre bambalinas. Este es el código nativo generado por el compilador JIT para un AMD Sempron, un procesador de 32 bits:

```
; PuntoC p1 = new PuntoC();
    mov         ecx,969668h
    call        FFCB0EAC
    mov         edx,eax
; PuntoS p2 = new PuntoS();
    lea         edi,[esp]
    pxor        xmm0,xmm0
    movq        mmword ptr [edi],xmm0
    movq        mmword ptr [edi+8],xmm0
    movq        mmword ptr [edi+10h],xmm0
```

El primer punto se construye llamando a una rutina en la que Visual Studio no nos permite entrar; existen formas, pero no merece la pena considerarlas en este punto. No importa lo que haga internamente la llamada: debe reservar memoria, probablemente mediante una segunda llamada, para luego inicializarla, y en todo caso… ¡se trata de una llamada, como mínimo, y éstas suelen ser costosas! Observe que el resultado se ha almacenado en el registro *edx*.

En contraste, para crear el segundo punto se obtiene la dirección efectiva del punto, dentro del marco de pila utilizado por el método. Las instrucciones que siguen utilizan las extensiones multimedia del procesador: se limpia un registro de ocho bytes, y se asigna directamente su valor en los tres campos que contiene el punto. No es necesario llamar a rutina alguna.

Con esta explicación, he intentado demostrarle que la construcción de una instancia de estructura suele resultar más eficiente que la de una instancia de clase. No obstante, los beneficios no terminan aquí. Cuando el método *Main* termina, la instancia de la clase *PuntoC* se vuelve inaccesible, si no la hemos asignado a una variable con un tiempo de vida más amplio. El coste de la liberación de la memoria no se paga en ese momento, pero la factura nos llega con intereses cuando se produce la siguiente pasada del recolector de basura. En cambio, la memoria de la instancia de la estructura

PuntoS se libera automáticamente cuando el método termina su ejecución, sin que nos cueste un nanosegundo adicional.

¿Cómo se comportan las instancias de estructuras con el resto de las operaciones? Hay que tener mucho cuidado. Cuando se trata de acceder a campos de la instancia, el acceso suele ser muy eficiente:

```
Console.WriteLine(p2.X);
```

En este caso, el código generado carga el valor de tipo **double** directamente desde su ubicación en la pila del método activo. Pero las tornas pueden cambiar si pasamos el objeto completo como parámetro de otros objetos. Imagine que ha escrito el siguiente método auxiliar, en la misma clase que define el método *Main* que hemos venido utilizando:

```
private static void Imprimir(PuntoS p)
{
   Console.WriteLine(p.X);
}
```

Para ejecutar este nuevo método es necesario pasar una copia de todo el punto. Es cierto que el compilador JIT puede sorprendernos al elegir una implementación en código nativo, pero se trata de una copia en toda regla. Esta situación provoca la primera regla de uso para tipos de estructuras:

Utilice estructuras para tipos de datos relativamente pequeños.

Si la estructura que desea declarar contiene muchos campos, piénselo otra vez… a no ser que nunca utilice la estructura como parámetro de otros métodos.

Hay más casos en los que una estructura puede comportarse ineficientemente, pero los veremos más adelante.

Limitaciones: constructores

Las ventajas de las estructuras se pagan mediante algunas limitaciones. Por ejemplo:

1. 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.

2. 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.

3. 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.

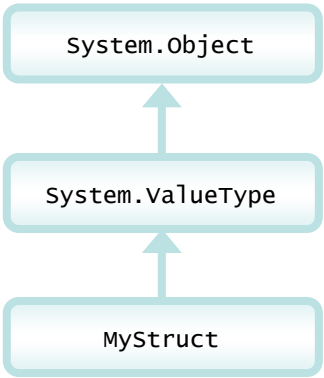
¿Por qué C# impone estas limitaciones? La razón hay que buscarla en el CLR: si fuese posible escribir constructores sin parámetros para las estructuras, tendríamos un serio problema de eficiencia, y nos enfrentaríamos a una complicación considerable del motor de ejecución. Piense en lo que ocurre cuando pedimos memoria para un vector de puntos:

```
PuntoC[] puntos1 = new PuntoC[128];
PuntoS[] puntos2 = new PuntoS[128];
```

En ambos casos, la estructura interna que almacena los 128 puntos, se limpia con ceros. En el caso de las clases, estos ceros representan referencias nulas, lo cual es aceptable. En el caso de las estructuras, se trata de puntos inicializados con ceros: habíamos quedado en que los considerábamos aceptables. De lo contrario, tendríamos que ejecutar 128 veces el código de inicialización de puntos que hubiéramos definido mediante el constructor sin parámetros.

Limitaciones: herencia y visibilidad

Técnicamente hablando, se considera que todas las estructuras descienden de la clase *ValueType*, una clase derivada directamente de *System.Object*. Aunque las estructuras son tipos con semántica de copia por valor, se considera que *ValueType*, el tipo común a todas ellas, es "todavía" un tipo con semántica de copia por referencia:



Las estructuras también presentan una importante limitación relacionada con la herencia:

4 No se puede heredar de una estructura.

Toda estructura se considera automáticamente como si hubiese sido declarada con el modificador **sealed**. 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. En cualquier caso, no se trata de una gran pérdida. Tenemos que aceptar las estructuras como un recurso muy útil relacionado con la eficiencia, y reconocer que cumplen muy bien con su papel.

Esta limitación trae consecuencias en los niveles de visibilidad soportados por las estructuras. De los cinco niveles admitidos por las clases, desaparecen **protected** y **protected internal**, y sólo sobreviven estos tres:

- **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.
- **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**.

La explicación esta vez es sencilla: ¿para qué declarar miembros protegidos, si la estructura no tendrá descendientes?

Métodos virtuales y estructuras

Las estructuras no pueden tener descendientes, pero ellas mismas descienden de una clase que les lega unos cuantos métodos virtuales. ¿Quién podría negarles la potestad de redefinirlos? Para ser exactos, son tres los métodos virtuales heredados de *ValueType*:

```
public override bool Equals(object obj);
public override int GetHashCode();
public override string ToString();
```

De ellos, *Equals* ha sido redefinido en *ValueType*, para implementar una comparación inicial aceptable para un tipo de estructura, y algo parecido ocurre con *GetHashCode*. La implementación por omisión que ofrece *Equals* utiliza reflexión, y eso significa algo de lentitud añadida. Un consejo importante, si va a comparar instancias de una estructura, o si va a usarla en contenedores con capacidad de búsqueda, consiste en redefinir *Equals*, para hacerla eficiente:

```
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; } }

    public override bool Equals(object obj)
    {
        if (obj is Vector)
        {
            Vector v = (Vector)obj;
            return x == v.x && y == v.y && z == v.z;
        }
        else
            return false;
    }
    ⋮
}
```

Hay que tener mucho cuidado si vamos a redefinir *Equals*. Para empezar, no se espera que este método provoque excepciones al ser ejecutado: aunque su parámetro se declara como **object** y puede

recibir valores de cualquier clase, en el caso de que el parámetro no sea otro vector, debemos devolver **false**, en vez de disparar una excepción.

Otro detalle importante es la forma en que comprobamos si el parámetro es un vector. Si *Vector* fuese una clase, en lugar de una estructura, habríamos mezclado la comprobación y la conversión en una sola operación, por medio del operador **as** de conversión de tipos. Este operador, sin embargo, sólo trabaja con tipos de referencia, por lo que hemos tenido que verificar antes de convertir.

Si redefinimos *Equals*, es aconsejable hacer lo mismo con *GetHashCode*, para que sigan funcionando en mutua armonía. De hecho, el propio compilador nos lo recuerda mediante una advertencia si no lo hacemos. Una posible implementación sería como la siguiente:

```
public override int GetHashCode()
{
    return x.GetHashCode() ^ y.GetHashCode() ^ z.GetHashCode();
}
```

La nueva implementación combina los códigos *hash* de los campos mediante una disyunción exclusiva, o *xor*. Tenga en cuenta que los tres campos son de tipo real; si fuesen enteros, nos habría bastado combinar los propios valores de los campos. Una función de *hash* debe ser todo lo rápida que se nos ocurra.

Aunque no es obligatorio, no es mala idea redefinir *ToString*. Es poco probable que la necesitemos en nuestro propio proyecto, pero nos será de gran ayuda durante la depuración:

```
public override string ToString()
{
    return String.Format("V({0},{1},{2})", x, y, z);
}
```

Por último, como se trata de una estructura para la que se ha redefinido *Equals*, se recomienda que redefinamos los operadores de igualdad y desigualdad:

```
public static bool operator ==(Vector v1, Vector v2)
{
    return v1.x == v2.x && v1.y == v2.y && v1.z == v2.z;
}

public static bool operator !=(Vector v1, Vector v2)
{
    return v1.x != v2.x || v1.y != v2.y || v1.z != v2.z;
}
```

Si *Vector* fuese una clase, no deberíamos redefinir estos operadores, porque lo correcto es que comparen las referencias entre sí, no los objetos referidos. Si no estuviésemos interesados en exprimir la estructura en búsqueda de velocidad de ejecución, podríamos llamar a *Equals* desde estos operadores, o viceversa. He preferido, no obstante, crear tres implementaciones independientes.

Interfaces y estructuras

Hemos visto que una estructura permite redefinir los métodos virtuales que hereda. Ahora bien, ¿deberían permitir la definición de nuevos métodos virtuales? A primera vista, no: al ser tipos sellados, nadie los redefiniría. Y efectivamente, no se pueden declarar métodos virtuales en una estructura… al menos, de manera explícita. Una estructura puede implementar uno o más tipos de interfaz, y ya sabemos que el CLR, internamente, define como virtuales los métodos de implementación de interfaces.

Por ejemplo, nuestra estructura *Vector* puede implementar la interfaz *IFormattable*, para permitir el uso de proveedores de formato y cadenas de formato:

```
public struct Vector : IFormattable
{
    ⋮
```

```
    public string ToString(
        string format, IFormatProvider formatProvider)
    {
        if (String.IsNullOrEmpty(format) ||
            format == "G" || format == "g")
            return String.Format(
                formatProvider, "V({0},{1},{2})", x, y, z);
        throw new FormatException(String.Format(
            "Cadena de formato incorrecta: '{0}'.", format));
    }
}
```

Recuerde que, aunque la implementación de esta variante de *ToString* no está marcada como virtual, el compilador internamente añade las marcas necesarias al método compilado.

El nuevo método *ToString* me permitirá mostrarle el comportamiento de las llamadas a métodos de una estructura. Supongamos que, por simplificar, implementamos la redefinición del método heredado *ToString*, el que no tiene parámetros, por medio de la versión creada para *IFormattable*:

```
public override string ToString()
{
    return this.ToString(null,
        System.Globalization.CultureInfo.CurrentCulture);
}
```

Sabemos que la versión de *ToString* con dos parámetros es un método virtual. ¿Cómo se compila esta llamada? En este caso, en vez de realizarse una llamada a través de la tabla de métodos virtuales, el compilador genera una llamada directa al método. La clave está en que la llamada se ejecuta a través de la referencia **this**, que en este caso está asociada a un tipo sellado. No hay posible confusión sobre el método que resultará ejecutado como resultado de la invocación. El compilador lo sabe y genera un tipo de llamada mucho más eficiente.

Ahora bien, ¿qué ocurriría en este otro caso?

```
IFormattable f = new Vector();
Console.WriteLine(f.ToString(
    null, System.Globalization.CultureInfo.CurrentCulture);
```

Aunque en este ejemplo no hay confusión posible en el tipo de instancia al que apunta la variable *f*, en el caso más general la variable podría apuntar a cualquier clase o estructura que implementase la interfaz *IFormattable*. Esta llamada tendría que ejecutarse con la ayuda de la tabla de métodos virtuales asociada a la instancia. Sin embargo, tenemos dos problemas:

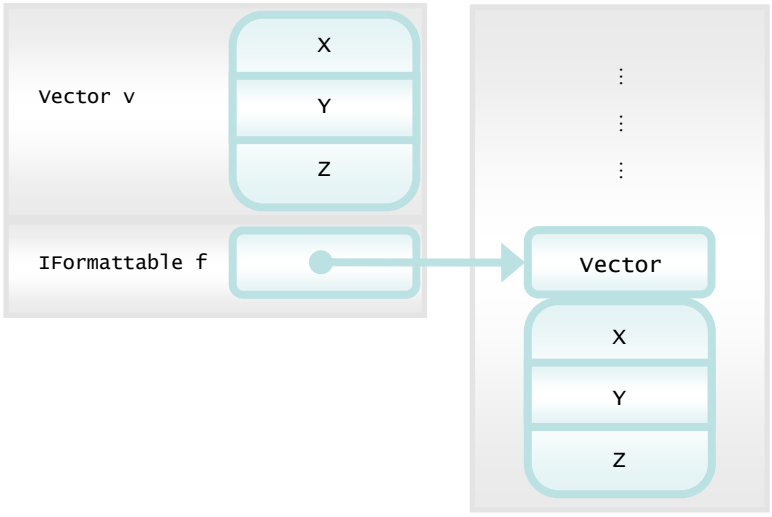
1. ¡*Vector* es una estructura! ¿Cómo es posible entonces que una variable de interfaz pueda apuntar a una instancia de la estructura?
2. Supongamos por un momento que el compilador resolviese la papeleta como lo haría un compilador "nativo": haciendo que la variable apuntase al inicio de la zona donde se almacena la estructura. En ese caso, seguiríamos con problemas: las instancias de una estructura no contienen, en su representación habitual, un puntero a la correspondiente tabla de métodos virtuales.

El sagrado misterio del boxing

La solución a este acertijo es una operación llamada *boxing*, cuyo nombre no traduciré, pues tampoco lo hace la documentación de la plataforma. La operación consiste en crear una instancia en memoria dinámica a partir de una instancia de una estructura, y tiene lugar, precisamente, cuando asignamos un tipo de estructura a una variable, parámetro o campo declarado como una referencia.

En nuestro ejemplo, el *boxing* ocurre gracias a esta asignación:

```
IFormattable f = new Vector();
```



Observe que no se trata de una simple copia de la representación "plana" del vector. Como sugiere la boina que he puesto sobre el vector representado en la derecha del diagrama, el *boxing* añade un puntero a la tabla de métodos virtuales que corresponde al tipo concreto de estructura copiada. Ese puntero es el que garantiza el buen funcionamiento de las llamadas a métodos virtuales que puedan invocarse a través de una variable de interfaz.

Existe una operación inversa, el *unboxing*, que debe deshacer el hechizo sacando el conejo de la chistera. Si sabemos que una referencia a interfaz apunta realmente a un *Vector*, con esta asignación recuperamos el vector en toda su gloria:

```
Vector v = (Vector)f;
```

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

NOTA Aunque aquí he ilustrado el funcionamiento del *boxing* con variables de interfaz, es más común que se produzca con referencias de tipo **object**. En cualquier caso, hay que tener mucho cuidado, pues es una operación costosa y es muy fácil introducirla sin que nos demos cuenta.

¿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*. Como hemos visto, todos los tipos de estructuras descienden 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.

El Conocimiento Secreto

- Los parámetros de estructura con traspaso por valor, se pasan físicamente por copia.

  Preste atención, por lo tanto, al tamaño estimado de las estructuras que defina. Si son lo suficientemente grandes, es preferible pasar la estructura como instancia que como parámetro.

- Tenga mucho cuidado si quiere que una estructura implemente alguna interfaz.

  Normalmente, esto se desea para manejar estructuras polimórficas basadas en el tipo de la interfaz. El problema está en que, para incluir una instancia de estructura en una estructura de este tipo, es necesario someter la instancia al *boxing*. No basta con pasar un punto o referencia a la zona de memoria donde está la estructura, sino que hay que crear un puntero a la tabla de métodos virtuales, como si se tratase de una instancia de un tipo de referencia.

  Personalmente, cuando defino una estructura que voy a usar lo suficiente, suelo incluir en su declaración algunas interfaces básicas como *IEquatable<T>* o *IComparable<T>,* según proceda. Mientras no intente usar una instancia de la estructura a través de una variable declarada con los tipos mencionados, no hay problema. ¿Para qué lo hago, entonces? Recuerde que los tipos de interfaz expresan un contrato, y que los contratos que establecemos en nuestro código deben ser lo más explícitos que sea posible.

  Aunque no lo he comprobado, puede que la idea tenga sentido, bajo determinadas premisas, si vamos a utilizar el tipo de estructura para instanciar un tipo genérico

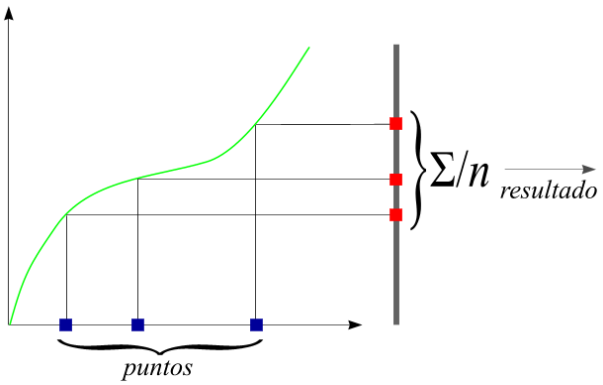
6

TIPOS 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.

Delegados

¿Podemos tener *variables de funciones*? Si fuese posible, podríamos desarrollar algoritmos que actuasen 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? Tendíamos 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 sin-

taxis disponible a partir 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);
```

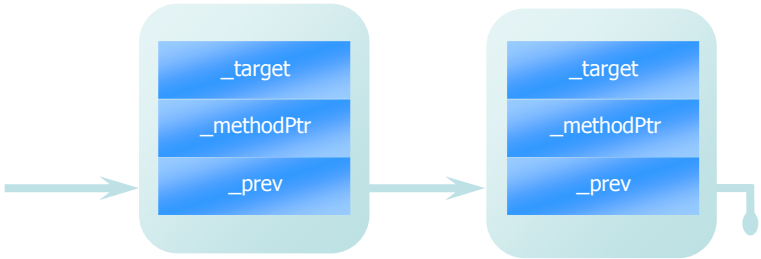
Es la ausencia de paréntesis, a continuación del nombre de la función, la que indica que queremos una referencia a la función, no el resultado de su evaluación.

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# 1.0 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.

Llamadas asíncronas

Los tipos delegados en .NET nos deparan una agradable sorpresa: son la base de un mecanismo potente y elegante de ejecución asíncrona. Lo mejor de todo es que la técnica es segura en lo que respecta al sistema de tipos y el traspaso de parámetros.

Imagine que desea ejecutar el siguiente método en paralelo con el hilo principal de su aplicación:

```
bool HazAlgo(double d, ref string s, out int i);
```

Ahora nos da igual lo que haga el método, pues lo que nos interesa realmente son sus parámetros. El primer paso es declarar un tipo delegado que se ajuste al prototipo del método. Por ejemplo:

```
delegate bool HazAlgoDelegate(double d, ref string s, out int i);
```

De manera automática, el nuevo tipo *HazAlgoDelegate* implementa un par de métodos con los siguientes prototipos:

```
// Estos son métodos automáticamente declarados para HazAlgoDelegate
public IAsyncResult BeginInvoke(
    double d, ref string s, out int i,
    AsyncCallback callback, object state);
public bool EndInvoke(ref string s, out int i, IAsyncResult result);
```

Como puede comprobar, los parámetros de los métodos del delegado se basan en los del método original que queremos llamar. Las reglas son estas:

1. El tipo de retorno de *BeginInvoke* siempre es *IAsyncResult*.
2. Todos los parámetros del método original se copian en el prototipo de *BeginInvoke*.
3. Además, a *BeginInvoke* siempre se le añaden dos parámetros adicionales, el primero del tipo *AsyncCallback*, y el segundo de tipo **object**.
4. El tipo de retorno de *EndInvoke* debe coincidir con el del método original.
5. Los parámetros de *EndInvoke* se generan a partir de los parámetros de salida y de referencia del método original, y se le añade un parámetro de tipo *IAsyncResult*.

Veamos ahora cómo implementar el modelo más sencillo de llamada asíncrona:

```
// Obtenemos un delegado que haga referencia al método a llamar.
HazAlgoDelegate d = HazAlgo;
// Ejecutamos el método BeginInvoke del delegado.
string s = "Valor inicial";
int i;
IAsyncResult result = d.BeginInvoke(3.14, ref s, out i, null, null);
// Ejecutamos algo mientras BeginInvoke termina.
OperacionAburrida();
// Ejecutamos EndInvoke para esperar por BeginInvoke
// y obtener los resultados.
bool b = d.EndInvoke(ref s, out i, result);
```

La llamada a *BeginInvoke* pide un hilo disponible a una caché interna de hilos que .NET mantiene para cada proceso. La ejecución de *HazAlgo* se inicia en ese hilo paralelo, mientras que el hilo principal sigue ejecutándose en paralelo al método asíncrono. En nuestro ejemplo, aprovechamos para ejecutar en paralelo una operación larga y aburrida y, al terminar, esperamos que *HazAlgo* haya terminado. En realidad, el método asíncrono puede o no haber terminado de ejecutarse, pero la llamada a *EndInvoke* bloquea el hilo inicial si el segundo hilo aún está ejecutando código. De paso, aprovechamos para recuperar los resultados de *HazAlgo* mediante los parámetros y el valor de retorno del método *EndInvoke*.

Hay unas cuantas variantes de esta técnica. Por ejemplo, en vez de sentarnos a esperar pasivamente a que la ejecución asíncrona haya concluido, podemos aprovechar el tiempo de espera para otras tareas:

```
// Otra forma de esperar por EndInvoke.
while (!result.Completed)
{
    result.AsyncWaitHandle.WaitOne(1000, false);
    OperacionNoMuyLarga();
}
// De todos modos, es necesario llamar a EndInvoke.
bool b = d.EndInvoke(ref s, out i, result);
```

Para la espera, hemos utilizado el valor de tipo *IAsyncResult* que nos ha devuelto *BeginInvoke*. La propiedad *Completed* nos indica si la ejecución asíncrona ha terminado o no. Mientras no lo haya hecho, llamamos al método *WaitOne* sobre la propiedad *AsyncWaitHandle* del valor devuelto por *BeginInvoke*. Este método bloquea el hilo que lo llama hasta que el hilo asíncrono termina de ejecutar el método paralelo… o cuando se agota el tiempo de espera indicado en el primer parámetro, y que se da en milisegundos. En nuestro caso, esperamos un segundo por cada vez, y entre espera y espera, aprovechamos para ejecutar alguna otra tarea no demasiado compleja. Observe que *siempre* hay que llamar a *EndInvoke*, existan o no parámetros de salida o valores de retorno.

¿Qué haríamos en el caso en que no nos interesase saber el final de la ejecución del método asíncrono? En ese caso, podríamos pasar a *BeginInvoke* la dirección de un método adicional, que se ejecutaría al finalizar la ejecución del segundo hilo. Recuerde que *BeginInvoke* tiene dos parámetros adicionales, y que el tipo del primero de ellos es un tipo delegado:

```
public delegate void AsyncCallback(IAsyncResult ar);
```

Primero declararíamos un método auxiliar que se ajustase al prototipo anterior:

```
private void EjecucionTerminada(IAsyncResult ar)
{
    string s = "";
    int i;
    bool b = ((HazAlgoDelegate)ar.AsyncDelegate).EndInvoke(
        ref s, out i, ar);
}
```

Este ejemplo no es quizás el más adecuado para esta técnica, pues estamos ignorando todos los resultados de la ejecución asíncrona. Observe, de todos modos, la conversión de tipos que realizamos para obtener el delegado original a partir del parámetro *IAsyncResult*.

Finalmente, la dirección de este método se pasaría como parámetro de *BeginInvoke*:

```
// Obtenemos un delegado que haga referencia al método a llamar.
HazAlgoDelegate d = HazAlgo;
// Ejecutamos el método BeginInvoke del delegado.
string s = "Valor inicial";
int i;
IAsyncResult result = d.BeginInvoke(3.14, ref s, out i,
    EjecucionTerminada, null);
// Podemos seguir a nuestro aire…
```

Cuando la ejecución de *HazAlgo* termine, se ejecutará *EjecucionTerminada*, y es allí donde se llamará a *EndInvoke*, por esta vez.

7

COMPONENTES

C# Y JAVA SON LENGUAJES ORIENTADOS a objetos. Para ser más exactos, C# es un lenguaje orientado a componentes. Java, sin embargo, no lo es. ¿Qué significa lo de "orientado a componentes"? ¿Por qué Java no cumple los requisitos?

Componentes versus clases

El paso de "simples clases" a componentes es muy parecido al que se produjo en la industria electrónica cuando entró en escena el circuito impreso. Antes de la popularización del circuito impreso, el ingeniero electrónico tenía que amañárselas para soldar entre sí válvulas, transistores, diodos, resistencias y condensadores. ¿Ha visto alguna vez las entrañas de un televisor o de una radio de principios de los 50s? Si el aparato tenía válvulas, los zócalos de éstas se fijaba al chasis metálico, y sus terminales servían de puente o apoyo a las restantes piezas. Diseñar un circuito sencillo requería una buena imaginación gráfica para distribuir las piezas en el espacio sin que se cruzasen por accidente. Los circuitos impresos dieron un vuelco a la situación, simplificando las tareas de interconexión de las partes implicadas.

Un componente de software es una clase a la que se le ha añadido la infraestructura necesaria para conectarse con facilidad a otros componentes dentro de un sistema de desarrollo dado. Tomemos una clase de .NET al azar; por ejemplo, la clase *Hashtable*. ¿Es *Hashtable* un componente? No, no lo es. Le falta la capacidad de crear instancias persistentes sobre las superficies de diseño de Visual Studio. Una clase de la plataforma adquiere esa capacidad cuando implementa la interfaz *IComponent*, del espacio de nombres *System.ComponentModel*, ya sea de forma directa, o de forma indirecta, usando la clase *Component* como ancestro. La interfaz *IComponent* se declara así:

```
public interface IComponent : IDisposable
{
    ISite Site { get; set; }
    event EventHandler Disposed;
}
```

Un componente .NET se comunica con la superficie de diseño que lo aloja por medio de la propiedad *Site*. Al componente se le exige que permita su destrucción determinista (p.72) y que nos avise mediante un evento si es destruida. Esto es necesario, por ejemplo, para evitar que otros componentes hagan referencia a un componente que ya ha sido eliminado de la superficie de diseño.

¿Cómo se comunica la superficie de diseño con el componente? La superficie contenedora asume que el componente "publica" propiedades y eventos. Con la lista de propiedades y eventos soportados por el componente activo, la superficie prepara el contenido que mostrará el inspector de propiedades. Finalmente, el programador utilizará el inspector de propiedades para establecer el valor inicial de las propiedades de cada componente, y para enganchar métodos a los eventos de cada componente.

La pregunta del millón es: ¿por qué se utilizan propiedades? Una propiedad es una especie de campo con superpoderes. ¿Por qué, entonces, no se trabaja directamente con los campos? La respuesta, evidentemente, tiene que ver con los "superpoderes" mencionados. El programador espera que, cuando asigne una cadena en la propiedad *Text* de un botón, el cambio se muestre inmediatamente en el botón ubicado sobre la superficie de diseño. Si trabajásemos directamente con los cam-

pos del botón, tras asignar un valor en el campo correspondiente, habría que ejecutar automáticamente un método para actualizar el aspecto del botón. La propiedad, por el contrario, encapsula para beneficio nuestro, el campo más el método de actualización en los casos en que sea necesario.

Está claro que podemos idear otras estrategias para poder modificar una instancia en tiempo de diseño… pero todas las alternativas al uso de propiedades son peores. Esto es lo que ocurre con Java, que no soporta ni propiedades ni eventos. Para poder utilizar una clase en tiempo de diseño, el autor de la clase tiene que suministrar información adicional que, a la larga, simula la existencia de propiedades de cara al entorno de desarrollo. Como consecuencia, es mucho más complicado crear un *componente* en Java que hacerlo con C# o con cualquier otro lenguaje que soporte propiedades y eventos.

Propiedades

Una propiedad imita el comportamiento de un campo mediante dos métodos internos que deben funcionar en sincronía. Si la propiedad es de sólo lectura, sólo es necesario un método. No existen propiedades de sólo escritura, porque no tendrían mucho sentido.

Las propiedades más sencillas son aquellas que encapsulan un campo, como en el siguiente ejemplo:

```
private int edad;

public int Edad
{
    get { return edad; }
    set { edad = value; }
}
```

Como puede ver, esta propiedad ha necesitado dos métodos. El método de lectura, identificado mediante **get**, no recibe parámetros, y debe devolver un valor del mismo tipo que la propiedad. El método de escritura, identificado con **set**, no devuelve nada, pero debe recibir un valor del mismo tipo que la propiedad. El parámetro no se declara: el valor asignado a la propiedad puede obtenerse mediante el identificador **value**.

Podemos utilizar una propiedad de este tipo para verificar que los valores asignados al campo sean siempre correctos. La siguiente propiedad complica un poco su método de escritura:

```
public int Edad
{
    get { return edad; }
    set
    {
        if (value < 0)
            throw new Exception("Edad incorrecta.");
        edad = value;
    }
}
```

Si el valor que se quiere asignar a la propiedad no nos gusta, impedimos la asignación lanzando una excepción.

Se pueden usar propiedades para sintetizar información a partir de otros campos y propiedades. La siguiente propiedad calcula la edad a partir de la fecha de nacimiento, en vez de almacenar la edad directamente en un campo:

```
private DateTime fechaNacimiento;

public int Edad
{
    get
    {
        DateTime hoy = DateTime.Today;
        if (hoy.DayOfYear >= fechaNacimiento.DayOfYear)
            return hoy.Year - fechaNacimiento.Year;
```

```
            else
                return hoy.Year - fechaNacimiento.Year - 1;
        }
    }
```

En casos como éste, hay que preguntarse si debemos declarar *Edad* como una propiedad de sólo lectura o como una función sin parámetros:

- Si el algoritmo consume mucho tiempo, es preferible usar una función sin parámetros. El programador que accede a una propiedad espera que su lectura sea igual de rápida que la lectura de un campo. En cambio, de una llamada a función se puede esperar cualquier cosa.
- Si el algoritmo de lectura cambia el estado observable de la instancia, es preferible usar una función.

En las versiones 1.0 y 1.1 de la plataforma, los métodos de acceso tenían siempre la misma visibilidad que la propiedad que los declaraba. A partir de .NET 2.0, es posible declarar propiedades con métodos de diferentes visibilidades. Por ejemplo:

```
public int NombreCompleto
{
    get { … }
    internal set  { … }
}
```

En este caso, cualquier clase, en cualquier ensamblado, puede leer el valor de la propiedad. Por el contrario, al declarar el método **set** como **internal**, sólo podrán escribir sobre la propiedad las clases situadas dentro del mismo ensamblado que la clase a la cual pertenece la propiedad.

Indexadores

Un *indexador*, o *indexer*, es un tipo muy especial de propiedad que sirve para simular el comportamiento de los tipos de vectores (*arrays*). Imagine que estamos hablando de C, y que escribo en la pizarra la siguiente expresión:

```
c[0]
```

Para simplificar, supongamos que *c* es una variable local. ¿Qué podemos inferir sobre el tipo de esta variable? Si el lenguaje es C, entonces la variable es un vector de algún tipo de datos. ¿Y si el lenguaje es C++? Habría que andar con más cuidado: puede que *c* sea un vector, pero puede darse también el caso que se trate de una variable de tipo clase (o estructura), para la cuál se haya sobrecargado el significado del operador `[]`.

En C# existe la sobrecarga o redefinición de operadores, aunque es bastante diferente de la sobrecarga en C++. Los autores de C# decidieron no permitir la sobrecarga de los corchetes, sustituyéndola por el uso de indexadores, que es una técnica más potente. De entrada, cuando se redefinen los corchetes en C++, sólo se puede usar un índice o argumento, mientras que los indexadores admiten cuantos argumentos o índices deseemos dentro de los corchetes. Además, para que la sobrecarga de corchetes en C++ permita la asignación, el tipo de retorno del operador tiene que ser una *referencia*, en el sentido de C++. Por el contrario, como los indexadores están basados en dos métodos de acceso, como las restantes propiedades, tenemos más libertad para definir el significado de la asignación.

Es un poco complicado encontrar un ejemplo de indexador que merezca la pena, y no es porque se usen poco. La mayoría de los ejemplos que tienen sentido son ejemplos en que sólo se permiten las lecturas, y los ejemplos con lecturas y escrituras terminan usando alguna estructura de datos privada que está basada, a su vez, en una clase con indexadores. Haré el intento, de todas maneras:

Vamos a definir una clase para representar vectores en tres dimensiones. Cada vector almacena tres valores reales, uno por cada eje de coordenadas:

```
public class 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; } }
    ⋮
}
```

Ahora bien, en muchos algoritmos, es más cómodo acceder a los componentes del vector por medio de subíndices. En vez de `v.X`, podríamos preguntar por `v[0]`, y en vez de `v.Y`, nos valdría `v[1]`. Este efecto lo podríamos conseguir mediante un indexador dentro de la clase *Vector*:

```
public double this[int index]
{
    get
    {
        switch (index)
        {
            case 0: return x;
            case 1: return y;
            case 2: return z;
        }
        throw new ArgumentOutOfRangeException("Indice incorrecto");
    }
    set
    {
        switch (index)
        {
            case 0: x = value; return;
            case 1: y = value; return;
            case 2: z = value; return;
        }
        throw new ArgumentOutOfRangeException("Indice incorrecto");
    }
}
```

Un indexador se define como una propiedad, excepto que:

- El nombre siempre es la palabra clave **this**.
- A la palabra clave **this** le sigue una lista de parámetros encerrada entre corchetes. Todos los parámetros deben pasarse por valor.

Al igual que ocurre con las otras propiedades, tenemos que definir un método de acceso de lectura y, de manera opcional, un método de acceso para escritura. Ambos métodos reciben los parámetros declarados entre corchetes. El método de lectura debe devolver un valor con el mismo tipo que el de la propiedad, y el método de escritura, aunque no devuelve nada, recibe un parámetro adicional llamado siempre **value**, cuyo tipo es el de la propiedad.

En el ejemplo del vector, hay un solo parámetro para el indexador, y es de tipo entero. Sin embargo, sólo consideramos correctos tres de los posibles valores que permite un entero. Si alguien intentar pasar otro valor, respondemos con una excepción.

C# nos permite declarar cuantos indexadores se nos antoje, siempre que se puedan distinguir entre ellos por medio de sus argumentos. Por ejemplo, la clase *DataRow*, que representa una fila de una tabla en memoria, define seis indexadores diferentes. Tres de ellos tienen un solo parámetro, y permiten lecturas y escrituras:

```
public object this[DataColumn column] { get; set; }
public object this[int columnIndex]   { get; set; }
public object this[string columnName] { get; set; }
```

Los tres restantes admiten dos parámetros, pero sólo permiten la lectura:

```
public object this[DataColumn column, DataRowVersion version] { get; }
public object this[int columnIndex,   DataRowVersion version] { get; }
public object this[string columnName, DataRowVersion version] { get; }
```

En realidad, el *Common Language Runtime* permite más libertades con los indexadores que las soportadas por C#. Mientras que C# nos obliga a usar **this** para declarar un indexador, la propia plataforma nos permite emplear cuantos nombres deseemos… y para cada uno de estos nombres, podemos definir todas las versiones que queramos, siempre que se puedan distinguir entre sí por sus argumentos. VB.NET no sufre esta restricción. Los seis indexadores de *DataRow* realmente se llaman *Item*, y con este nombre aparecen en la ayuda de la plataforma.

NOTA

Desconozco la razón tras esta limitación de C#. Quizás los autores encontraron ambigüedades en la interpretación cuando se levanta la restricción. En todo caso, he mencionado este asunto porque va a tropezar con la dichosa y misteriosa propiedad *Item* con mucha frecuencia.

Propiedades con implementación automática

C# 3 ha añadido una pequeña, aunque largamente esperada, novedad. Se trata de propiedades con implementación automática, o si lo prefiere, implícita. La mayor parte de las propiedades, en la práctica, se declaran de manera "preventiva": sabemos que es mala idea dejar un campo a la intemperie, y nos curamos en salud encapsulando su acceso dentro de una propiedad. Por lo tanto, y al menos en su forma inicial, la mayoría de las propiedades terminan implementándose directamente a través de un campo.

Ahora podemos ahorrarnos algo de tiempo declarando propiedades de la siguiente manera:

```
public struct Vector
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
    ⋮
}
```

La sintaxis es casi idéntica a la de las propiedades abstractas… pero falta el modificador **abstract**. El compilador declara internamente un campo del tipo apropiado, con visibilidad privada, e implementa los métodos de acceso correspondientes. Como el campo no tiene un nombre conocido, sólo podemos manejarlo indirectamente, a través de la propiedad.

Observe que es necesario declarar ambos métodos de acceso. Si quisiéramos una propiedad que no pueda modificarse fuera de la clase que la declara, tendríamos que incluir un modificador de visibilidad al declarar el método de escritura:

```
public double X { get; private set; }
```

La implementación, sin embargo, no es la mejor posible. Si se accede a la propiedad desde la misma clase que la declara y la propiedad no se ha declarado virtual, el compilador podría traducir automáticamente las referencias a la propiedad por referencias al campo base. Pero, al menos en esta versión, el compilador insiste en utilizar los métodos de acceso.

Otro problema tiene que ver con la inicialización del campo base de la propiedad: no podemos utilizar un inicializador. Si la propiedad se define dentro de una estructura, tendremos problemas adicionales para darle un valor inicial a la propiedad que no sea el valor por omisión. Supongamos que deseamos un constructor para la estructura *Vector* que hemos usado como ejemplo. El siguiente código sería lo más natural… aunque el compilador no lo acepte:

```
// ERROR: el compilador se quejará porque Vector es una estructura.
public Vector(double x, double y, double z)
{
    this.X = x;
    this.Y = y;
    this.Z = z;
}
```

La culpa la tiene la regla de asignación definida de C#, que exige en este caso que todos los campos de una estructura estén explícitamente inicializados antes de llamar a cualquier método. Como la asignación a una propiedad automática se traduce en una llamada al método de escritura, el compilador protesta en la primera línea del código. Por fortuna, hay un truco para salir del paso:

```
public Vector(double x, double y, double z) : this()
{
    this.X = x;
    this.Y = y;
    this.Z = z;
}
```

La llamada al constructor implícito resuelve el problema de la asignación definida, aunque al precio de inicializaciones innecesarias. Con un poco de suerte, el compilador JIT podría eliminar las asignaciones redundantes.

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 implementa-

ción 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.

Clases parciales

Ha pasado mucho tiempo desde la invención de Pascal por Niklaus Wirth. Desde entonces, el hardware ha mejorado espectacularmente. El software y las ideas relacionados con el software han cambiado con mucha más lentitud. Tomemos como ejemplo los compiladores: en los años 70, las limitaciones de memoria dinámica obligaba a diseñarlos como máquinas que transformaban un flujo de datos en varias pasadas. Este diseño siguió utilizándose cuando ya habían desaparecido las limitaciones que lo habían motivado.

Por desgracia, esta característica de la implementación del compilador se propagó al diseño de los propios lenguajes de programación. Un ejemplo clásico ocurre en Pascal. Suponga que necesitamos dos procedimientos *A* y *B*. El procedimiento *A* llama al procedimiento *B*, y este necesita también llamar al procedimiento *A*. En Pascal "clásico", tendríamos que utilizar una declaración **forward** para permitir la recursividad mutua:

```
{ Esto es Pascal }
procedure B; forward;

procedure A;
begin
    ⋮
end;

procedure B;
begin
    ⋮
end;
```

La declaración que termina con **forward** es un anuncio de la verdadera declaración, que tiene lugar más adelante. Sin ella, cuando compilásemos el método *A*, no sabríamos cuál es el prototipo real del método *B*. En Delphi, la mayoría de estas restricciones se han suavizado, pero todavía encontramos requerimientos en el orden de declaración que son consecuencia indirecta de las limitaciones de hardware en la década de los 70s.

En este momento, la mayoría de los compiladores para lenguajes modernos asumen que es posible guardar una representación completa del proyecto que compilan en la memoria dinámica del ordenador. La fase de análisis sintáctico crea una estructura de datos llamada *Árbol de Sintaxis Abstracta*, o AST. Las restantes fases de la compilación trabajan sobre esta estructura. En la plataforma .NET, esta tendencia se lleva a sus últimas consecuencias. Cuando se compila un proyecto, el resultado debe ser el mismo que si mezcláramos todos los ficheros de código del proyecto en un único y gigantesco fichero fuente.

Este comportamiento sirve para explicar un útil recurso añadido a C# 2.0: las *clases parciales*. En la primera versión, todo el código correspondiente a una clase debía escribirse en el mismo fichero, de manera continua. Ahora, por el contrario, podemos fragmentar la definición de una clase. Los fragmentos pueden escribirse en ficheros diferentes, o incluso dentro de un mismo fichero. Por ejemplo, esto es válido (aunque absurdo):

