

SERIE DESARROLLO



Aprenda C# 3.0 desde 0.0

Parte 3, lo nuevo

GUILLERMO "GUILLE" SOM



**SOLID
QUALITY**
MENTORS

ADVERTENCIA LEGAL

Todos los derechos de esta obra están reservados a Guillermo Som y Solid Quality Press

El editor prohíbe cualquier tipo de fijación, reproducción, transformación o distribución de esta obra, ya sea mediante venta, alquiler o cualquier otra forma de cesión de uso o comunicación pública de la misma, total o parcialmente, por cualquier sistema o en cualquier soporte, ya sea por fotocopia, medio mecánico o electrónico, incluido el tratamiento informático de la misma, en cualquier lugar del mundo.

La vulneración de cualesquiera de estos derechos podrá ser considerada como una actividad penal tipificada en los artículos 270 y siguientes del Código Penal.

La protección de esta obra se extiende al mundo entero, de acuerdo con las leyes y convenios internacionales.

© Guillermo Som, 2008

© Solid Quality Press, 2008

Aprenda C# 3.0 desde 0.0

Parte 3, lo nuevo

Serie Desarrollo

Autor: Guillermo “Guille” Som

Editado por Solid Quality Press

Apartado de correos 202

03340 Albufera, Alicante, España

<http://www.solidq.com>

Precio: 15,00€

ISBN: 978-84-936417-4-0

Prólogo

¿Guille?, “El” Guille, ¿Y quién es ese Guille?

Así fue como tomé conciencia que este personaje existía (vamos, existe ☺).

En realidad, a partir de un error de concepto. Yo tratando de pelear contra todos los molinos de viento que os podáis imaginar (y no en La Mancha, que era en Buenos Aires), tratando que alguien me dejara hacer algo donde publicar información técnica en la lengua de Don Cervantes.

En aquellas, yo era uno de tantos que, chapurreando algo de inglés, rondaba por los foros (si si, antes de los ñús, así de viejo soy), porque era hasta donde sabía, el único lugar donde podía participar.

Dictaba cursos por aquellas épocas y cuando avanzando un poco en el tiempo, un participante de un curso me comentó de “el sitio del Guille” no pude menos que ver de qué se trataba.

Así fue como vi ese sitio que, como todos en aquellos tiempos, tenía poco de estética, pero mucho de contenido.

Y, lo que más me llamó la atención, fue que se entendía a la primera leída.

Pasó el tiempo y avanzando me encontré participando en los “ñús” de Microsoft... y ahí estaba el señor este, contestando y contestando...

Si hasta varias veces mandamos a buscar información a gente que venía de Office, tratando de programar en “bebeá” (que en Latino América no se dice “úve” a la “v”)

Y quiso el destino que me encargaran tratar de difundir cosas nuevas .Net, (en esa época ERA nueva), y comencé a ser una voz conocida en Latino América, Y tuve la gran oportunidad de conocer a mucha, pero mucha gente por América del Sur (y central), y siempre alguna referencia se hacía al “sitio del Guille”.

Y finalmente, en una reunión de oradores internacionales de Ineta tuve la oportunidad, el placer y el honor de conocerlo.

Y así fue como en los últimos siete años, nos hemos cruzado, reunido al menos un par de veces al año. (Que menos, si trabajamos en la misma empresa, damos conferencias aquí y acullá, y conocemos mucha gente de uno y otro lado del charco).

He visto, como la gente queda, a boca abierta, por hora y media viendo a este señor como si fuese una especie de estrella de rock.

He respondido, cientos de veces, a mensajes pidiéndome si puedo conseguir que “El Guille nos venga a dar una conferencia”.

SI hasta he tenido pedidos de Subsidiarias de Microsoft para ello.

Hasta que no lo vivís, es imposible que comprendáis de quien estoy hablando.

Pero claro; se lo conoce como “el experto de Visual Basic”

¿Y qué hago yo entonces “prologando” este libro de C#?

Bueno, hay varias razones:

- 1) Se lo debo a mi amigo desde hace rato
- 2) Comparto con él tanto el placer del conocer, como el placer del enseñar
- 3) Tengo muy claro que la teoría de la relatividad (perdón, tío Einstein), explicada por el Guille sería materia común en escuela de niños
- 4) Más allá que le he dicho hasta el cansancio que eso de escribir “;”, y “{“ y “}” me resulta incómodo con mi teclado... y que muchas veces me pierdo para saber dónde termina el if, o la función, o el “néimespáse” entre esa maraña de llaves
- 5) Sé que él entiende de la misma forma que yo esta ciencia de la programación “nética”... no importa el lenguaje con que programes, sino aquello que estás programando.

¿Quieres aprender en serio a usar adecuadamente .Net?

¿Quieres que tu código sea claro sin vericuetos crípticos inentendibles de modo que dentro de mucho tiempo te resulte tan claro como ahora?

¿Quieres sacar el máximo jugo a tus posibilidades de desarrollo con la plataforma .Net?

¿Quieres ENTENDER y no sólo “saber cómo se hace”?

Este es el medio. No lo dudes.

Y prometo continuar mi insistencia para que Guille no abandone jamás esta actitud de compartir, de enseñar y de ayudar. Tanto, que hasta se ocupa de quienes necesitan ayuda sin tener nada que ver con la tecnología. Y es por todas estas cosas que lo admiro.

“Larga vida y prosperidad” A mi amigo Guille... y a ti, querido lector

Daniel “Dani” Seara

Solid Quality Mentor

Humano de la tierra

Aprenda C# 3.0 desde 0.0

Parte 3, lo nuevo

Introducción

Este libro forma parte de una trilogía sobre la versión 3.0 de C# llamada globalmente **Aprenda C# 3.0 desde 0.0**, esta tercera parte está dedicada exclusivamente a las nuevas características de C# 3.0 y surge a raíz de (y basado en) mi primer libro electrónico: **Novedades de Visual Basic 9.0**.

Las otras dos partes de esta trilogía las publicaré a lo largo del año próximo, y el contenido que tengo previsto para cada uno de ellos es:

- **Aprenda C# 3.0 desde 0.0 - Parte 1: Lo elemental**
 - Todo lo elemental o básico del lenguaje: tipos de datos, *generic*, variables, bucles, expresiones, control de flujo, lo esencial del IDE de Visual Studio 2008 (y la versión Express), *arrays*, colecciones, colecciones *generic*, clases, interfaces, estructuras, constructores, enumeraciones, delegados, eventos, etc., etc., etc.
- **Aprenda C# 3.0 desde 0.0 - Parte 2: Lo esencial**
 - Todo lo relacionado con la programación orientada a objetos (POO), sobrecarga de operadores, conversiones personalizadas, los delegados y eventos a fondo, más sobre el IDE de Visual Studio y profundizar en los temas que se vieron en la primera parte, para ampliar conocimientos, etc., etc., etc.

¿Por qué escribo un libro de C#?

Para quién no me conozca, decir que siempre he programado con lenguajes de la familia BASIC, desde los que se incluían en la ROM de los ordenadores hasta las versiones de Visual Basic para .NET, pasando por Quick Basic, y aunque he trabajado con otros lenguajes, al final siempre he acabado usando Visual Basic como lenguaje para mis proyectos, pero como me gusta programar por programar, desde que apareció C# lo estudiado y usado a la misma par que Visual Basic .NET, principalmente para incluir en los artículos que escribo para mi sitio Web (www.elguille.info) todo el código en los dos lenguajes principales de .NET.

La idea de escribir un libro de C# siempre me ha rondado por la cabeza, y la verdad es que son muchos los que alguna vez me han preguntado ¿Para cuándo un libro de C# como tu libro [Manual Imprescindible de Visual Basic 2005](#)?

Pero como siempre anda uno a la última pregunta, es decir, que faltan horas en el día, pues se van dejando las cosas, surgen otras nuevas y así un día, una semana un mes... ¡y hasta un año tras otro!