```
public partial class A
{
    public A() { … }

    public int Valor
    {
        get { return valor; }
        set { valor = value; }
    }
}

public class B
{
    ⋮
}

public partial class A
{
    private int valor;
}
```

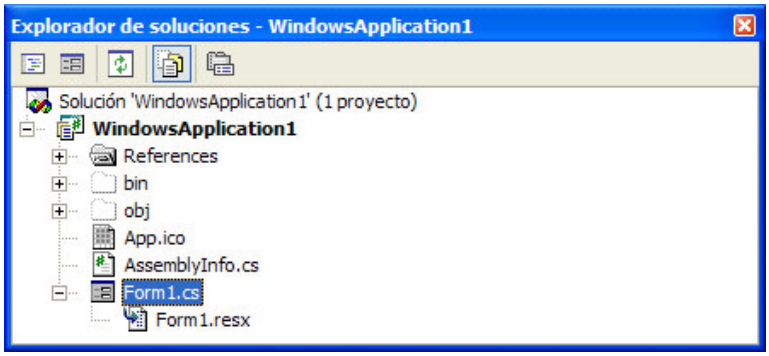
La clase *A* ha sido dividida en dos fragmentos, y ambos fragmentos deben utilizar el modificador **partial** para que el compilador acepte esta división. Para resaltar la división, he intercalado una segunda clase, aunque no es necesario: los fragmentos podrían incluso escribirse uno detrás del otro. En el primer fragmento de *A* he declarado un constructor público y una propiedad. El segundo sólo contiene la declaración del campo privado que se utiliza para implementar la propiedad. Este reparto de declaraciones es completamente arbitrario: puedo distribuir las declaraciones en la forma que me apetezca.

En el ejemplo anterior, la clase *A* heredaba directamente de *System.Object*, y no implementaba tipos de interfaz. Supongamos ahora que *A* debe implementar la interfaz *IDisposable* (p.73):

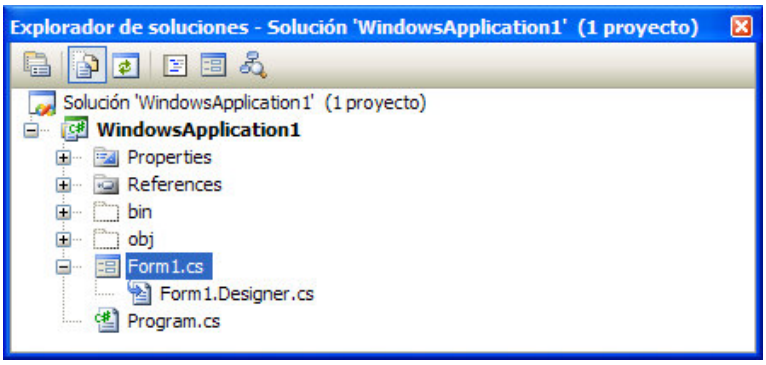
```
public partial class A : IDisposable
{
    public A() { … }
}

partial class A
{
    public void Dispose()
    {
        ⋮
    }
}
```

Podíamos haber repetido la mención al tipo de interfaz en ambos fragmentos, pero como muestra el ejemplo, podemos incluir la mención en sólo uno de ellos. Es más, esta vez he omitido el modificador de visibilidad de la clase en el segundo fragmento. El mismo principio es aplicable a la declaración de una clase ancestro: podemos repetir la declaración en cada fragmento, o podemos omitirla en uno o más de ellos.

A pesar de la aparente inocuidad de las clases parciales, Visual Studio 2005 le saca buen partido. En la versión 2003, Visual Studio ubicaba el código generado automáticamente para formularios, conjuntos de datos y componentes en el mismo fichero donde el programador escribía su propio código. La siguiente imagen muestra los elementos de un proyecto de aplicación recién creada en Visual Studio 2003:



Al fichero *Form1.cs*, que es el que corresponde a la ventana principal, se le asocia un fichero de recursos, de extensión *resx*, usado internamente por Visual Studio. No obstante, como ya he dicho, tanto el código generado por el entorno de desarrollo como el código escrito por el usuario, utilizan el mismo fichero… con todos los peligros que puede imaginar. Esta otra imagen muestra lo que ocurre en Visual Studio 2005:



Por algún motivo, ya no aparece el fichero *resx*, pero tenemos un nuevo fichero de extensión *cs*. Es decir, tenemos un nuevo fichero con código en C# asociado al formulario. En este nuevo fichero, Visual Studio genera las instrucciones necesarias para inicializar el formulario y sus controles. El fichero *Form1.cs* es ahora completamente nuestro. La clase *Form1* se declara en *Form1.Designer.cs* de esta manera:

```
partial class Form1
{
    ⋮
}
```

No se menciona ni la clase base ni la visibilidad del formulario. El segundo fragmento de la clase *Form1*, en el fichero *Form1.cs*, comienza así:

```
public partial class Form1: Form
{
    ⋮
}
```

Métodos parciales

Visual Studio genera código basado en clases parciales con bastante frecuencia, pero hay un problema con las clases parciales que nos puede complicar la vida: las técnicas de inicialización de la clase. Normalmente, parte de la inicialización se realiza en las propias declaraciones de campos, y el resto se encapsula dentro de los constructores de la clase. Las cosas se ponen feas cuando tenemos una clase dividida en dos fragmentos, con los constructores declarados en el fragmento que no podemos modificar manualmente. Por ejemplo:

```
// "Nuestro" fragmento
public partial class A
{
    ⋮
}

// Fragmento administrado por Visual Studio.
partial class A
{
    private int i;

    public A()
    {
        this.i = 1111;
    }
}
```

Si tuviésemos que añadir un nuevo campo dentro de nuestro fragmento, para uso propio, ya no podríamos inicializarlo en el constructor, pues no podemos tocar el constructor. Si para inicializar el campo nos bastase una expresión de inicialización, no habría mayor problema. Tenga en cuenta, no obstante, que las expresiones de inicialización de campos de instancia tienen unas cuantas limitaciones. Por ejemplo, no se puede inicializar un campo de instancia mediante una llamada a un método de instancia de la misma clase.

La versión 3 de C# ofrece una solución a este problema: los *métodos parciales*. Es una solución "parcial", para las dificultades con la inicialización, pues requiere colaboración en los dos fragmentos, pero hay que reconocer que su alcance es mucho mayor. Además, la colaboración no implica un coste adicional en tiempo de ejecución, como veremos enseguida.

La idea es muy simple: igual que se puede declarar dos fragmentos de una clase mediante el modificador **partial**, podemos también declarar dos fragmentos de un mismo método. La diferencia es que sólo podemos incluir una implementación del método en uno de los fragmentos:

```
// Fragmento administrado por Visual Studio.
partial class A
{
    partial void OnCreate();

    public event EventHandler Evento;

    public A()
    {
        OnCreate();
        ⋮
    }
}

// "Nuestro" fragmento
public partial class A
{
    private void Respuesta(object sender, EventArgs e)
    {
        ⋮
    }

    partial void OnCreate()
    {
```

```
            this.Evento += Respuesta;
        }
    }
```

En el ejemplo anterior, el primer fragmento *anuncia* la posible existencia de un método llamado *OnCreate*, cuyo cuerpo no se incluye en el anuncio. No obstante, el constructor de la clase se permite el lujo de ejecutar dicho método, que puede existir o no en otro fragmento. Si por esas casualidades nadie diese vida a *OnCreate*, la llamada a *OnCreate* simplemente no generaría código alguno. ¿Adivina cuál es una de las restricciones de los métodos parciales? Efectivamente: su tipo de retorno debe ser **void** (¿por qué?). En este ejemplo, por el contrario, el segundo fragmento proporciona un cuerpo a *OnCreate*, y dentro de ese cuerpo aprovechamos para enganchar un manejador a un evento. Este es un tipo de inicialización que, por cierto, no se podría conseguir con inicializadores de campos.

Naturalmente, podemos ejecutar métodos parciales no sólo desde un constructor, sino dentro de cualquier otro código de la misma clase. Observe que el método parcial *OnCreate* se declara sin el modificador de acceso: se trata de un asunto interno de la clase parcial, y se asume que el método parcial, si se llega a completar, será privado y por lo tanto, invisible fuera de la clase que lo declara y utiliza.

8

ADMINISTRACION DE MEMORIA

UNA CARACTERÍSTICA COMÚN A .NET y la máquina virtual de Java 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.

Recogiendo la basura

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 un 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*:

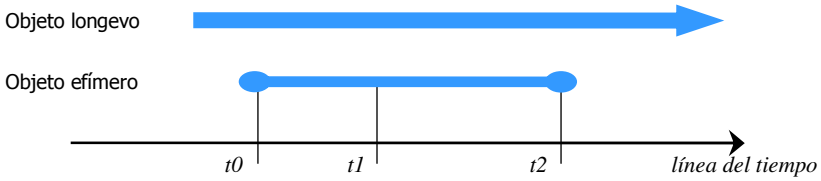
```
private FileStream recurso;
```

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! (si, 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 data 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.

9

GENERICIDAD

SI EN BIOLOGÍA ABUNDAN LOS ESCARABAJOS, en Informática florecen listas de todo tipo: listas en las que es rápido insertar y algo más lento buscar, listas en las que la búsqueda va como un relámpago y las inserciones van como las bajadas de impuestos, listas para amar, listas para odiar… Lo mejor, o lo peor, es que todas estas variedades de listas se subdividen a su vez en listas de enteros, listas de cadenas, listas de valores reales, listas de personas, listas de fechas y así hasta la náusea.

Tipos genéricos

Las diferencias entre una lista de fechas y una de personas son mínimas. Sin embargo, en lenguajes populares como Delphi y Java, y en la primera versión del propio C#, si quisiéramos listas de personas y de fechas, tendríamos que duplicar el trabajo, programando dos clases independientes.

La solución a este problema son los *tipos genéricos*, y se conoce desde hace tiempo gracias a lenguajes como CLU y Ada. Como en muchos otros casos, la idea se popularizó gracias a C++, que implementa una variante *sui generis* de la idea bajo el nombre de *templates* o *plantillas*. El mecanismo de C++ es muy potente, pero está plagado de problemas:

1. En el fondo, es un mecanismo de macro con verificación sintáctica. Muchos problemas no se detectan durante la definición del tipo genérico, sino cuando intentamos crear un tipo concreto a partir del tipo genérico. Cuesta bastante localizar y arreglar estos problemas.
2. Como ocurre con muchas otras características de C++, el formato OBJ no almacena información sobre tipos genéricos. Esto exige que, para reutilizar un tipo genérico, haya que depender de los ficheros de cabecera.

Java ha añadido recientemente tipos genéricos a su repertorio, y su implementación, aunque es más consistente que la de C++, sigue teniendo limitaciones: como la máquina virtual es un fetiche intocable para Sun, la implementación corre a cargo del compilador. En tiempo de ejecución se pierde toda información sobre genericidad, y para rematar, el código generado está plagado de conversiones de tipos en tiempo de ejecución.

Por el contrario, la implementación de este recurso en la plataforma .NET es una joyita digna de enmarcar. Es la propia plataforma la que ha sido modificada para que soporte la genericidad. Con esto, se logran varios objetivos:

1. En principio, todos los lenguajes que se ejecuten sobre la plataforma pueden implementar tipos genéricos, pues más de la mitad del trabajo la tienen ya adelantada. En casos extremos, en los que no merezca la pena complicar el lenguaje o el compilador, un lenguaje puede limitarse a usar tipos genéricos definidos en otros lenguajes, sin permitir definir nuevos tipos.
2. Al contrario de lo que ocurre en Java, los nuevos tipos genéricos son ciudadanos de primera clase de la plataforma. Toda la información relacionada con estos tipos está disponible en tiempo de ejecución, por medio del API de reflexión.
3. Esto, a su vez, permite generar código más eficiente: al contrario de lo que ocurre en Java, no hace falta añadir conversiones ocultas de tipos.

Resumiendo: si no le bastaban las razones para preferir .NET a Java, ahora tiene una más.

Clientes y proveedores de tipos genéricos

Pongámonos de acuerdo con la terminología: llamaremos *cliente* de un tipo genérico al código que utilice un tipo genérico definido en alguna otra parte, y llamaremos *proveedor* al código que define un tipo genérico. Es posible ser, al mismo tiempo, cliente y proveedor de tipos genéricos. Esta distinción es útil para explicar el uso habitual de la genericidad:

- Es muy probable que usted sea cliente de tipos genéricos definidos en otra parte; principalmente, por la propia plataforma.
- Es poco probable que usted define tipos genéricos propios (en circunstancias normales).

La causa de esto es, en gran medida, el propio espíritu de la programación orientada a objetos: escribir una vez, usar muchas veces, que en este caso llega casi al extremo. Este es también el motivo por el que utilizaré un ejemplo bastante visto para mostrarle cómo se define una clase genérica:

```
public class Pila<T>
{
    private T[] elementos;
    private int cantidad;

    public Pila(int capacidad)
    {
        this.cantidad = 0;
        this.elementos = new T[capacidad];
    }

    public void Añadir(T valor)
    {
        elementos[cantidad++] = valor;
    }

    public T Extraer()
    {
        return elementos[--cantidad];
    }

    public bool EstaVacia
    {
        get
        {
            return cantidad == 0;
        }
    }
}
```

Al declarar la clase *Pila*, hemos añadido un parámetro a la declaración y lo hemos llamado *T*. En el resto de la declaración, he resaltado las referencias a dicho parámetro. En todos los casos, *T* se utiliza como si fuese un tipo de datos cualquiera: declaramos un vector de elementos de tipo *T*, inicializamos el vector con el operador **new**, asignamos un parámetro de este tipo en un elemento de vector del mismo tipo, y devolvemos valores de tipo *T*. No importa cuál sea el tipo exacto de *T*, pues todos los tipos soportados por .NET permiten sin excepción las operaciones mencionadas.

A la hora de usar el tipo *Pila*, lo más frecuente es darle un tipo concreto al parámetro *T*:

```
Pila<int> p = new Pila<int>(128);
p.Añadir(1000); p.Añadir(2000); p.Añadir(3000);
Console.WriteLine(p.Extraer());
```

Como puede imaginar, no podemos añadir una cadena a una pila de enteros, y las pilas de diferentes tipos bases no son compatibles.

Antes he dicho "lo más frecuente" porque podemos aprovechar la clase *Pila* postergando la asignación de un tipo concreto. Eso es lo que ocurriría si usáramos la pila dentro de la declaración de otra clase genérica. Por ejemplo:

```
public class Compilador<TipoNodo>
{
    private Pila<TipoNodo> pila;

    ⋮
}
```

En este caso, tenemos un parámetro de tipo al que hemos llamado *TipoNodo* (¡para demostrar que los nombres de parámetros pueden tener más de una letra!) en la clase que se define. Como un parámetro de tipo puede usarse en casi cualquier parte donde pueda usarse un tipo normal, esta vez lo utilizamos para instanciar un caso concreto del tipo genérico *Pila*.

Genericidad y restricciones

Si nos limitamos al comportamiento común a todos los tipos de datos posibles, poca cosa podremos hacer con los tipos genéricos. Para poder ir más lejos, necesitamos exigir condiciones adicionales para los parámetros de tipos. Suponga, por ejemplo, que necesitamos averiguar cuál es el valor mínimo de los elementos almacenados en una pila. Para calcular el valor mínimo necesitamos comparar. ¿Soportan la comparación todos los tipos en .NET? No, porque las comparaciones son odiosas. Si se tratase de comparar para averiguar igual o desigualdad, podríamos aprovechar el método *Equals* que es soportado por todos los tipos en .NET. Pero no es este tipo de comparación la que necesitamos para hallar el mínimo de una colección.

¿La solución? Exijamos que la clase genérica *Pila* sólo se instancie con tipos que soporten la comparación. ¿Qué significa "soportar comparaciones"? Significa que los tipos admitidos tendrán que implementar ciertos métodos. ¿No es eso mismo lo que garantizan los tipos de interfaz? Observe entonces la declaración inicial de la nueva versión del tipo *Pila*:

```
public class Pila<T> where T : IComparable
{
    ⋮
}
```

El tipo *IComparable* se define así:

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

El método *CompareTo* compara dos objetos y debe devolver un valor negativo, si el primer objeto es el menor, un valor positivo si el primero es el mayor, y cero si ambos objetos son iguales. Si los objetos pertenecen a diferentes clases, el método puede quejarse disparando una excepción. Con estas reglas del juego, podemos añadir el siguiente método a la pila:

```
public T Menor()
{
    if (cantidad == 0)
        throw new Exception("¡Pila vacía!");

    T menor = elementos[0];
    for (int i = 1; i < cantidad; i++)
        if (elementos[i].CompareTo(menor) < 0)
            menor = elementos[i];
   return menor;
}
```

A pesar de que sabemos que cualquier tipo que pasemos en *T* soportará *IComparable*, no por ello podemos usar directamente el operador **<**. Tenemos que llamar explícitamente a *CompareTo*. ¿Qué tipos implementan dicha interfaz? Somos afortunados, porque la lista es grande. Por ejemplo, el tipo *System.Int32*, el famoso **int**, implementa *IComparable*. De esta manera, podríamos declarar pilas de enteros, de reales, de tipos lógicos y en general, de todos los tipos que implementen la mencionada interfaz.

Otro detalle: si la pila está vacía, *Menor* lanza una excepción. Esto es lo correcto, pues no existe un valor "menor" dentro de un conjunto vacío.

Incluso podemos mejorar la implementación. El método *CompareTo* de *IComparable* recibe un parámetro de tipo **object**, y eso significa que nos pueden pasar casi cualquier cosa en dicho parámetro, cuando realmente nos interesaría limitar las comparaciones a tipos idénticos o compatibles. Y hay más: si creamos una pila de tipos enteros, por ejemplo, para cada comparación tendríamos que convertir el entero, que es una estructura, en un puntero a objeto. ¿Cómo se llama esta operación? Repita conmigo en voz alta: *boxing*! Y ya sabemos que el *boxing* es una operación costosa que debemos evitar a toda costa…

ICompare es una interfaz heredada de la versión 1.1, antes de la llegada de los tipos genéricos. En .NET 2.0 existe una nueva interfaz genérica, que es la que realmente tenemos que usar:

```
public interface IComparable<T>
{
    int CompareTo(T obj);
}
```

Tome nota de cómo modificamos la restricción de tipo en la declaración de la pila:

```
public class Pila<T> where T : IComparable<T>
{
    ⋮
}
```

Aunque parezca complicado, no hay nada ilegal en este uso de los parámetros de tipos. Por ejemplo, el tipo **int** está realmente declarado de este modo:

```
public struct Int32 : IComparable, IFormattable, IConvertible,
    IComparable<int>, IEquatable<int>
{
    ⋮
}
```

El tipo entero sigue soportando *IComparable* a secas por compatibilidad con .NET 1.1, pero también soporta la versión genérica, que en muchos casos es más eficiente.

NOTA De paso, observe que .NET permite que dos clases tengan el mismo nombre básico si tienen diferente cantidad de parámetros genéricos. Compruebe también que la implementación del método *Menor* no cambia al pasar de *IComparable* a *IComparable<T>*.

Y este es sólo la forma más simple de restricción. Podríamos exigir que el parámetro de tipo implementase más de un tipo de interfaz:

```
public class Pila<T> where T : IComparable, Comparable<T>
```

Podríamos pedir también que el tipo fuese descendiente de alguna clase:

```
public class Pila<T> where T : TipoBase
```

Naturalmente, *TipoBase* no podría ser una clase sellada, porque entonces la genericidad no tendría mucho sentido. Tenemos otros tres tipos de restricciones:

Restricción	Significado
where T: **struct**	El parámetro de tipo debe ser un tipo de estructura.
where T: **class**	El parámetro de tipo debe ser un tipo de referencia.
where T: **new**	El parámetro de tipo debe incluir un constructor público sin parámetros.

Observe que si usamos la primera de estas tres restricciones, la última se convierte en redundante, pues todos los tipos de estructura tienen un constructor público sin parámetros.

Métodos genéricos

En un mundo perfecto, donde no fuese necesario corregir el rumbo en plena marcha, puede que la técnica que voy a explicar ahora no fuese necesaria. Además de declarar tipos genéricos, .NET nos permite programar *métodos genéricos*. Esto es: métodos genéricos dentro de una clase o estructura que no necesariamente tendría que ser genérica. Pongamos por caso que estamos programando una aplicación con Windows Forms que utiliza pilas genéricas. Si vaciásemos pilas con cierta frecuencia, nos interesaría declarar, en una clase nuestra, un método como el siguiente:

```
public static void VaciarPila<T>(Pila<X> pila) where X : IComparable<X>
{
    while (!pila.EstaVacia)
        pila.Extraer();
}
```

Lo mejor sería que fuese la propia clase *Pila<T>* la que implementase *Vaciar* (sin la coletilla de *Pila*). Pero, como he dicho antes, este mundo no es perfecto. El método se ejecutaría de la siguiente manera:

```
Pila<int> p = new Pila<int>(128);
p. Añadir(1000); p.Añadir(2000); p.Añadir(3000);
⋮
VaciarPila<int>(p);
```

Lo interesante, sin embargo, es que esta forma de llamada también es aceptada por el compilador:

```
VaciarPila(p);
```

En este caso, el compilador pone en marcha la *inferencia de tipos*: por el tipo de los parámetros, es posible deducir los tipos que hay que suministrar para los parámetros genéricos de un método.

Cómo funciona la inferencia de tipos

La inferencia de tipos en las llamadas a métodos genéricos es un pequeño regalo que nos hace el compilador. Como ocurre con casi todo regalo, el compilador invierte un esfuerzo razonable en esta tarea, pero no garantiza que siempre sea posible culminarla exitosamente. Hay dos motivos para que la inferencia de tipos no se realice con toda la sofisticación imaginable:

1. El natural equilibrio entre esfuerzos y resultados.
2. Un lenguaje en el que se lleva la inferencia de tipos a sus últimas consecuencias puede ser un lenguaje difícil de dominar, y las aplicaciones desarrolladas con él serán difíciles de mantener.

Para justificar lo anterior, debo explicarle primero cómo funciona la inferencia de tipos en C#. Para simplificar la presentación, utilizaré un ejemplo hipotético. Existe una clase estática *Math* definida dentro del espacio de nombres *System*. Esta clase ofrece una familia o grupo de métodos llamados todos *Min*, que sirven para calcular el mínimo entre dos valores del mismo tipo. Sólo mencionaré un par de ellos:

```
public static decimal Min(decimal d1, decimal d2);
public static long Min(long l1, long l2);
```

Supongamos que la genericidad hubiese estado disponible desde la primera versión de la plataforma. En tal caso, es más que probable que la clase *Math* tuviese un único método llamado *Min*, aunque se trataría de un método genérico:

```
public static T Min<T>(T v1, T v2) where T: IComparable<T>
{
    if (v1.CompareTo(v2) <= 0)
        return v1;
    else
        return v2;
}
```

Supongamos que el compilador tropieza con esta llamada:

```
// ¡No olvide que Math no contiene métodos genéricos!
Console.WriteLine(Math.Min(1, -1));
```

Olvidémonos, de momento, de la existencia de tipos como **byte** o **sbyte**. La inferencia tiene lugar considerando cada argumento por separado: el primer argumento se considera como de tipo entero. Como el primer argumento del método genérico *Min<T>* es de tipo *T*, eso significa que *T* debe ser el tipo **int**. El segundo argumento permite deducir lo mismo. Mezclamos todas las deducciones y vemos si existe alguna incoherencia. No la hay. Comprobamos si podemos entonces aplicar el algoritmo tradicional de resolución de sobrecargas: como hay un único método candidato, hemos terminado exitosamente.

Pongamos ahora un ejemplo ligeramente más complicado:

```
// La inferencia de tipos falla en ejemplos como éste:
Console.WriteLine(Math.Min(1.0, -1));
```

El actual compilador de C# se quejaría y con razón, pues la especificación del lenguaje también lo hace. En el ejemplo anterior, el primer argumento sirve para deducir que *T* es el tipo **double**, pero el segundo argumento exige un tipo entero, y a continuación, el compilador consideraría que se trata de tipos incompatibles, ignorando la existencia de una conversión implícita desde el tipo entero al real. Puede parecernos una limitación tonta, pero ¿imagina el caos de inferencias que podría provocar la inclusión de conversiones, en general, en el algoritmo descrito? Lo más importante es que tenemos una salida para casos como éste. Nos bastaría con decir claramente lo que queremos decir:

```
// Esta es la única salida:
Console.WriteLine(Math.Min<double>(1.0, -1));
```

Este ha sido un ejemplo sencillo, pero el algoritmo de inferencia detecta parámetros genéricos enterrados dentro de vectores y tipos genéricos cerrados. Por ejemplo, la clase *Array* tiene el siguiente método estático:

```
public static int IndexOf<T>(T[] array, T value);
```

El siguiente fragmento realiza una llamada con inferencia de tipos al método anterior:

```
int[] vector = new int[] { 1, 2, 3, 4 };
Console.WriteLine(Array.IndexOf(vector, 1));
```

Es cierto que, en este ejemplo concreto, el segundo parámetro basta para deducir que *T* debe ser el tipo entero. Pero lo que nos importa es que la misma deducción tiene lugar a través del primer parámetro. En este caso, hay que ejecutar un algoritmo de *unificación* de tipos.

Nudismo y covarianza

Existe un tipo de restricción en los parámetros de un tipo genérico, algo extraña a simple vista, que en inglés se conoce como *naked constraint*, o *restricción desnuda*. Consiste en pedir que un parámetro de tipo, llamémosle *U*, sea descendiente de otro parámetro genérico, al que llamaremos *T*. Este sería el patrón general de una restricción desnuda:

```
class MiClase<T, U>
    where U: T …
```

Observe que las restricciones desnudas pueden poner en un brete al compilador y al API de reflexión de tipos genéricos. Por ejemplo, el tipo *T* podría ser lo mismo una clase que un tipo de interfaz. Más intrigante aún es saber por qué se permite este tipo de restricción, y para qué se utiliza. Veamos: es posible que necesitemos una clase genérica con dos parámetros de tipos. Digamos que estos parámetros se utilizan para declarar sendos campos en la clase. Con una restricción desnuda, podemos hacer que el segundo campo almacene referencias de clases compatibles con la clase que utilicemos en el primer parámetro… ¿Se da cuenta de que esto ya lo garantiza la regla de la asignación polimórfica?

La clave nos la da la guía del lenguaje de la propia Microsoft. Aunque por uniformidad, podemos usar restricciones desnudas en una clase genérica, el recurso es verdaderamente útil cuando se utiliza con métodos genéricos. Por ejemplo, podemos usarlo para realizar *asignaciones covariantes* entre contenedores genéricos. Analice el siguiente fragmento de código:

```
string[] strArray = new string[] { "Había", "una", "vez" };
object[] objArray = strArray;
foreach (object obj in objArray)
    Console.WriteLine(obj);
```

Observe, sobre todo, la segunda instrucción: C# nos permite asignar un vector de cadenas a un vector de objetos. Más en general, podemos asignar un vector de una clase derivada a un vector de una clase base, siempre que base y derivada sean tipos de referencia.

El truco, no obstante, introduce algunos problemillas por la puerta trasera;

```
string[] strArray = new string[] { "Había", "una", "vez" };
object[] objArray = strArray;
// ¡Esto no debería permitirse…
objArray[0] = new Buttton();
// … para evitar desastres como el siguiente!
Console.WriteLine(strArray[0].Length);
```

La tercera instrucción es una receta para el desastre: hemos asignado un objeto donde *debería* ir una cadena. Sin embargo, ¿se le ocurre alguna razón por la que el compilador debería rechazar la tercera instrucción? Recuerde que la verificación de tipos es una operación básicamente *local*. Si permitimos que se ejecute impunemente la tercera instrucción, el desastre llegaría al ejecutarse la cuarta: intentamos aplicar la propiedad *Length* sobre un objeto que no la soporta.

Hay lenguajes, como Eiffel, que intentan resolver problemas como el anterior extendiendo el alcance de la verificación de tipos más allá del ámbito meramente local. Si el CLR adoptase esta política, el análisis tendría que ejecutarse en tiempo de carga, pues la adición dinámica de cualquier ensamblado podría invalidar la seguridad del proceso. La solución adoptada por Microsoft, por consiguiente, ha sido añadir una verificación en tiempo de ejecución para las asignaciones a los elementos de un vector. La tercera instrucción sería aceptada por el compilador, pero provocaría un error en tiempo de ejecución.

NOTA

El prefijo de instrucción **readonly**, del Lenguaje Intermedio (IL), sirve para marcar referencias a elementos de vectores como de sólo lectura. De esta manera, el compilador JIT puede evitar la inclusión de la verificación en tiempo de ejecución en muchos casos, que de otra manera serían complicados de detectar durante la traducción a código nativo.

¿No sería más sencillo prohibir la compatibilidad covariante entre vectores de tipos de referencia? No me lo parece, pero además, al CLR no le queda otra opción, pues Java permite ese tipo de asignaciones, con una implementación muy parecida.

En principio, se puede imaginar una regla de compatibilidad parecida para los tipos genéricos. Imagine que, en vez de vectores, quisiéramos usar listas genéricas:

```
List<string> strList = new List<string>(
    new string[] { "Había", "una", "vez" });
// ¡Esto no se permite!
List<object> objList = strList;
```

A pesar de que la asignación covariante "tiene sentido" (Eiffel la permite), C# la prohíbe. Lo interesante es que el CLR permite, al menos en teoría, este tipo de compatibilidad, mediante opciones disponibles al definir el tipo de datos.

… y en este punto, regresamos al tema de la sección: ¿qué ocurre si tenemos una lista de cadenas y necesitamos manejarla como una lista de objetos? Si la operación no se ejecuta con demasiada frecuencia, quizás podríamos crear una nueva lista con el tipo adecuado y transferir a ella los elementos almacenados en la lista original. Como los elementos pertenecerían a tipos de referencia, en realidad sólo estaríamos duplicando la memoria del contenedor, no la de los elementos. La

dificultad estaría en evitar la repetición innecesaria del código de copia. Y es aquí donde nos pueden ser útiles las *restricciones desnudas* que mencionábamos al inicio de la sección. Analice el siguiente fragmento de código:

```
public static class CovariantConverter
{
    public static List<T> Convert<S, T>(List<S> source)
        where S: T
    {
        List<T> result = new List<T>(source.Count);
        foreach (S value in source)
            result.Add(value);
        return result;
    }
}
```

La clase estática *CovariantConverter* define un método genérico *Convert*, con dos parámetros de tipos. El primero de ellos indica el tipo de elemento de la lista original, y el segundo corresponde al tipo de elemento de la lista resultado. Observe que añadimos una restricción desnuda para relacionar ambos parámetros. ¿Para qué la necesitamos? La respuesta está en la llamada al método *Add* sobre la lista de destino. De no ser por la restricción, dicha llamada no sería aceptada por el compilador. Con la ayuda del método genérico anterior, podemos remedar el funcionamiento de una asignación covariante entre listas:

```
List<string> strList = new List<string>(
    new string[] { "Había", "una", "vez" });
List<object> objList =
    CovariantConverter.Convert<string, object>(strList);
```

Note que no podemos usar inferencia de tipos para omitir los argumentos de tipo de la llamada a nuestro método *Convert*. Parte de la información necesaria no se encuentra en la propia llamada, sino en el lado izquierdo de la asignación.

NOTA

La especificación formal de C# 2 se incluyó en el libro *The C# Programming Language*, de Hejlsberg, Wiltamuth y Golde, mucho antes del estreno oficial de la versión 2.0 de la plataforma. Cuando se produjo el estreno, las extensiones al lenguaje habían cambiado bastante. Por ejemplo, en la sección sobre restricciones genéricas, se prohibía explícitamente el uso de restricciones desnudas, que sí son admitidas por la versión final. Naturalmente, no hay motivo para alarmarse por anécdotas como ésta. Sólo la menciono porque es una de esas ocasiones en las que vemos lo que ocurre antes del estreno, tras las cortinas del escenario.

Genericidad y tipos anidados

Como dije al presentar la genericidad, en la práctica es mucho más frecuente que utilicemos tipos genéricos ya definidos, que tengamos que definir nuestras propias clases y estructuras genéricas. Por este motivo, he utilizado una pila, uno de los tipos más manoseados, como ejemplo de clase genérica. .NET, en realidad, ha hecho bien los deberes, y nos ofrece una amplia gama de clases genéricas para las estructuras de datos más solicitadas. Esta es una de mis favoritas:

```
public class Dictionary<TKey,TValue> :
    IDictionary<TKey,TValue>,
    ICollection<KeyValuePair<TKey,TValue>>,
    IEnumerable<KeyValuePair<TKey,TValue>>,
    IDictionary, ICollection, IEnumerable,
    ISerializable, IDeserializationCallback
```

No se asuste por la larga lista de interfaces implementadas… aunque tampoco es difícil descifrar de qué se trata con un poco de paciencia. He traído la clase *Dictionary* a colación no sólo por su innegable utilidad, sino también para mostrarle una declaración con dos parámetros genéricos.

De todos modos, quiero mostrarle la implementación de alguna clase genérica un poco más compleja que las tontas pilas. ¿Qué tal si implementamos un árbol binario ordenado? La novedad del árbol es que, en realidad, necesitaremos dos clases diferentes: una para representar los nodos, y otra

para encapsular el conjunto de nodos en el árbol, propiamente hablando. Vamos a declarar la clase de los nodos, a la que llamaremos *Node*, dentro de la clase del árbol, al que bautizaremos *BinaryTree*:

```
namespace IntSight.DataStructures
{
   public class BinaryTree<T> where T : IComparable<T>
   {
      protected class Node
      {
         ⋮
      }

      ⋮
   }
}
```

He situado la clase del árbol dentro de un espacio de nombres hipotético para poder presentar los nombres completos de las clases, que en este caso serían:

```
IntSight.DataStructures.BinaryTree[T]
IntSight.DataStructures.BinaryTree[T].Node
```

Observe que la clase de los nodos "hereda" el parámetro genérico de la clase contenedora. La palabra "herencia" no es muy apropiada en este contexto, pero espero que transmita la idea correctamente. Un error frecuente sería, por ejemplo, definir un segundo parámetro genérico para la clase *Node*.

En realidad, esta idea de transmisión de los parámetros genéricos es fruto de las manipulaciones del compilador de C#. Las clases reales definidas de cara al *Common Language Runtime* serían:

```
IntSight.DataStructures.BinaryTree`1
IntSight.DataStructures.BinaryTree`1+Node
```

Aquí puede ver el convenio usado por la plataforma para las clases genéricas: al nombre de la clase se añade el acento grave o inverso, y luego, el número de parámetros genéricos de la clase. En el caso de la clase anidada, el nombre de la misma se separa del nombre de la clase contenedora por un carácter +. El compilador es el encargado de adecentar este convenio, utilizando puntos tanto para separar espacios de nombres como clases.

La implementación de la clase *Node* discurría por estos cauces:

```
protected class Node
{
    public T value;
    public Node left;
    public Node right;

    public Node(T value)
    {
        this.value = value;
    }

    public Node Contains(T value)
    {
        int result = value.CompareTo(this.value);
        if (result == 0)
            return this;
        else if (result < 0)
            return left == null  ? null : left.Contains(value);
        else
            return right == null ? null : right.Contains(value);
    }
}
```

Al tratarse de una clase anidada no pública, no hay que preocuparse excesivamente por encapsular cada uno de sus campos. He preferido dar acceso directo a los tres campos que existirán en cada

nodo: un valor, declarado mediante el parámetro de tipo, y referencias a los árboles a la izquierda y a la derecha del nodo. Observe estas declaraciones con cuidado:

```
public Node left;
public Node right;
```

No hace falta incluir parámetros de tipo en esta declaración, porque se trata de una abreviatura de la declaración completa:

```
public BinaryTree<T>.Node left, right;
```

EJERCICIO PROPUESTO

Para no complicarme demasiado, la implementación del método *Contains* en la clase *Node* es recursiva. Puede experimentar transformando esta búsqueda en una operación iterativa.

La clase *BinaryTree* contendrá un puntero al nodo raíz del árbol, que será nulo cuando el árbol esté vacío. La búsqueda se implementará delegando en el método *Contains* de *Node*, y la parte más complicada, al menos relativamente, será la adición de nuevos valores al árbol:

```
public class BinaryTree<T> where T : IComparable<T>
{
   // … Aquí se declararía la clase anidada Node …

   protected Node root;

   public void Add(T value)
   {
      Node parent = null, current = root;
      while (current != null && current.value.CompareTo(value) != 0)
      {
         parent = current;
         current = value.CompareTo(current.value) < 0 ?
            current.left : current.right;
      }
      if (current == null)
      {
         current = new Node(value);
         if (parent == null)
            root = current;
         else if (value.CompareTo(parent.value) < 0)
            parent.left = current;
         else
            parent.right = current;
      }
   }

   public void Add(params T[] values)
   {
      foreach (T value in values)
         Add(value);
   }

   public bool Contains(T value)
   {
      return root != null && root.Contains(value) != null;
   }
}
```

Observe que tenemos dos versiones sobrecargadas de *Add*. La segunda de ellas nos permitirá añadir varios nodos en una sola operación. No pierda esta clase de vista, porque la necesitaremos un par de veces más en esta introducción. A partir de este punto, dejaré de resaltar en amarillo los parámetros de tipo, para no abusar de su vista y de su paciencia.

Tipos anulables

La adición de tipos genéricos a la plataforma .NET ha hecho posible la aparición de los *tipos anulables*, o *nullable types*: un recurso que puede sernos muy útil para trabajar con los valores nulos

característicos de las bases de datos relacionales. Esta aplicación, sin embargo, no es la única posible para estos tipos.

Imagine que tenemos una tabla en SQL Server, con una columna de tipo byte. ¿Cuántos valores diferentes podemos representar mediante dicha columna? Si la columna no admite valores nulos, son 256 valores; en caso contrario, hay que sumar un nuevo valor, el nulo, con lo que llegaríamos a 257 valores diferentes. Si quisiéramos leer el valor de esa columna y almacenarlo en un campo de una clase escrita en C# no podríamos usar simplemente un campo de tipo **byte**. Tendríamos que usar dos campos diferentes: uno, probablemente de tipo **bool**, para indicar si hemos leído un valor nulo de SQL o no, y el campo de tipo byte ya mencionado.

Manejar dos campos para cada columna puede complicar el código fuente y dificultar su mantenimiento, pero los problemas gordos sólo acaban de llegar. Los valores nulos de SQL se interpretan como "información desconocida". Existen reglas muy claras que establecen lo que debe ocurrir si intentamos usar un valor nulo en una expresión. Por ejemplo, si sumamos *x* a un valor nulo, el resultado debe ser también nulo, sin importar el valor concreto de la *x*. En efecto, si depositamos un euro en una cuenta bancaria cuyo saldo desconocemos, ¿cuál será el nuevo saldo? Desconocido, evidentemente.

Volvamos a la aplicación escrita en C#, y supongamos que tenemos que sumar el valor de dos campos, a los que llamaremos *x* e *y*, cuyos valores han sido extraídos de una base de datos:

```
private decimal x, y;
```

Supongamos también que las correspondientes columnas admitían valores nulos, por lo que hemos asociado un par de campos lógicos a los campos ya mencionados:

```
private bool nx, ny;
```

El valor de *nx* será verdadero cuando la columna asociada al campo x contenga un nulo, y algo parecido ocurrirá con el par formado por *ny* e *y*. La pregunta del millar de liras falsas: ¿cómo puedo implementar la suma de estos dos pares de campos, de manera que tengan en cuenta los valores nulos? Esta sería la respuesta:

```
private bool nz;      // Nos dirá si el resultado es nulo o no.
private decimal z;    // La suma, si el resultado no es nulo.

if (nx || ny)
    nz = true;
else
{
    nz = false;
    z = x + y;
}
```

No es tan complicado como la Mecánica Cuántica… pero es pesado de escribir y complicado de mantener. ¿Y si le pidiese que calculase las raíces de una ecuación de segundo grado, teniendo en mente la posibilidad de que los coeficientes de la ecuación contengan nulos?

Los tipos anulables resuelven la mayor parte de estas dificultades. La limitación de nuestro amigo, el tipo **byte**, era que sólo podía almacenar 256 míseros valores, ¿no? Le hacemos beber batido de kriptonita y lo convertimos en un tipo anulable:

```
private byte clark;      // Superman sin capa y con gafas.
private byte? superman; // Clark Kent con capa y sin gafas.
```

El signo de interrogación tras el nombre del tipo confiere superpoderes a éste. En realidad, se trata de una notación abreviada para esta declaración:

```
private System.Nullable<byte> superman;
```

Tras los presuntos superpoderes, se escondía un tipo genérico: la estructura *System.Nullable<T>*, cuya declaración se parece a esto:

```
public struct Nullable<T> where T : struct
{
    private bool hasValue;
    internal T value;
    ⋮
}
```

Como puede ver, *Nullable* encapsula un valor del tipo necesario más un campo lógico que indica si se trata de un valor nulo o no. Hay otros dos detalles muy importantes en el fragmento anterior:

1. *Nullable* sólo admite tipos con semántica de valor para su parámetro genérico. No podemos declarar *Nullable<String>*, o su equivalente *String*?, porque *System.String* es una clase.
2. El propio tipo *Nullable* es una estructura.

Y hay que destacar otro detalle, aunque esta vez la declaración no nos da pistas:

3. En realidad, el parámetro genérico de *Nullable* rechaza también los tipos anulables, aunque sean tipos de valor. Esto es, aunque se permite *Byte*?, no se permite *Byte*??.

¿Por qué no se admiten tipos de referencia? La respuesta es que no necesitan kriptonita. La kriptonita añadía un nuevo valor al dominio original del tipo, ¿no es así? Pues bien, los tipos de referencia ya cuentan con un valor especial adicional: ¡el valor **null**, es decir, el puntero nulo! Si necesitamos leer una columna de tipo cadena en un campo o variable de C#, podemos adoptar el convenio de representar los valores nulos de SQL como punteros nulos en C#. Sólo los tipos de valor necesitan el "módulo de expansión" que implementa *System.Nullable*.

¿Por qué *Nullable* es una estructura? Fácil: si *Nullable* fuese una clase, en vez de una estructura, tendríamos que reservar memoria dinámica para sus instancias. Como los tipos bases admitidos por *Nullable* no requieren memoria dinámica por ser tipos de valor, *Nullable* introduciría un coste innecesario en memoria y en tiempo de ejecución.

No me pregunte, en cambio, por qué no se puede aplicar *Nullable* a un tipo *Nullable*. Supongo que la definición e implementación de las operaciones con estos tipos se complicaría en caso de que fuese posible. Pero sólo es una conjetura personal…

Operaciones con tipos anulables

Veamos qué podemos hacer con los tipos anulables. Comencemos por las asignaciones:

```
int? i = 1234;
int? j = null;
```

A una variable de un tipo anulable, podemos asignarle un valor del tipo original, lo cuál no es extraño, pero también permiten la asignación de un puntero vacío, lo cuál si resulta raro. En ambos casos, el compilador traduce los valores a la derecha de la asignación en llamadas a los constructores de la clase *Nullable*:

```
Nullable<int> i = new Nullable<int>(1234);
Nullable<int> j = new Nullable<int>();
```

Lo que no podemos hacer es asignar directamente un valor anulable en una variable del tipo original:

```
int? i = 1234;
int j = i;          // ¡Esto no es aceptado por el compilador!
```

La forma más fácil de extraer la información almacenada en una entidad anulable es usar estas dos propiedades, definidas en la clase *Nullable<T>*:

```
public bool HasValue { get; }
public T Value { get; }
```

Por lo tanto, la secuencia de asignaciones que rechazaba el compilador puede escribirse de esta manera:

```
int? i = 1234;
⋮
int j = i.Value;
```

En este ejemplo, estamos convencidos de que la variable *i* contiene un valor no nulo. ¿Y si no fuese así? En tal caso, la lectura de la propiedad *Value* provocaría una excepción. En general, si queremos evitar excepciones, tendríamos que combinar el uso de *Value* con *HasValue*, como en este fragmento:

```
int? i = 1234;
⋮
if (i.HasValue)
{
    int j = i.Value;
    ⋮
}
```

El ejemplo anterior tiene un pequeño problema de eficiencia: la implementación de la propiedad *Value* verifica si hay algún valor asignado en la variable… pero se trataría de una verificación por duplicado, al menos en este ejemplo. Por este motivo, existe una función adicional en *Nullable*, llamada *GetValueOrDefault*, que no realiza esta verificación:

```
int? i = 1234;
⋮
if (i.HasValue)
{
    int j = i.GetValueOrDefault();
    ⋮
}
```

La implementación de *GetValueOrDefault* es muy simple:

```
public T GetValueOrDefault()
{
    return this.value;
}
```

Si llamásemos a *GetValueOrDefault* sin comprobar antes el contenido de *HasValue*, y si tuviésemos la mala suerte de que la instancia contuviese un valor nulo… de todos modos obtendríamos un resultado bastante sensato: el valor por omisión correspondiente al tipo original. Por ejemplo, si estamos trabajando con enteros, obtendríamos un cero; si se trata de un tipo lógico, obtendríamos el valor falso. En general, obtendríamos el resultado de asignar ceros en cada uno de los bytes de la instancia.

Existe una segunda variante de *GetValueOrDefault*, que nos permite elegir el valor por omisión:

```
public T GetValueOrDefault(T valor_por_omisión);
```

Con esta función podríamos escribir código como el siguiente:

```
int? i = 1234;
⋮
int j = i.GetValueOrDefault(-1);
```

Parafraseando la última instrucción: dame el valor almacenado en *i*, si es que existe, para asignarlo en *j*, y si el valor es nulo, asignemos *-1*. Sin embargo, existe una alternativa mejor al uso de esta variante de *GetValueOrDefault*: el llamado *operador de fusión* (*coalescence operator*), representado mediante dos signos de interrogación consecutivos:

```
int j = i ?? -1;
```

Se trata de un operador binario que devuelve el valor de su primer operando si éste resulta ser no nulo. En caso contrario, devuelve el valor del segundo operando. El segundo operando puede ser de tipo anulable o no. En el ejemplo anterior, el segundo operando es una constante entera, por lo que el resultado del operador siempre será un tipo entero no anulable.

¿Por qué es preferible el operador de fusión al método *GetValueOrDefault*? La ventaja del operador de fusión es que no está obligado a evaluar su segundo operando si encuentra que el primero no devuelve un nulo. *GetValueOrDefault*, como todo método con parámetros, exige que se evalúe cada uno de sus parámetros antes de ser ejecutado. En nuestro ejemplo, el segundo operando era una constante, pero tenga presente que podríamos escribir una expresión compleja en su lugar.

NOTA

El operador de fusión puede usarse también con tipos de referencia. Se evaluaría el primer operando, y si el resultado no es un puntero nulo, se devuelve este valor. En caso contrario, se devolvería el resultado de la evaluación del segundo operando.

Promoción de operadores

Las novedades relacionadas con los tipos anulables no se detienen aquí. ¿Recuerda que nos quejábamos de lo difícil que resulta imitar el comportamiento de las expresiones que contienen nulos en SQL? Los tipos anulables tienen también presente este problema, y nos echan una mano alterando la semántica de los operadores habituales de C#. La documentación se refiere a los operadores que trabajan con operandos anulables con el nombre de *lifted operators*, que podríamos traducir más o menos como *operadores promovidos*. Las reglas son sencillas, pues son las mismas que en SQL: en una expresión, si alguno de sus operandos es nulo, toda la expresión se anula.

```
int? ni = 1000;
int? nn = null;

ni = ni + 1;
Console.WriteLine(i);
ni = ni * nn;
Console.WriteLine(i);
```

La suma del ejemplo anterior involucra a un tipo anulable y una constante entera. Como el operando anulable no contiene un nulo, la operación funciona normalmente y se asigna *1001* en la variable *ni*. En la multiplicación, por el contrario, el segundo operando contiene un nulo, y se asigna un nulo en *ni*. Un detalle: las llamadas a *WriteLine* provocan la evaluación del método *ToString* de la clase *Nullable<T>*. Cuando *ToString* se ejecuta sobre un valor no nulo, devuelve el valor original como cadena. Si aplicamos *ToString* sobre un valor nulo, obtendremos una cadena vacía.

En el ejemplo, en vez de multiplicar directamente por la constante **null**, utilicé una variable que contenía **null**. El compilador permitiría esta asignación:

```
ni = ni * null;
```

Pero se daría cuenta de que, independientemente del contenido de *ni*, la expresión siempre se evaluaría a **null**. La expresión se simplificaría de manera automática, y recibiríamos una advertencia por parte del compilador.

Tenga muy presente que, con estas novedades, C# intenta duplicar la semántica de expresiones de SQL… pero eso no quiere decir que la semántica de expresiones de SQL sea digna de un premio Turing. ¿Qué ocurriría si ejecutásemos el siguiente fragmento de código?

```
int? ni = 1000;
int? nn = null;

if (ni > nn)
    Console.WriteLine("Mayor");
else if (ni < nn)
    Console.WriteLine("Menor");
else if (ni == nn)
    Console.WriteLine("Igual");
else
    Console.WriteLine("Este mundo loco, loco, loco...");
```

Si su respuesta ha sido que algo apesta en Dinamarca, ha acertado. Es una lástima que el concurso no ofreciese premios.

El Conocimiento Secreto

- Recuerde que puede dejar que el compilador infiera los argumentos de tipos en llamadas a métodos genéricos.

  De todos modos, debo advertirle que el abuso de esta característica puede volver ilegible su código. Además, existe el peligro de que el compilador infiera un tipo derivado cuando lo que usted pretendía era usar un ancestro del tipo deducido.

10

ITERACION

CUALQUIER APLICACIÓN MEDIANAMENTE compleja utiliza *contenedores* de variado pelaje con generosidad. Puede tratarse de pilas, listas ordenadas y sin ordenar, árboles de todos los colores e incluso conjuntos de datos o simples vectores. Para todas estas estructuras, es crucial la operación que permite recuperar cada uno de sus elementos de manera secuencial. Aunque una adecuada encapsulación ayuda bastante, C# ofrece una instrucción especial, **foreach**, que simplifica el recorrido sobre los elementos de cualquier contenedor debidamente preparado.

La otra cara de la moneda es cómo *preparar* un contenedor para que soporte la instrucción mencionada. En la primera versión de la plataforma, era una tarea difícil, pues casi siempre había que implementar una máquina de estados. A partir de la segunda versión, es el propio compilador quien se hace cargo de los detalles sucios, gracias a la adición de *iteradores* al lenguaje.

El patrón de enumeración

La instrucción **using** que he presentado antes corresponde a una decisión de diseño nada usual; este autor no conoce nada parecido, al menos en los lenguajes más populares. La singularidad de **using** consiste en que se trata de una adaptación sintáctica para simplificar el uso de un recurso de la biblioteca de clases. Es normal concebir el diseño de un lenguaje de programación mediante capas:

1. En el nivel inferior, la definición lexical: qué es un identificador, cómo se forman las cadenas de caracteres, qué importancia se le da a los cambios de líneas.
2. A partir de los elementos lexicales, se define la sintaxis del lenguaje: una instrucción **if** está formada por una condición, una parte **then** obligatoria y una parte **else** funcional. A su vez, la parte **then** está formada por… Creo que la idea queda clara, ¿no?
3. El lenguaje que hemos definido puede ejecutarse sobre diferentes plataformas. Para cada plataforma se espera que, más allá de una colección pequeña de clases o funciones comunes, surjan bibliotecas de clases o funciones diferentes.

De acuerdo a esta visión estratificada del lenguaje, no sería buena idea vincular entidades del nivel dos con entidades del nivel tres. ¿O sí? Confieso que he hecho trampas, omitiendo un estrato que se ubicaría entre el segundo y el tercero.

2½ Una vez definida la sintaxis, casi siempre mediante una gramática libre de contexto, la especificación del lenguaje debe completarse asignando atributos a las distintas construcciones gramaticales, y exigiendo que se cumplan determinadas reglas relacionadas con estos atributos. Por ejemplo, la expresión que sigue a **if** en una instrucción condicional debe ser de tipo *Boolean*.

¿En qué se diferencia la exigencia de que la condición de un **if** o de un **while** sea de tipo *Boolean* de requerir que el tipo de la variable en una instrucción **using** implemente la interfaz *IDisposable*? La única diferencia es que el tipo lógico es un tipo "predefinido"… pero este concepto de tipo predefinido es muy flexible. C# ha cruzado una línea muy delgada, y creo que el paso está justificado.

Una vez abierta la veda, es lógico esperar más instrucciones vinculadas a determinados tipos de datos. Probablemente, la más popular de ellas sea la instrucción **foreach**, para recorrer todos los elementos de una colección.

```
foreach (Tipo variable in expresión)
   instrucción
```

Olvídese por el momento del tipo de la variable declarada por la instrucción. Lo más importante es determinar qué tipo de expresiones se admiten a la derecha de la palabra reservada **in**. Hay dos posibilidades:

1. Que el tipo de la expresión descienda o implemente el tipo de interfaz *IEnumerable*, definido en el espacio de nombres *System.Collections*. Todos los vectores, con independencia de su tipo base, descienden formalmente de la clase *System.Array*, y ésta implementa *IEnumerable*, por lo que la expresión puede ser un vector de tipo arbitrario.

2. Si vuela como un pato, nada como un pato, anda patosamente (como un pato) y grazna como un pato… ¿qué más da si no es un pato? Quiero decir, que si la expresión no implementa la interfaz *IEnumerable*, pero tiene métodos públicos cuyos prototipos coinciden con los de *IEnumerable*, C# acepta pato por liebre (¿o se trataba de un gato?).

Veamos entonces la declaración del tipo *IEnumerable*:

```
public interface IEnumerable
{
   IEnumerator GetEnumerator();
}
```

Es decir, *IEnumerable* sólo ofrece un método que sirve para obtener un puntero a otro tipo de interfaz, *IEnumerator*, también declarado en *System.Collections*:

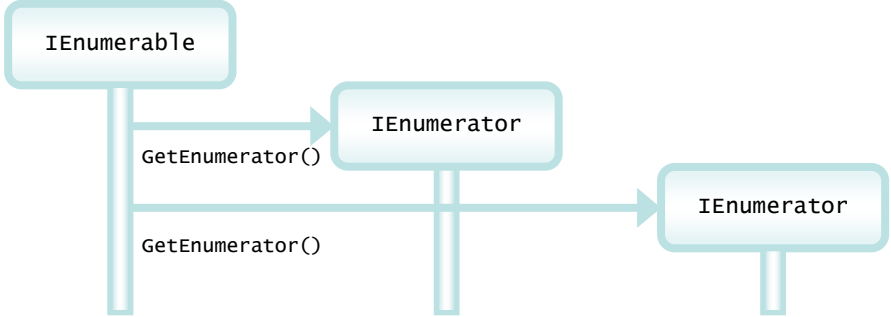
```
public interface IEnumerator
{
   bool MoveNext();
   object Current { get; }
   void Reset();
}
```

Supongamos que la expresión usada en la instrucción **foreach** implementa *IEnumerable*. En tal caso, la instrucción se implementa, a grandes rasgos, de la siguiente manera:

```
IEnumerator enumerador = expresión.GetEnumerator();
while (enumerador.MoveNext())
{
   Tipo variable = (Tipo)enumerador.Current;
   instrucción;
}
```

En realidad, la implementación es un poco más compleja, porque hay que comprobar si la expresión implementa, además de *IEnumerable*, el tipo *IDisposable*. Pero por ahora podemos arreglárnoslas con esta versión simplificada.

¿Por qué se definen dos tipos de interfaz diferentes para definir un recorrido? Gracias a este diseño, es posible realizar varios recorridos simultáneamente sobre una misma colección, porque las estructuras de datos que controlan la iteración se pueden situar en la implementación de *IEnumerator*. Ni siquiera es necesario anidar las iteraciones: cada valor *IEnumerator* que obtengamos ejecutando *GetEnumerator* debe ser independiente de los demás:



Hay otro detalle importante: la propiedad *Current* de *IEnumerator* está declarada con el tipo **object**, y en consecuencia, para asignar su valor en la variable del bucle es necesario realizar una conversión de tipo explícita. Como puede imaginar, esta conversión tiene un coste. Esto se resuelve en .NET 2.0, como veremos más tarde, mediante los nuevos *tipos genéricos*, pero se trata de un lujo que no está disponible en la versión 1.1. ¿Es posible librarnos de esta conversión?

Para resolverlo tendríamos que usar un pato… es decir, el tipo de la expresión en **foreach** debe definir un método público llamado *GetEnumerator*. No es necesario que este *GetEnumerator* devuelva un valor de tipo *IEnumerator*. Llamemos *T* al tipo devuelto por *GetEnumerator*. Este tipo debe declarar un método público y una propiedad pública:

```
public bool MoveNext();
public X Current { get; }
```

La diferencia respecto al *IEnumerator* "ortodoxo" está en el tipo de la propiedad *Current*, que ahora no tiene que ser necesariamente **object**, sino el tipo que el autor de la clase considere apropiado. Si el compilador, al traducir una sentencia **foreach**, descubre este patrón y encuentra, además, que el tipo declarado para la propiedad *Current* coincide con el tipo de la variable del bucle, puede ahorrarse la temida conversión de tipos.

Una pregunta frecuente es si este mecanismo no es demasiado "pesado" para recorrer los elementos de un vector. Pero ocurre que el compilador de C# maneja este caso de forma separada, y traduce una instrucción **foreach** que recorre un vector de la forma más eficiente posible. Es más: hay insistentes rumores de que, al convertir finalmente un bucle sobre un vector en código nativo ejecutable, el compilador JIT introduce unas cuantas optimizaciones para no tener que realizar la verificación de límites de la variable de control. Le advierto que no lo he comprobado en persona, pero *si non é vero, é ben trovato*.

Iteradores

La otra gran novedad en C# versión 2.0 son los *métodos iteradores*. La terminología se las trae: en realidad, la documentación de C# habla de "bloques de iteración". Un bloque de iteración es un bloque que contiene al menos una instrucción **yield return** o **yield break**. Para que estas nuevas instrucciones puedan usarse, el método debe devolver un valor de tipo *IEnumerable*, *IEnumerator* o de los primos genéricos de estos dos tipos. Resumiendo, que a primera vista cuesta trabajo distinguir estos engendros.

Todo esto es, en gran parte, culpa de la vieja sintaxis heredada de los tiempos de C. En Freya, cuya sintaxis es pascaloide, un iterador es un método que se declara mediante la palabra clave **iterator**: de igual manera que un constructor se declara con la palabra clave **constructor**:

```
BinaryTree = class[X]
    ⋮
public
    iterator Nodes: X;
end;
```

En C++ y C#, por el contrario, para distinguir entre un constructor y un método "normal" hay que observar si el método indica un tipo de retorno… aunque este sea **void**. Y un iterador como el anterior tendría que declararse así:

```
public IEnumerable<X> Nodes() { … }
```

Es decir, sin conocer el contenido de *Nodes*, es imposible saber si se trata de un método regular o de un método con bloques de iteración en su interior.

¿Cómo se usan los bloques de iteración? Este es un ejemplo simple, aunque un poco extremo:

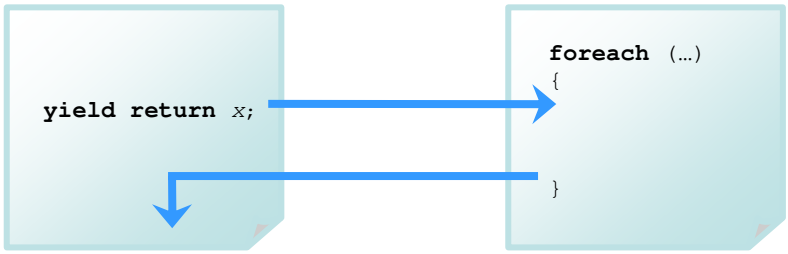
```
public class Iteradores
{
    public static IEnumerable<int> Digitos()
    {
```

```
            yield return 0; yield return 1; yield return 2;
            yield return 3; yield return 4; yield return 5;
            yield return 6; yield return 7;
            yield return 8; yield return 9;
        }
    }
```

He declarado un método *Digitos* estático, para evitar tener que crear una instancia para usarlo, que contiene un bloque de iteración, cuyo valor de retorno es uno de los adecuados. Podemos usar *Digitos* de esta manera:

```
foreach (int i in Iteradores.Digitos())
    Console.WriteLine(i);
```

Este código imprimiría los dígitos decimales, del 0 al 9, uno por línea. Conceptualmente, se puede representar el funcionamiento de la instrucción **yield return** de esta manera:



Para simplificar, he supuesto que el iterador se llama como parte de un bucle **foreach**; luego veremos que no es indispensable. El bucle cedería el control al iterador, y éste ejecutaría sus instrucciones hasta llegar a un **yield return**. El control pasaría temporalmente al cuerpo del bucle. Cuando el bucle pidiese el siguiente elemento, el control regresaría a la instrucción siguiente al **yield return** que interrumpió la ejecución. Hablando en jerga informática, el iterador funcionaría como una *corrutina*.

Por supuesto, usted y yo sabemos cómo funciona la pila de llamadas de un microprocesador y nadie nos convencerá a las buenas de que los iteradores funcionan *exactamente* así. Lo que realmente sucede es que el compilador desmenuza el código del iterador y, a partir de los trozos, monta un método *MoveNext* de una clase creada para uso interno. Sin entrar en demasiados detalles técnicos, la técnica comienza sustituyendo cada **yield return** por la siguiente secuencia equivalente:

```
yield return x;          ⇒          _estado = σi
                                    _current = x;
                                    return true;
                                σi: …
```

Estas instrucciones almacenan el estado interno del iterador en un campo de la clase generada, y asignan el valor de retorno en otro campo, para inmediatamente terminar la ejecución del iterador. Luego, el compilador añade una instrucción **switch** al inicio del iterador, que lee en qué estado se encuentra el iterador para decidir adónde saltar. Esta es sólo la idea básica. Además, el compilador debe determinar cuáles campos de la clase son utilizados dentro del iterador para convertirlos en campos de la clase interna generada. También hay que transformar las variables locales del iterador en campos de la clase interna, para que no se pierdan sus valores en cada paso del bucle.

Veamos ahora un ejemplo más real de iterador. Esta vez se trata de un iterador que devolverá la secuencia de los números de Fibonacci:

```
public static IEnumerable<int> Fibonacci()
{
    yield return 0;
    int i = 0, j = 1;
    while (true)
    {
        yield return j;
        int temp = i;
        i = j;
```

```
            j += temp;
        }
    }
```

Como ve, ¡este iterador ejecuta un bucle infinito! Esto, en sí, no es grave… siempre que advirtamos al cliente del iterador sobre esta característica, pues el responsable de detener la generación de números en algún momento será el cliente. Por ejemplo:

```
foreach (int i in Iteradores.Fibonacci())
    if (i > 1000)
        break;
    else
        Console.WriteLine(i);
```

EJERCICIO PROPUESTO

Intente generar una clase que implemente *IEnumerable* a partir del código del iterador *Fibonacci*. Luego, programe el iterador en una aplicación y compruebe con Reflector o con ILDASM el código que ha generado el compilador. Compárelo con su propia implementación.

¿Qué tal si implementamos un iterador para nuestra clase *BinaryTree*? Voy a mostrarle una implementación que recorre los nodos en orden simétrico: para cada nodo, primero se recorre el árbol de la izquierda, a continuación se visita el valor del propio nodo, y luego se pasa al árbol de la derecha. Como se trata de un árbol binario ordenado, este recorrido devuelve los valores almacenados en su orden correcto.

Esta es la implementación del iterador:

```
public IEnumerable<T> Nodes()
{
    Node current = root;
    Stack<Node> stack = new Stack<Node>();
    while (true)
        if (current != null)
        {
            stack.Push(current);
            current = current.left;
        }
        else if (stack.Count > 0)
        {
            current = stack.Pop();
            yield return current.value;
            current = current.right;
        }
        else
            yield break;
}
```

En realidad, hay dos formas de implementar un iterador de este tipo: o usamos recursividad, o utilizamos una pila auxiliar. El método anterior muestra el uso de una pila. En este caso, para la pila he usado la clase *Stack*, que ofrece .NET de serie.

Composición de iteradores

Para explicar por qué es preferible usar una pila explícita, tenemos que ver cómo sería la implementación recursiva:

```
public IEnumerable<T> NodosRecursivos()
{
    if (left != null)
        foreach (T valor in left.NodosRecursivos())
            yield return valor;

    yield return this.value;
```

```
        if (right != null)
            foreach (T valor in right.NodosRecursivos())
                yield return valor;
    }
```

¿Ve el problema? Cada vez que se ejecute una instrucción **foreach**, se creará una instancia de la clase interna sintetizada por el compilador. El gasto en memoria sería excesivo, y habría que contar también con el coste adicional de las llamadas recursivas. La ventaja de la implementación recursiva, no obstante, es su mayor simplicidad. Es muy fácil equivocarse al programar la iteración sobre un árbol mediante una pila auxiliar. Si no me cree, intente programar iteradores que visiten los nodos en pre-orden y en post-orden.

Métodos anónimos

En los lenguajes que no soportan iteradores (la gran mayoría) es común utilizar otra técnica para recorrer estructuras complejas, como nuestros árboles binarios ordenados. La técnica consiste en definir el iterador como un método que recibe como parámetro un puntero a método. En C#, naturalmente, "puntero a método" debe traducirse como "tipo delegado". Suponga que tenemos una clase que esconde un vector en sus tripas, como la famosa pila:

```
public class Pila<T>
{
    private T[] elementos;
    private int cantidad;
    ⋮
}
```

Podríamos definir un iterador que recorriese todos los elementos de la pila mediante un método como el siguiente:

```
public void Recorrer(Action<T> accion)
{
    for (int i = cantidad; i > 0;)
        accion(elementos[--i]);
}
```

El tipo genérico *Action*, utilizado en el parámetro de *Recorrer*, está declarado en *System*:

```
// Delegado predefinido en System
public delegate void Action<T>(T obj);
```

A este tipo de métodos se les llama *iteradores cerrados*. Por lo general, resulta más sencillo programar iteradores cerrados que iteradores "abiertos": lo comprobaremos enseguida, cuando le muestre cómo programar un iterador cerrado para *BinaryTree*. La mala noticia es que los iteradores cerrados no son tan flexibles como los abiertos:

1. Es más complicado abortar un recorrido con un iterador cerrado. De entrada, tendríamos que ampliar el prototipo del parámetro de acción para que el cliente del iterador avise si desea continuar o no. El delegado *Action* tendría que definirse de forma parecida a ésta:

   ```
   public delegate void Action1<T>(T obj, ref bool abortar);
   ```

   En teoría, podríamos haber hecho que *Action1*, en vez de tener un parámetro adicional, devolviese un valor de tipo lógico. Pero sería una mala decisión: estaríamos obligando al cliente del iterador a devolver siempre "algo" para terminar la acción. Con un parámetro pasado por referencia, podríamos inicializar *abortar* a **true**. El cliente no tendría que tocar este parámetro a menos que realmente desease interrumpir el recorrido.

2. La limitación más importante concierne a las formas en las que podríamos usar el iterador. Imagine que tenemos dos listas ordenadas, y que queremos mezclar sus elementos en una nueva lista que conserve el orden. El algoritmo de mezcla ordenada es un clásico de la Informática: pedimos un elemento de cada lista para compararlos entre sí. El elemento menor se inserta en el resultado, y la lista de donde procede avanza un paso en el recorrido. Este tipo

de recorrido se puede implementar con los iteradores abiertos… ¡aunque no con la instrucción **foreach**! Pero no podemos usar un iterador cerrado en un algoritmo de mezcla ordenada.

Ahora que ya conoce las limitaciones, veamos cómo programaría un recorrido en orden simétrico para un árbol binario ordenado:

```
public class BinaryTree<T> where T : IComparable<T>
{
   public void ForEach(Action<T> action)
   {
      if (root != null) root.ForEach(action);
   }
}
```

Para mantener la terminología usada por C#, he llamado *ForEach* al iterador cerrado. Su implementación comprueba si la raíz es nula o no, para delegar el recorrido en un método de la clase *Node*:

```
public class BinaryTree<T> where T : IComparable<T>
{
   protected class Node
   {
      ⋮

      public void ForEach(Action<T> action)
      {
         if (left != null) left.ForEach(action);
         action(value);
         if (right != null) right.ForEach(action);
      }
   }
}
```

¿Ha visto qué fácil ha sido? Para mostrar cómo se utilizaría *ForEach* necesitamos una "acción": un método que reciba un valor entero y que no tenga valor de retorno. Algo como esto nos serviría:

```
private void ImprimirNumero(int i)
{
    Console.WriteLine(i);
}
```

En realidad, podríamos usar directamente *Console.WriteLine*, pero supondremos que lo que necesitamos hacer con cada nodo no es tan simple como imprimir su valor en pantalla. Para ejecutar el bucle con la ayuda de *ForEach* haríamos lo siguiente:

```
⋮
BinaryTree<int> tree = new BinaryTree<int>();
tree.Add(0, 9, 1, 8, 7, 2, 3, 6, 4, 5);
tree.ForEach(ImprimirNumero);
⋮
```

Aquí ya podemos ver el problema: tenemos, por un lado, una llamada a *ForEach* que hace referencia a un método llamado *ImprimirNumero*. ¿Dónde está definido este método? Por lo general, estará en el quinto pino, fuera de nuestro alcance. Como resultado, analizar este sencillo fragmento de código puede ser bastante pesado.

A partir de su segunda versión, C# alivia nuestros sufrimientos permitiéndonos escribir métodos anónimos. Observe:

```
BinaryTree<int> tree = new BinaryTree<int>();
tree.Add(0, 9, 1, 8, 7, 2, 3, 6, 4, 5);
tree.ForEach(delegate(int value) { Console.WriteLine(value); });
```

La zona resaltada en amarillo corresponde a la definición de un método en toda regla. El método no tiene nombre, y de ahí lo de "anónimo". Solamente declaramos los parámetros, porque el tipo de retorno puede ser inferido por el compilador; en este caso, el método anónimo no devuelve

valor alguno. La gran ventaja es que la llamada a *ForEach* y la acción que ejecutamos para cada nodo están juntas físicamente, y es más sencillo comprender qué hace el fragmento de código.

Como imaginará, el compilador de C# genera métodos internos para implementar los métodos anónimos. El compilador debe detectar si el método utiliza variables locales accesibles desde el punto donde se declara el método. En el ejemplo anterior, el método anónimo sólo utiliza el parámetro del propio método. Pero podemos complicarlo un poco:

```
BinaryTree<int> tree = new BinaryTree<int>();
tree.Add(0, 9, 1, 8, 7, 2, 3, 6, 4, 5);
int total = 0;
tree.ForEach(delegate(int value) { total += value; });
Console.WriteLine(total);
```

Esta vez, el método anónimo lee y escribe el valor de una variable local del método donde es declarado. El compilador lo resuelve creando una clase auxiliar, moviendo la variable local a dicha clase, como un campo, y definiendo el método oculto dentro de esta clase.

Antes mencioné que el compilador puede inferir el valor de retorno del método anónimo. En la práctica, puede hacer mucho más:

```
int count = 0;
tree.ForEach(delegate { count++; });
```

El compilador acepta que omitamos toda la lista de parámetros si no la vamos a utilizar. Observe que no he puesto paréntesis vacíos después de la palabra **delegate**. Si los pusiéramos, el compilador protestaría, porque interpretaría que el método anónimo no debe recibir parámetros.

> **NOTA** Tanto los métodos anónimos como los bloques de iteración son recursos implementados por los compiladores, y no necesitan ayuda alguna por parte de la plataforma. Los tipos genéricos, por el contrario, son responsabilidad del entorno de ejecución.

Expresiones lambda

Los métodos anónimos cobran mayor importancia en la siguiente versión de C#, pues son la base de las *expresiones lambda*, que son a su vez fundamentales para LINQ, el lenguaje de consulta que implementará la versión 3 de C#. El nombre de expresión lambda está tomado del cálculo funcional y de los lenguajes de programación funcionales, y es una señal del interés de los diseñadores de C# por aprovechar ciertas características de estos lenguajes.

Antes de explicar en qué consisten estas expresiones, veamos otro ejemplo en el que intervendrán métodos anónimos. La clase *System.Array*, que es la clase base de todos los tipos de vectores en C#, no es una clase genérica, pero en ella se definen varios métodos estáticos genéricos como el siguiente:

```
public static T[] FindAll<T>(T[] vector, Predicate<T> condicion);
```

FindAll recibe un vector y una condición, y devuelve un nuevo vector con todos los elementos del vector original que satisfacen la condición. La condición se representa mediante un delegado compatible con el siguiente tipo:

```
public delegate bool Predicate<T>(T obj);
```

Esto es: los métodos compatibles con el delegado *Predicate* reciben una instancia y deben devolver un valor lógico, indicando si la instancia cumple con la condición deseada o no. Por ejemplo, puede que nos interese seleccionar los números impares presentes en un vector de enteros:

```
int[] result = Array.FindAll<int>(
    new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
    delegate(int x) { return x % 2 != 0; });
foreach (int i in result)
    Console.WriteLine(i);
```

Como ya hemos visto, en estos casos conviene usar un método anónimo, para que la llamada a *FindAll* y la condición concreta que exigimos estén cercanas espacialmente.

Una expresión lambda es simplemente una nueva manera de escribir métodos anónimos. Gracias a ellas, la llamada a *FindAll* en el ejemplo anterior se podrá simplificar de esta manera:

```
int[] result = Array.FindAll(
    new[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }, x => x % 2 != 0);
```

He resaltado en amarillo la expresión lambda. Por una parte, tenemos que teclear menos, y eso es siempre de agradecer. Pero lo principal es que ahora es más claro el objetivo de la llamada… o eso me parece a mí.

Para entender cómo ha sido posible tal simplificación, veamos los pasos por los que ha evolucionado la expresión de nuestro ejemplo:

```
1  delegate(int x) { return x % 2 != 0; }
2  (int x) => { return x % 2 != 0; }
3  (int x) => x % 2 != 0
4  (x) => x % 2 != 0
5  x => x % 2 != 0
```

La primera expresión corresponde a un método anónimo, completamente legal desde el punto de vista de C# versión 2.0. La primera mutación es puramente sintáctica, y su resultado es la segunda expresión. Hemos cambiado la palabra clave **delegate** por el símbolo => y luego hemos intercambiado la posición de este símbolo con la de la lista de parámetros del método anónimo.

El siguiente cambio es igual de radical, y afecta al bloque de instrucciones. Cuando se trabaja con métodos anónimos, es muy frecuente que el bloque de instrucciones sólo contenga una instrucción, y que ésta instrucción sea un **return** seguido de una expresión. El cambio ha consistido en eliminar las llaves que delimitaban el bloque, porque al existir una sola instrucción se vuelven superfluas, a la vez que eliminamos la palabra clave **return** y el terminador de instrucciones; es decir, el punto y coma. El resultado es la tercera expresión de la secuencia.

La tercera transformación va más allá de lo meramente sintáctico: el compilador de C# 3.0 permite inferir los tipos de parámetros de una expresión lambda… siempre que sea posible, claro. En nuestro caso, lo es. El parámetro de tipo con el que instanciamos el método genérico *FindAll* tiene que ser obligatoriamente el tipo entero, pues el primer parámetro de *FindAll* es un vector de valores enteros. Seamos sinceros: al presentar el ejemplo, hemos establecido explícitamente el parámetro de tipo en *FindAll*, pero esto es innecesario incluso en C# 2.0. ¡Compruébelo!

En consecuencia, podemos eliminar el tipo en la declaración de parámetros de la expresión lambda. Como sólo tenemos un parámetro en esa declaración, y su tipo es inferido por el compilador, podemos dar el último paso de la metamorfosis y eliminar los paréntesis alrededor de la *x*:

```
x => x % 2 != 0
```

Podemos "leer" la expresión anterior de la siguiente manera:

- Función que recibe un parámetro *x*, y devuelve **true** si *x* es un número impar.

Al presentar la forma final de la llamada a *FindAll*, apliqué otras dos simplificaciones. Ya he mencionado una de ellas: podemos dejar que el compilador infiera el parámetro de tipo de la llamada a *FindAll*, pues la llamada tiene lugar sobre un vector de enteros. La segunda simplificación corresponde a un nuevo truco de C# 3.0 relacionado con las expresiones de creación de vectores:

```
new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }      // Antes
new[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }          // Ahora
```

En la nueva versión del lenguaje, no hace falta indicar que se trata de un vector de enteros, pues todos los elementos del vector pertenecen, sin lugar a dudas, a este tipo.

¿Equivalencia estructural?

¿Cuál es el tipo de datos de una expresión lambda? Evidentemente, se trata de un tipo delegado pero ¿cuál tipo, en concreto? Por suerte, existe bastante flexibilidad en la atribución de tipos delegados a expresiones. Imagine que define estos dos tipos, aparentemente equivalentes:

```
public delegate void MyHandler1(object sender, EventArgs e);
public delegate void MyHandler2(object sender, EventArgs e);
```

Necesitamos un método con un prototipo compatible, aunque sea trivial:

```
public void AHandler(object sender, EventArgs e)
{
}
```

Con las piezas presentadas, podemos ejecutar estas dos asignaciones sin mayor problema:

```
MyHandler1 eh1 = AHandler;
MyHandler2 eh2 = AHandler;
```

Sin embargo, el compilador se queja si intentamos lo siguiente:

```
eh1 = eh2;
```

Ni siquiera con una conversión explícita de tipos, el compilador se mostraría dispuesto a pasar por el aro. Y no intente el truco sucio de convertir primero al tipo común *Delegate* para luego transformar el delegado en el tipo del lado izquierdo: el compilador le mirará de reojo y no dirá nada, pero será el entorno de ejecución el que protestará ruidosamente.

Lo que ocurre es que *AHandler*, sin los paréntesis necesarios para su ejecución, no es una verdadera expresión, con todos sus derechos y deberes, sino un truco sintáctico, que se atiene a otras reglas. Esta instrucción, por ejemplo, sería incorrecta:

```
AHandler.BeginInvoke(null, null, null, null);
```

En contraste, esta instrucción sí es aceptada:

```
((MyHandler1)AHandler).BeginInvoke(null, null, null, null);
```

Si lo prefiere, puede considerar que *AHandler*, a secas, pertenece a un tipo de datos cocido sólo a medias. Mientras no termine su cocción, es compatible con cualquier tipo delegado con un prototipo compatible. Pero eso no significa que exista equivalencia estructural entre los tipos delegados… o si lo prefiere, entre tipos delegados debidamente cocinados.

Volviendo a las expresiones lambda, es fácil ver que éstas pertenecen a tipos medio cocidos, y que son compatibles con cualquier tipo delegado con prototipo compatible. En particular, .NET define toda una familia de tipos delegados genéricos en el espacio *System*:

```
public delegate TResult Func<TResult>();
public delegate TResult Func<T1, TResult>(T1 arg1);
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);
```

… y así sucesivamente, hasta llegar a cuatro parámetros de entrada. Si necesita guardar una referencia a una expresión lambda, puede echar mano de estos tipos predefinidos:

```
Func<int, bool> esPar = x => x % 2 == 0;
```

En este ejemplo, la expresión recibe un entero, y devuelve un valor lógico que indica si el entero es o no un número par.

Árboles de expresiones

El tipo *Func* es, al fin y al cabo, un tipo delegado. ¿Cree posible transmitir un delegado, que no es más que un puntero a un método, a través de una conexión remota? Si quiere programar aplicaciones de bases de datos en varias capas, la transmisión remota de condiciones, de una manera u otra,

es indispensable. ¿Sería posible guardar una copia de un delegado de una sesión a otra, utilizando persistencia? Se me ocurren varias ideas, pero todas son algo barrocas… y poco estables. ¿Y si necesitamos "sintetizar" una función nueva en tiempo de ejecución? Es cierto que podemos utilizar el API de emisión de código, pero es una técnica de muy bajo nivel.

Existe una alternativa a guardar una referencia a una expresión lambda en un delegado: convertir la expresión en un árbol de expresiones. Esta vez, el tipo que necesitamos es una clase genérica, definida en el espacio *System.Linq.Expressions*:

```
public class Expression<TDelegado> { … }
```

En teoría, el lenguaje no ofrece ningún recurso general para convertir un tipo delegado en una clase arbitraria, como *Expression*. Es el compilador el encargado de tratar estas asignaciones especiales de manera diferente. He aquí un sencillo ejemplo:

```
Expression<Func<int, bool>> esPar = x => x % 2 == 0;
Func<int, bool> fnPar = esPar.Compile();
if (fnPar(2) && ¡!fnPar(3))
    Console.WriteLine("El cielo NO se va a caer.");
```

El fuego de artillería se concentra en la primera instrucción, pues el compilador traduce esa simple línea en código equivalente al siguiente:

```
ParameterExpression px =
    Expression.Parameter(typeof(int), "x");
Expression body = Expression.Equal(
    Expression.Modulo(px, Expression.Constant(2, typeof(int))),
    Expression.Constant(0, typeof(int)));
Expression<Func<int, bool>> esPar =
    Expression.Lambda<Func<int, bool>>(body,  px);
```

Como ve, el compilador construye un árbol a partir de la expresión lambda. Los nodos del árbol son fácilmente reconocibles en la expresión original.

Una vez construido el árbol, para ejecutarlo debemos llamar al método *Compile*, que nos devuelve un valor de tipo delegado:

```
Func<int, bool> fnPar = esPar.Compile();
```

Observe que el tipo exacto del delegado corresponde al parámetro de tipo especificado al declarar la expresión.

11

REFLEXION

ES OPINIÓN GENERAL ENTRE QUIENES se dedican a la Inteligencia Artificial que la consciencia humana está relacionada, de algún modo, con la capacidad autorreferencial. Esto es, simplificando un poco, que todo sistema consciente debe poder emitir opiniones o diagnósticos sobre su propio funcionamiento. La autorreferencialidad sería una característica necesaria, aunque no suficiente, para la inteligencia.

Sistemas hipocondríacos

Por este motivo, desde que surgieron los primeros lenguajes de programación se intentó que algunos de ellos pudiesen analizarse a sí mismos. Uno de los más exitosos fue, o más bien es, LISP. Se trata de un lenguaje funcional con buen soporte para el manejo de listas. Y da la "casualidad" de que los programas LISP son, precisamente, listas. Otro ejemplo de este tipo es Prolog, el lenguaje de programación lógica. Prolog también manejaba listas con soltura, y en las versiones más modernas, ampliaban bastante la lista de estructura de datos soportadas.

Los lenguajes antes mencionados no implementan el llamado paradigma procedimental. Puede que el pionero en añadir reflexión a un lenguaje de este tipo haya sido Smalltalk: no lo puedo afirmar con seguridad porque nunca me atrajo la idea de programar con una mano atada a la espalda. En cualquier caso, el hecho es que, cuando entra Java en escena, ya trae consigo un API de *reflexión* respetable. ¿Ha descubierto ya el factor común? Efectivamente: todos los mencionados son lenguajes que comenzaron su existencia como lenguajes interpretados. Es cierto que todos ellos terminaron con compiladores nativos, pero la moraleja es la siguiente: hasta hace muy poco, era consenso general que añadir información disponible en tiempo de ejecución sobre un programa, era un desperdicio… a no ser que el programa tuviese que ser interpretado.

¿Cómo demostrar que mi afirmación no es calumniosa? Ahí tiene el caso de C++. Los autores querían que fuese el lenguaje más eficiente del mundo. Consideraron, por ejemplo, que añadir cuatro bytes en cada instancia de objeto para almacenar un puntero a una tabla de métodos virtuales, era un derroche inaceptable. Tenga en cuenta que estamos hablando de los ordenadores del principio y mitad de los 80s. Como resultado, no se permitió que existiese un ancestro común a todas las clases (como sí lo hay en Delphi, muy similar a C++, pero diseñado en los 90s). A principios de los 90s, C++ comenzó a incluir lo que entonces se llamaba ***Runtime Type Information*** (RTTI), pero sólo como opción, y en cualquier caso, con muchas limitaciones en la información disponible. Delphi marcó una nueva etapa: no sólo ofrecía mucha más RTTI de serie que C++, sino que ésta era uno de los pilares del API de persistencia. Y Delphi nunca fue un lenguaje interpretado.

A no ser que haya empezado a leer el libro en esta página, ya imaginará lo que voy a decir a continuación: que el API de reflexión de C# y de todos los lenguajes de .NET, es mucho más grande, más grueso y además funciona mejor que el de Java. Bueno, ¿para qué negar la realidad?

La plataforma .NET pone a disposición del interesado, en tiempo de ejecución, toda la información que quepa imaginar, superando ampliamente la oferta y el alcance de lo que ofrece la máquina virtual de Java. Es más: esta diferencia es parte de los principios de diseño. Cuando presenté los tipos genéricos, mostré cómo se manifestaba la diferente filosofía de diseño de ambas plataformas: mientras que Java implementa los genéricos mediante trucos del compilador, en .NET los tipos genéricos son ciudadanos de primera y participan plenamente en la implementación del API de reflexión.

En la plataforma .NET, la reflexión es indispensable para los sistemas de persistencia, de programación remota, de seguridad e incluso para la implementación de la recolección de basura. La reflexión en .NET no se limita a averiguar información sobre un ensamblado, módulo o aplicación ya existente, sino que ofrece también varios esquemas para la generación de código ejecutable. El de más bajo nivel, y por lo tanto más potente, se conoce con el nombre del espacio de nombres donde se alojan la mayoría de sus clases: *Reflection.Emit*.

Por último, el API de reflexión nos permite acceder a una técnica novedosa disponible en todos los lenguajes .NET: el uso de *atributos*.

Información sobre tipos

Cuando utilizamos cualquiera de las técnicas de reflexión, tropezamos con la clase *System.Type*, cuyas instancias representan descriptores de tipos. Se trata de una clase abstracta: las clases derivadas concretas distinguen, por ejemplo, entre descriptores de tipos ya existentes en un ensamblado o descriptores de tipos creados sobre la marcha mediante generación de código. En concreto, cuando pedimos el descriptor de un tipo "normal", ya compilado, obtenemos en realidad una instancia de la clase *System.RuntimeType*. Ahora bien, esta es una clase interna, por lo que debemos manejar la instancia que recibimos mediante los miembros heredados de la clase base; esto es, a través de la clase *System.Type*.

Hay unas cuantas formas de obtener un descriptor de tipo. Si quiere el descriptor de un tipo concreto, del que no tiene una instancia en la mano, debe usar el operador **typeof** de C#. Imagine que se encuentra en el método de inicio de una aplicación de consola:

```
class Program
{
    static void Main(string args[])
    {
        System.Type t = typeof(Program);
        Console.WriteLine(t.FullName);
    }
}
```

La propiedad *FullName* de la clase *Type* devuelve el nombre completo de la clase asociada al descriptor, que incluye el espacio de nombres al que pertenece la clase. Si la clase *Program* del ejemplo reside dentro del espacio *ConsoleApplication1*, la aplicación escribirá en la consola:

```
ConsoleApplication1.Program
```

Si tenemos una instancia y queremos obtener el descriptor correspondiente a la clase de la instancia, debemos usar el método *GetType*, que se declara en *System.Object*. ¿Qué tal si le administramos a *Type* una dosis de su propia medicina?

```
class Program
{
    static void Main(string args[])
    {
        System.Type t = typeof(Program);
        Console.WriteLine(t.FullName);
        t = t.GetType();
        Console.WriteLine(t.FullName);
    }
}
```

EJERCICIO PROPUESTO

¿Qué debe escribir el último ejemplo en la consola? No necesita ejecutar el ejemplo para responder…

Estas dos técnicas son sólo las más directas. Una vez que tenga una instancia de *System.Type* en la mano, podrá navegar por las relaciones de herencia e implementación de tipos de interfaz que afectan a la clase representada por el descriptor. La clase base se obtiene mediante una propiedad:

```
public Type BaseType { get; }
```

La lista de interfaces implementadas o heredadas se obtiene mediante una función:

```
public Type[] GetInterfaces();
```

En este último caso, la función devuelve un vector de descriptores de tipos. Si el tipo no implementa ni hereda interfaces, *GetInterfaces* devuelve un vector de tamaño cero, en lugar de un puntero nulo. Podemos mezclar estas llamadas para averiguar la genealogía de un tipo de datos:

```
static void Main(string[] args)
{
    Type t = typeof(int);
    while (t != null)
    {
        Type[] intfs = t.GetInterfaces();
        Console.Write(t.FullName);
        if (intfs.Length > 0)
        {
            Console.Write(" implementa");
            foreach (Type intf in intfs)
            {
                Console.Write(" ");
                Console.Write(intf.Name);
            }
        }
        Console.WriteLine();
        t = t.BaseType;
    }
}
```

Cuando la indagación parte del tipo **int**, como en el fragmento anterior, se obtiene lo siguiente:

```
System.Int32 implementa IComparable IFormattable IConvertible
        IComparable`1 IEquatable`1
System.ValueType
System.Object
```

Observe que para escribir el nombre de los tipos de interfaz he usado la propiedad *Name* en vez de *FullName*. La clase **int** implementa dos tipos de interfaz genéricos, y el nombre completo de estos dos tipos es muy largo, pues contiene el nombre completo del ensamblado donde se define el propio tipo entero. Si siente curiosidad, cambie *Name* por *FullName* y compare los resultados.

Reflexión y ensamblados

La plataforma .NET es muy puntillosa, y con toda la razón, sobre la identidad de los tipos de datos. Dado el nivel extraordinario de dinamismo que permite, es muy importante que verifique, por ejemplo, que la clase *SqlConnection* que ha cargado en una aplicación es la *SqlConnection* "de verdad", *made in Microsoft*, por así decirlo, y no una clase pirata que delega todos los métodos en la verdadera *SqlConnection* a la vez que nos mantiene informados de la actividad de una máquina zombi. En principio, no hay nada que me impida declarar una clase llamada *System.Data.SqlCliente.SqlConnection* en un ensamblado mío. Lo que .NET no me permitirá es dar el cambiazo a la DLL que contiene la clase verdadera.

Por esta razón, las "coordenadas" completas de un tipo de datos deben indicar también en cuál ensamblado ha sido definido. En consecuencia, nuestro paseo por el API de reflexión debería haber comenzado por los ensamblados y módulos. La clase que representa un ensamblado cargado en el espacio de memoria de un proceso se llama *Assembly*, y se declarada en *System.Reflection*. Hay unas cuantas maneras de obtener una instancia de esta clase. Por ejemplo, si ya tenemos un descriptor de tipos a nuestra disposición, podemos obtener el descriptor del ensamblado donde se declara el tipo por medio de la propiedad *Assembly*:

```
Console.WriteLine(typeof(int).Assembly.FullName);
```

La instrucción anterior mostraría la siguiente cadena en la consola:

```
mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089
```

Esto se conoce como el *display name*, o nombre para mostrar, del ensamblado. Si en vez de utilizar el tipo predefinido **int** hubiésemos partido de la clase *Program* de nuestra aplicación de consola, obtendríamos algo parecido a lo siguiente:

```
ConsoleApplication1, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null
```

En ambos casos, se muestran los cuatro elementos que participan en la identidad de un ensamblado:

1. El nombre simbólico del ensamblado.
2. La versión. Recuerde que .NET permite la presencia y ejecución simultánea de distintas versiones de un mismo ensamblado, según las necesidades de otros componentes.
3. La cultura… que casi siempre se traduce en el idioma o dialecto.
4. Por último, la clave pública del fabricante, si el ensamblado ha sido firmado.

NOTA

El ensamblado *mscorlib* tiene características especiales. Es el ensamblado donde se definen los tipos básicos de .NET, y por lo general todos los restantes ensamblados hacen uso de él. Teóricamente, el compilador de C# permite compilar nuevos ensamblados que no hagan referencia automáticamente a *mscorlib*... pero hasta el momento, no he tenido necesidad de hacer tal cosa.

También se puede acceder a determinados ensamblados a través de métodos estáticos de la propia clase *Assembly*:

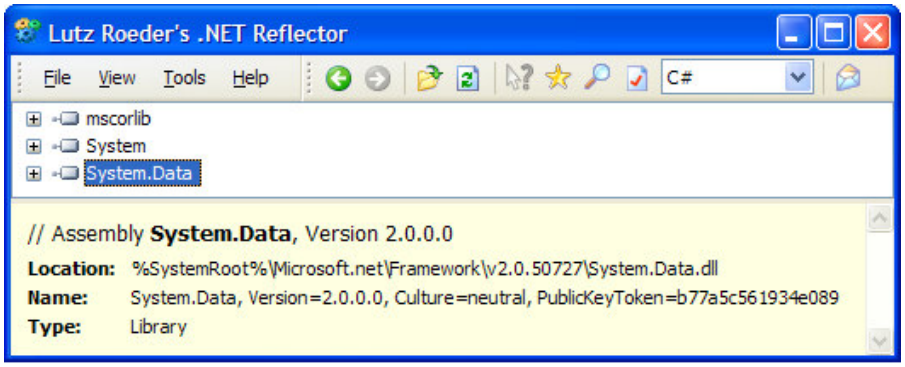
```
public static Assembly GetExecutingAssembly();
public static Assembly GetCallingAssembly();
public static Assembly GetEntryAssembly();
```

El primer método devuelve el ensamblado donde se define el método desde donde se ejecuta la llamada. El segundo nos indica el ensamblado donde se define el método que nos ha llamado. El último método de la lista nos indica en qué ensamblado se definió el método *Main* que se ha utilizado como punto de entrada de la ejecución de la aplicación en curso.

Observe que todas estas técnicas permiten obtener descriptores de ensamblados que ya se han cargado en el proceso. Muchas aplicaciones, sin embargo, necesitan cargar ensamblados en tiempo de ejecución. Este sería el caso, por ejemplo, de una aplicación extensible mediante *plug-ins*. La forma preferida para cargar un ensamblado en memoria es usar el método estático *Load*, de la clase *Assembly*, pasándole como parámetro el nombre completo del ensamblado:

```
System.Reflection.Assembly asm = System.Reflection.Assembly.Load(
    "System.Data, Version=2.0.0.0, Culture=neutral, " +
    "PublicKeyToken=b77a5c561934e089");
```

No se deje amedrentar por la aparente complejidad de la cadena: esta identificación es un invariante, que no cambia de máquina en máquina. Si usted quiere cargar un ensamblado que ya conoce, sólo tiene que averiguar cuál es su nombre completo. Basta con abrir el ensamblado con Reflector y seleccionar el nodo raíz, con cualquier lenguaje excepto IL en el combo de selección del lenguaje:



¿Y si el ensamblado no existe en el momento en que compilamos la aplicación? Esta es la situación típica cuando una aplicación carga *plug-ins* de terceros. En tal caso, hay también alternativas:

1. Que el usuario que suministra el *plug-in* nos suministre por su cuenta el nombre completo del mismo. No es una opción muy realista.

2. Que el usuario nos suministre la ubicación física del fichero. Entonces podemos usar el método estático *LoadFrom*, también de la clase *Assembly*, para cargar el ensamblado:

```
System.Reflection.Assembly asm = System.Reflection.Assembly.LoadFrom(
    @"C:\Windows\Microsoft.net\Framework\v2.0.50727\System.Data.dll");
```

Le advierto que *LoadFrom* tiene sus inconvenientes, porque no incluimos información completa sobre la identidad del ensamblado.

Suponga que finalmente hemos cargado un ensamblado que nos dicen que contiene clases para extender nuestra aplicación. Tenemos ya una variable de tipo *Assembly*; digamos que se llama *asm*. ¿Qué podemos hacer con el descriptor del ensamblado? La principal operación sería localizar un tipo por su nombre, o simplemente, podemos pedir los descriptores de todos los tipos declarados dentro del ensamblado:

```
foreach (Type t in asm.GetTypes())
    if (t.IsPublic)
        Console.WriteLine(t.FullName);
```

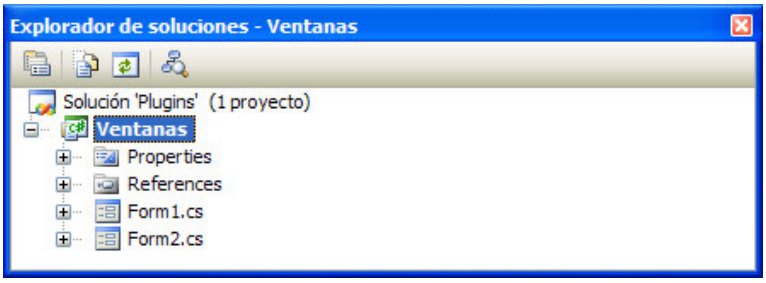
Para no hacer demasiado largo el listado de tipos, el fragmento anterior declara los tipos internos mediante la propiedad *IsPublic* del descriptor de tipo. Esto quiere decir que *GetTypes* devuelve todos los tipos del ensamblado, ¡incluyendo los tipos presuntamente internos!

Si le escandaliza este comportamiento, medítelo un poco: recuerde que la encapsulación tiene como objetivo la reducción de dependencias entre módulos de software. Su objetivo no es hacer de guardián de la propiedad intelectual. Si realmente necesitamos proteger nuestro código de ojos irrespetuosos, lo que debemos hacer es *ofuscar* el ensamblado.

Manejo dinámico de instancias

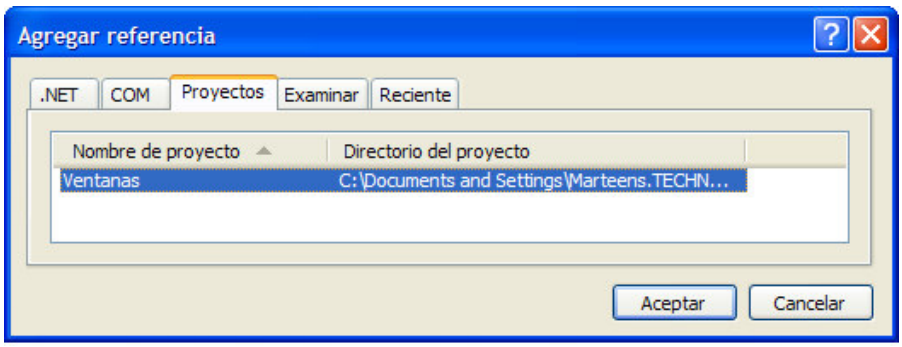
Una vez que tenemos un descriptor de tipo en la mano, el API de reflexión nos permite crear instancias del tipo, y acceder con total libertad a los miembros del tipo, incluso si no teníamos conocimiento del tipo durante la compilación. De hecho, es este último escenario el que nos interesa más, pues corresponde a la manera en que tendría que trabajar una aplicación extensible mediante *plug-ins*.

Si quiere jugar un poco con la técnica, cree una solución en Visual Studio y añádale primero un proyecto de tipo *Biblioteca de clases*. Luego, añada dos formularios de Windows Forms a la biblioteca. Para distinguirlos entre sí, puede cambiar la propiedad *BackColor* en los dos formularios. Guarde entonces el proyecto y llámelo *Ventanas*:



Simularemos que esta biblioteca es una extensión de la verdadera aplicación, que vamos a crear a continuación. Esta vez añadiremos un proyecto *Aplicación para Windows* a la solución. Márquela como proyecto de inicio, para que al pulsar el botón de ejecución, sea el ejecutable generado por este proyecto el que se ejecute. Echaremos mano de un truco: nos interesa, para simplificar el código fuente, que al ejecutarse el nuevo proyecto, la DLL del otro proyecto esté disponible en el

mismo directorio que el ejecutable. Hay una forma muy sencilla de lograrlo: hacer que el nuevo proyecto haga referencia al proyecto anterior. Para establecer esta dependencia, seleccione el nodo *Referencias* del proyecto ejecutable y ejecute el comando *Agregar referencia* del menú local. En el diálogo que aparece, active la tercera página y seleccione el proyecto de biblioteca:



Recuerde que sólo necesitamos este paso para evitar mover manualmente la DLL al directorio donde nos interesa que esté. Por este motivo, fingiremos que el ensamblado *Ventanas* no se carga automáticamente dentro del proceso ejecutable.

Intercepte el evento *Load* del formulario principal del proyecto de aplicación:

```
private void Form1_Load(object sender, EventArgs e)
{
    string filename = Path.Combine(
        Application.StartupPath, "ventanas.dll");
    if (File.Exists(filename))
    {
        Assembly asm = Assembly.LoadFrom(filename);
        foreach (Type t in asm.GetTypes())
            if (t.IsPublic && t.IsSubclassOf(typeof(Form)))
                CrearVentana(t);
    }
}
```

Primero, preparamos una cadena con la ruta completa hasta la carpeta donde suponemos que estará el fichero *Ventanas.dll*. El método estático *Combine*, de la clase *Path*, concatena la ruta con el nombre del fichero, teniendo cuidado de que haya exactamente una barra de separación entre la ruta y el nombre de fichero. Una vez que comprobamos que existe el fichero con las extensiones, cargamos el ensamblado en nuestro proceso con la ayuda del método *LoadFrom*.

NOTA

En honor a la verdad, hay que recordar que el ensamblado ya ha sido cargado junto con el ejecutable. Podríamos obtener el descriptor del ensamblado a través del descriptor del tipo *Form2* del ensamblado *Ventanas*, o incluso a través de *Form1*... siempre que tuviésemos cuidado al cualificar correctamente la clase. Si tiene tiempo, compruebe que en ambos casos se nos entrega el mismo descriptor de ensamblado.

Esta vez, cuando recorremos todos los tipos públicos del ensamblado con las extensiones, separamos aquellos tipos que descienden, directa o indirectamente, del tipo *Form*. El método que hemos usado, *IsSubclassOf*, sólo funciona con la herencia de clases. Cuando hay tipos de interfaz por medio, prefiero usar el método *IsAssignableFrom*, aunque hay que tener cuidado con este último pues invierte el papel de los objetos.

Si el tipo es público y es un formulario, llamamos al método *CrearVentana*, para que cree una instancia de esta clase y la muestre en pantalla:

```
private void CrearVentana(Type t)
{
    Form f = (Form)t.InvokeMember(null,
        BindingFlags.CreateInstance | BindingFlags.Instance |
        BindingFlags.Public, null, null, new object[0]);
    f.Show();
}
```

Sabemos que los formularios creados por Visual Studio tienen un constructor público sin parámetros. Vamos a aprovechar este conocimiento para evitarnos la búsqueda de un constructor adecuado dentro del descriptor del tipo. *CrearVentana* delega casi todo el trabajo al siguiente método de la clase *System.Type*:

```
public object InvokeMember(
    string name, BindingFlags invokeAttr, Binder binder,
    object target, object[] args);
```

InvokeMember es un verdadero campeón. La anterior es solamente una de sus tres variantes. Gracias a él, podemos poner en funcionamiento cualquiera de los miembros del tipo de datos cuyo descriptor tenemos a mano. Da igual si se trata de leer un campo, ejecutar una lectura sobre una propiedad, modificar una propiedad, ejecutar un método de instancia o estático, o incluso ejecutar un constructor: *InvokeMember* es lo que necesitamos.

Esta versatilidad se paga con un prototipo complicado. La versión que hemos usado pide primeramente el nombre del miembro del tipo al que vamos a acceder. Como se trata de un constructor, podemos pasar un puntero nulo, porque el método va a ignorarlo. Es en el segundo parámetro donde decimos que vamos a ejecutar el constructor. *BindingFlags* es un enumerativo cuyos valores pueden combinarse entre sí. *CreateInstance* pide un constructor, *Public*, que éste sea público, e *Instance*… bueno, hay que poner *Instance* por alguna razón que escapa a mi comprensión. El parámetro de tipo *Binder* sirve para suministrar un componente auxiliar por si hay que elegir entre alternativas sobrecargadas del método, y si es necesario, para convertir algún que otro parámetro. En nuestro caso, no lo necesitamos. Pasamos entonces la instancia del objeto, que al tratarse de un constructor, no existe, y finalmente, la lista de parámetros, para la que suministramos un vector de longitud cero.

InvokeMember se ocupa de todos los detalles sucios, crea el formulario y nos lo devuelve con una etiqueta donde se lee *"Esto es un objeto, y no una pipa"*. Nosotros, que sabemos que se trata de un formulario, aplicamos una conversión de tipo para quedarnos con un puntero decente sobre la criatura. Una vez que tenemos un formulario en la mano, podemos hacer con él todas las cosas que se le hacen a un formulario. En este caso, mostrarlo en pantalla ejecutando su método *Show*.

En principio, podríamos haber simplificado bastante la implementación de *CrearVentana*, e incluso eliminarla, sustituyendo la llamada a *InvokeMember* por una llamada al método *CreateInstance* de la clase *Assembly*:

```
foreach (Type t in asm.GetTypes())
    if (t.IsPublic && t.IsSubclassOf(typeof(Form)))
        ((Form)asm.CreateInstance(t.FullName)).Show();
```

La desventaja de *CreateInstance* es que repite la búsqueda del tipo dado su nombre, y por supuesto, que es un método menos flexible que *InvokeMember*.

Atributos

El ejemplo que acabamos de ver es una demostración de fuerza bruta: localizamos todos los formularios dentro de un ensamblado y los creamos todos de sopetón, porque nos da la gana. En la vida real, necesitaríamos que cada tipo encontrado nos diese más información. Por ejemplo, lo lógico es que no creásemos la ventana de manera inmediata, sino que creásemos un comando de menú para que el usuario crease la ventana cuando le apeteciera. ¿Qué texto debería mostrar el comando de menú? Es muy importante que comprenda que necesitamos ese texto *antes* de que hayamos tenido tiempo y oportunidad para crear la primera instancia de la clase.

La respuesta a este problema, y a otros cuántos, está en una técnica novedosa popularizada por .NET. Este fragmento de código muestra una típica declaración de propiedad en Delphi:

```
// Esto es Delphi:
property Edad: Integer read FEdad write SetEdad default 18;
```

La parte en la que debe reparar es la cláusula **default**. Como sugiere su nombre, sirve para indicar cuál es el valor por omisión de la propiedad. Sólo lo indica, sin embargo: la cláusula **default** no inicializa la propiedad. Su presencia no modifica en absoluto el código de inicialización de la clase. De hecho, si no hacemos algo, el campo *FEdad* se inicializa automáticamente a cero. El valor indicado por la cláusula se copia en una zona especial del descriptor de la clase en tiempo de ejecución. Si hemos utilizado una cláusula como la anterior, somos responsables de inicializar correctamente el campo durante la construcción de las instancias de la clase.

¿Qué sentido tiene entonces? Es el subsistema de persistencia de Delphi el que aprovecha esta información, en el momento en que tiene que guardar el estado de una instancia de la clase. Suponga que está trabajando con el diseñador de formularios de Delphi. Ha traído una instancia de la clase que contiene la propiedad *Edad*, y ha efectuado un par de cambios en otras propiedades. Cuando guardamos el estado de la instancia, Delphi comprueba que el valor de *Edad* sigue siendo el valor especificado en la cláusula **default** asociada a la propiedad… e infiere que no hace falta guardar el valor de la propiedad.

Resumamos, caminando de espaldas sobre nuestros pasos:

1. El subsistema de persistencia de Delphi intenta ahorrar espacio (y en algunos casos, tiempo de ejecución) guardando solamente el valor de las propiedades que han sido modificadas expresamente por el programador.
2. Para ello, necesita saber cuál es el valor por omisión de cada propiedad. Por desgracia, es muy complicado averiguarlo analizando el código fuente del constructor de la clase. Además, el valor se necesita *en tiempo de ejecución*.
3. La solución consiste en indicar explícitamente el valor por omisión de las propiedades, aunque esto introduzca redundancia en la declaración de la clase.
4. Para que podamos indicar el valor por omisión de una propiedad, los autores de Delphi previeron una extensión sintáctica especial para la declaración de propiedades.

La lógica de los tres primeros puntos es irreprochable. El cuarto punto es un disparate. No podemos modificar la sintaxis de un lenguaje de programación cada dos por tres. Ahora son los valores por omisión, pero mañana quién sabe.

Cuando Anders Hejlsberg diseñó C#, ya traía la lección aprendida. En vez de complicar la sintaxis del lenguaje con cláusulas *ad hoc*, recurrió a una técnica más potente: el uso de *atributos*. Veamos cómo se declara el valor por omisión de una propiedad en C#:

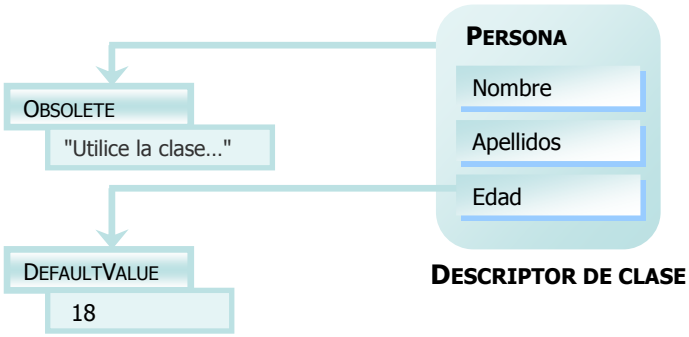
```
[DefaultValue(18)]
public int Edad
{
    get { … }
    set { … }
}
```

Un atributo es información que se asocia a una declaración, de manera que esté disponible para la aplicación en tiempo de ejecución. La declaración puede ser de propiedad, de campo, de método, de parámetro e incluso de módulo o de ensamblado. Por ejemplo, en la siguiente declaración he asociado un atributo a la clase y otro a una de sus propiedades:

```
[Obsolete("Utilice la clase Cliente")]
public class Persona
{
    [DefaultValue(18)]
    public int Edad { get { … } set { … } }

    ⋮
}
```

El atributo *Obsolete* se refiere en realidad a una clase llamada *ObsoleteAttribute*, que desciende de la clase predefinida *Attribute*: se trata de un convenio para abreviar el nombre las clases de atributos.

Esta clase *ObsoleteAttribute* ofrece un constructor público que recibe una cadena como parámetro. Por lo tanto, al asociar el atributo a la clase *Persona*, el compilador crea una instancia de *ObsoleteAttribute* y la engancha al descriptor interno de la clase *Persona*:



- Aunque hayamos creado mil y una instancias de la clase *Persona*, sólo hay una instancia del atributo en memoria, como máximo.
- Aunque todavía no hayamos creado una sola instancia de *Persona*, de todos modos podemos recuperar sus atributos a través del descriptor de la clase.

Algo parecido ocurre con la propiedad *Edad* y el atributo *DefaultValue*: se crea internamente una instancia de la clase *DefaultValueAttribute* con el correspondiente constructor, y se asocia al descriptor de la propiedad, que existe a su vez dentro del descriptor de la clase *Persona*.

¿Quién hace uso de esta información aportada por los atributos? Depende. En el caso de *DefaultValue*, como he dicho, el subsistema de persistencia aprovecha esta información para generar código de inicialización de componentes. En el caso de *Obsolete*, es el propio compilador quien comprueba si las clases mencionadas en un proyecto contienen este atributo, para quejarse en caso afirmativo. En el caso del atributo *WebServiceAttribute*, que se puede aplicar a una clase, es la parte del motor de ASP.NET que implementa servicios Web la que busca este atributo y, cuando lo encuentra en una clase, prepara la infraestructura de la clase para permitir llamadas remotas a sus métodos:

```
[WebService(
     Description="Mi servicio",
     Namespace="http://intsight.com/")]
public class MiServicioWeb
{
    ⋮
}
```

En el caso de *WebService*, he utilizado una sintaxis diferente para crear el atributo: en vez de pasar parámetros posicionales, he utilizado dos parámetros con nombres. No se deje engañar por la terminología: *Description* y *Namespace* no son verdaderos parámetros de un constructor de la clase *WebServiceAttribute*, sino nombres de propiedades de esta clase. Para crear el objeto correspondiente al atributo, primero se crea la instancia llamando a un constructor sin parámetros, y luego se le asignan valores a las dos propiedades mencionadas.

NOTA Recuerde: un atributo es simplemente una forma de añadir anotaciones al código fuente, con la peculiaridad de que podemos recuperar luego estas anotaciones en tiempo de ejecución. Un atributo, en sí, no tiene otros efectos en la aplicación o biblioteca, aunque el propio compilador y el entorno de desarrollo utilizan atributos para sus fines particulares.

Diseño de atributos

Volvamos a nuestro ejemplo inicial: una aplicación extensible mediante *plug-ins*. Ya sabemos cómo abrir un ensamblado, recorrer sus tipos e identificar los formularios. Pero necesitamos más control: necesitamos marcar aquellos formularios que realmente queremos que se utilicen para extender la aplicación nodriza. Además, si queremos crear comandos en un menú para crear ventanas, tendremos que indicar el texto que deberían mostrar esos comandos, al menos. En una situación ideal,

deberíamos indicar también el icono asociado al comando, y el orden relativo respecto a otras extensiones de ventanas. Pero de momento, nos conformaremos con el texto. Marcaremos los formularios disponibles como extensión mediante un atributo, y dicho atributo nos indicará el texto del comando de menú que crearemos.

Vamos a añadir una nueva clase en el proyecto DLL, en *Ventanas*:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=false)]
public class MenuItemAttribute : Attribute
{
    private string menuText;

    public MenuItemAttribute()
    {
        this.menuText = String.Empty;
    }

    public MenuItemAttribute(string menuText)
    {
        this.menuText = menuText ?? String.Empty;
    }

    public string MenuText
    {
        get { return menuText; }
        set { menuText = value ?? String.Empty; }
    }
}
```

EJERCICIO PROPUESTO

Uno de los constructores y el método de acceso para escritura de una propiedad de esta clase utilizan el operador de fusión (p.88). ¿Por qué, o para qué?

Nuestra clase se llama *MenuItemAttribute*, cumpliendo con el requisito de las clases de atributos, que deben contener *Attribute* como sufijo, y desciende de la clase *System.Attribute*. La propia clase ha sido marcada con un atributo predefinido:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=false)]
```

Este atributo indica que *MenuItem* sólo podrá usarse con clases (¡esto excluye las estructuras!), y que para cada clase, el atributo sólo podrá mencionarse una sola vez, como máximo. Hay dos constructores públicos en la clase: uno que no admite parámetros, y otro que recibe un parámetro de tipo cadena. La cadena se utiliza para inicializar la propiedad pública *MenuText*. Esto significa que estos dos usos de la clase son equivalentes:

```
[MenuItem("Texto para el comando")]
[MenuItem(MenuText = "Texto para el comando")]
```

En el primer caso, inicializamos la propiedad indirectamente, usando el constructor que recibe una cadena de caracteres. En el segundo caso, la asignación se ejecuta directamente sobre la propiedad, como si tuviésemos un parámetro con nombre.

A continuación, utilizaremos el nuevo atributo para marcar todos los formularios del proyecto *Ventanas* en este plan:

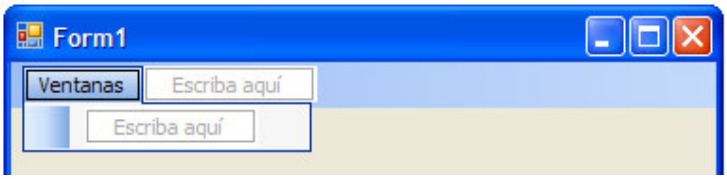
```
[MenuItem("Primera ventana")]
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
}
```

En estos casos, hay que tener en cuenta si deseamos que un atributo sea heredado junto con una clase. No olvide que en Visual Studio podemos hacer que una clase de formulario tenga como clase

base otro formulario definido por nosotros mismos. En nuestro caso, no nos interesa, pues cada formulario disponible debe tener su propio comando de menú, con texto propio. Tenemos dos opciones: asignar **false** en la propiedad *Inherited* de *AttributeUsage*, o excluir atributos heredados en el código que lee los atributos posteriormente. Nosotros utilizaremos la segunda vía

Reflexión sobre atributos

Estamos ahora en la aplicación nodriza. Lo primero que haremos es añadir un esqueleto de menú a su formulario principal. Sólo tenemos que arrastrar un componente *MenuStrip* al formulario. Luego, crearemos un comando directamente sobre la barra, como muestra la siguiente imagen:



Muy importante: llamaremos *miVentanas* al objeto de comando que corresponde al menú desplegable *Ventanas*.

Tenemos que modificar la respuesta al método *Load* del formulario. Antes, recorríamos los tipos publicados por el ensamblado de extensión, y creábamos inmediatamente cada ventana encontrada. Ahora, crearemos un comando de menú para cada tipo que descienda de *Form* y que venga decorado por un atributo *MenuItem*:

```
private void Form1_Load(object sender, EventArgs e)
{
    string filename = Path.Combine(
        Application.StartupPath, "ventanas.dll");
    if (File.Exists(filename))
    {
        Assembly asm = Assembly.LoadFrom(filename);
        foreach (Type t in asm.GetTypes())
            if (t.IsPublic && t.IsSubclassOf(typeof(Form)))
                CrearMenu(t);
    }
}
```

CrearMenu es el método que se encargará de crear el comando de menú a partir de la información que extraiga del atributo *MenuItem*. Para obtener los atributos asociados a un tipo de datos determinado, utilizaremos el siguiente método de la clase *Type*:

```
public object[] GetCustomAttributes(Type attributeType, bool inherit);
```

En esta versión del método, pasamos el descriptor de tipo del atributo en que estamos interesados. El segundo parámetro indica si aceptaremos atributos definidos en un ancestro: en nuestro caso, no. Observe que *GetCustomAttributes* devuelve un vector de objetos, al menos nominalmente. Esto significa que tendremos que realizar una conversión de tipos sobre el valor devuelto por este método.

Esta es, finalmente, nuestra implementación de *CrearMenu*:

```
private void CrearMenu(Type t)
{
    Ventanas.MenuItemAttribute[] attrs =
        (Ventanas.MenuItemAttribute[])t.GetCustomAttributes(
            typeof(Ventanas.MenuItemAttribute), false);
    if (attrs.Length > 0)
    {
        string menuText = attrs[0].MenuText;
        ToolStripItem item = miVentanas.DropDownItems.Add(menuText);
        item.Tag = t;
```

```
            item.Click += new EventHandler(item_Click);
        }
    }
```

Como ve, la mayor complicación es la conversión de tipos. Se trata de una conversión entre tipos de vectores, y sólo es posible porque *Object* y *MenuItemAttribute* son, ambos, tipos de referencia. Si tuviésemos que repetir esta llamada muchas veces, puede que nos interesase encapsular la conversión dentro de un método genérico como el siguiente:

```
public static T[] GetAttrs<T>(Type t, bool inherit) where T : Attribute
{
    return (T[])t.GetCustomAttributes(typeof(T), inherit);
}
```

La limitación de *GetAttrs* es que, al menos en .NET 2.0, no puede beneficiarse de la inferencia de tipos, y nos obliga a repetir el tipo del atributo dos veces, en vez de tres.

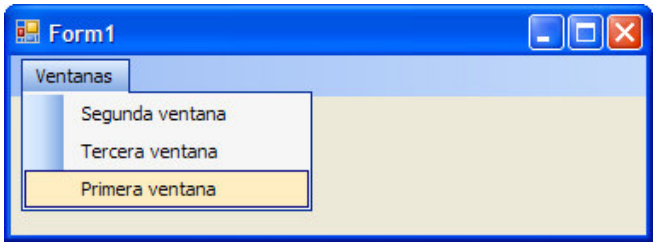
En *CrearMenu* he aprovechado un truco: en ningún caso, *GetCustomAttributes* devuelve un puntero nulo. Si no se encuentran atributos del tipo indicado, el método devuelve un vector de longitud cero. Esto nos evita comprobaciones innecesarias sobre el valor de retorno. El resto de *CrearMenu* es muy sencillo: si encontramos el atributo en el descriptor del tipo, creamos un comando de menú, añadiéndolo dentro de la lista de comandos del menú desplegable *Ventanas*. Aprovechamos la propiedad *Tag* del objeto de comando para recordar a qué tipo de formulario está asociado el comando, y finalmente enlazamos el evento *Click* del comando con un método de respuesta a eventos que hemos llamado *item_Click*:

```
private void item_Click(object sender, EventArgs e)
{
    ToolStripItem item = sender as ToolStripItem;
    if (item != null)
    {
        Type t = item.Tag as Type;
        if (t != null)
            CrearVentana(t);
    }
}
```

El método *CrearVentana* al que hace referencia *item_Click* es nuestro viejo amigo, cuya implementación repito aquí para mayor comodidad:

```
private void CrearVentana(Type t)
{
    Form f = (Form)t.InvokeMember(null,
        BindingFlags.CreateInstance | BindingFlags.Instance |
        BindingFlags.Public, null, null, new object[0]);
    f.Show();
}
```

La siguiente imagen corresponde a la aplicación en funcionamiento:



Note que el orden en que se descubren los tipos de formularios y se crean los comandos, es arbitrario. Piense en algún mecanismo que nos ofrezca algo más de control sobre el orden de los comandos.

INDICE ALFABETICO

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W