A raíz de la publicación de mi primer libro electrónico sobre las Novedades de Visual Basic 9.0, e incluso antes, (cuando lo anuncié el 25 de febrero que lo iba a publicar), hubo gente que me volvía a preguntar ¿para cuándo el libro de C#? y... ¡Por fin! ¡Aquí está! ¡Ya es una realidad!

Como he comentado antes, lo que he hecho (o esa era inicialmente mi intención), es adaptar mi libro de Novedades de Visual Basic 9.0 a las novedades de C# 3.0, aunque al final, este libro contiene cosas que en muchos casos son iguales en ambos libros, al menos en cuanto a las cosas que había que explicar, pero en otros muchos casos, he añadido cosas totalmente diferentes, particularmente porque C# y Visual Basic son distintos, y no solo en la sintaxis, sino en el número de novedades que cada lenguaje ha introducido, ya que muchas de las novedades de Visual Basic 9.0 no lo son en la versión 3.0 de C# y en otros casos, porque Visual Basic incluye ciertas novedades que C# no y en todo lo referente a LINQ, el número de cláusulas de Visual Basic con respecto a las de C# es muchísimo más amplio, para paliar esa desventaja, he optado por explicar las funciones o métodos extensores que podemos usar desde C# (y también desde Visual Basic, pero que al tener instrucciones propias no era necesario) para realizar las mismas acciones, y ya que estaba en la tarea de explicar algunas de ellas, me he tomado el tiempo para intentar explicarlas todas ellas, o al menos las que yo considero más usuales.

Otra diferencia importante es todo lo relacionado con *LINQ to XML*, que en Visual Basic está integrado totalmente, es decir, podemos usar código de XML directamente en el editor de Visual Basic, mientras que en C# siempre debemos usar las clases que .NET Framework ofrece para trabajar con esta tecnología de LINQ destinada a datos XML.

Y como resulta que mientras escribía este libro ha aparecido la versión final de *ADO.NET Entity Framework* (incluida en el Service Pack 1 de .NET y Visual Studio 2008), y en ese nuevo marco de trabajo de acceso a datos se incluye *LINQ to Entities*, he extendido el último capítulo del libro para explicar cómo podemos usarlo desde nuestro código escrito con Visual C# 2008, con idea de que el lector tenga una idea de cómo crear las entidades y cómo usar los diferentes objetos de la base de datos, incluyendo un pequeño truco para paliar la falta del diseñador de entidades para la creación automatizada del código necesario para usar los procedimientos almacenados.

Al igual que el libro de Novedades de Visual Basic 9.0, este libro y los otros dos de la serie Aprende C# 3.0 desde 0.0 están publicados por Solid Quality™ Press.

¿A quién va dirigido este libro?

En este libro explico, con todo lujo de detalles, todas las nuevas características de C# 3.0 (o Visual C# 2008), con abundantes ejemplos y, al menos esa ha sido mi intención, de una forma clara y fácil de entender. Aún así, este libro no es un manual para no iniciados, por tanto, es recomendable que quién quiera sacarle el máximo partido a este libro tenga nociones de cómo trabajar con versiones anteriores de C#, concretamente Visual C# 2005 (o C# 2.0). En particular el conocimiento de las peculiaridades de C#, así como el conocimiento de programación orientada a objetos, y las características ofrecidas por la versión 2.0 de .NET Framework, ayudará a un mejor entendimiento del contenido de este libro.

Material de apoyo

Todo el código mostrado en el libro, así como otras pruebas y proyectos usados en los diferentes ejemplos, está disponible en la dirección Web:

<http://www.solidq.com/ib/DownloadEbookSamples.aspx?id=2>.

Desde ahí podrá bajar un archivo en formato ZIP con todos los proyectos usados para los diferentes ejemplos del libro. Existe una carpeta con los proyectos de cada capítulo, de forma que resulte fácil seguir los ejemplos mostrados, y en el código se indica qué código corresponde con cada listado.

La mayoría, al estar escritos con Visual C# 2008, los podrá usar tanto con la versión comercial de Visual Studio 2008 como con la versión gratuita (Visual C# 2008 Express Edition); solo para las soluciones que utilizan proyectos en C# y Visual Basic, necesitará utilizar Visual Studio 2008.

Debido a que *LINQ to Entities* se incluye con el Service Pack 1 de Visual Studio 2008, necesitará tener instalado ese Service Pack para poder usar los proyectos correspondientes a esa tecnología.

Agradecimientos

Quiero agradecer a todos los mentores de **Solid Quality Mentors** el apoyo que he tenido desde siempre, pero quiero agradecer de forma particular a mi amigo **Daniel Seara** por todo el apoyo y motivación que me ha dado para que terminara en un tiempo razonable la escritura de este libro, porque -todo hay que decirlo- cuando me pongo a escribir textos tan largos, se me pasan los días como minutos y cuando quiero darme cuenta ha pasado casi un año, así que... hay que agradecer a Dani la insistencia para que no me dejara llevar por la “languidez” y terminar a tiempo el libro. También agradecerle que haya escrito el prólogo del libro, ¡Todo un honor, hermano!

También quiero agradecer a **José Quinto** por las revisiones que ha hecho de los capítulos del libro, y por las inquietudes que ha tenido, para que aclarara mejor los conceptos, capítulos que finalmente **Javier Loria** y Daniel han vuelto a revisar y darle forma para que todos los puedan leer de una forma cómoda y sin tantos modismos, que si bien en España son de uso diario, los amigos de Latinoamérica podían no entender o podían haberlo interpretado de otra forma.

A mi amigo **Paco Marín**, editor, entre otras, de la revista **dotNetManía**, quiero también agradecerle la confianza que siempre ha puesto en mi, y que en muchas ocasiones *casi* me ha obligado a que pensara en C# (y dejara de hacerlo en Visual Basic, que es mi principal lenguaje de programación), particularmente en los primeros años de vida de la revista, en la que mantuve la sección **dnm.Inicio**, en la que estuve explicando los pormenores de la programación con .NET usando código de C#. También quiero agradecerle que me esperara casi todos los meses para que le entregue casi a última hora los artículos, aunque cada vez es más duro conmigo y ahora, como ha decidido (sabiamente) no trabajar los fines de semana, me lo ha puesto un poco más complicado, pero como ahora la sección que mantengo mensualmente en la revista (**Isla.VB**) es sobre el lenguaje que he “mamado” desde *chiquitillo*, normalmente suelo llegar a tiempo... o casi.

Tampoco quiero olvidarme del “gefe”, **Fernando Guerrero**, que ha sabido motivarme y alentarme para que mi primer libro publicado con la empresa que él dirige, fuese una realidad y estuviera listo en un tiempo récord, además de alentarme para que siga escribiendo libros, como esta trilogía sobre

C# 3.0 y otros que aún tengo pendientes de convertir en una realidad, y que espero que a lo largo del año 2009 se conviertan en una realidad.

Y cómo no, en particular quiero agradecer a mi “parienta” **Mari Carmen**, por toda la paciencia que siempre tiene conmigo, particularmente cuando desconecto con el mundo real para centrarme en “mis cosas”, y en algunas ocasiones, esas desconexiones suelen más ser largas de lo deseado, porque a pesar de lo que muchos piensan, escribir, incluso de lo que uno sabe, no es tarea fácil, sobre todo si se quiere hacer bien y no contar las cosas que uno “cree” que pueden ser. Tampoco quiero olvidarme de mis dos “monstruitos” **David** y **Guille**, ya que los hijos también nos motivan a hacer las cosas que hacemos, y casi siempre, incluso inconscientemente, las hacemos por ellos.

A todos, incluso a los que no he mencionado, darles las gracias por todo el apoyo que me han dado. Y por supuesto a los lectores, ya que al fin y al cabo, es para quién espero que todo este esfuerzo les sea de gran ayuda y puedan conocer todo lo que C# 3.0 trae de nuevo.

Nos vemos

Guillermo

Nerja, 18 de septiembre de 2008

Contenido

Prólogo.....	3
¿Guille?, “El” Guille, ¿Y quién es ese Guille?	3
Aprenda C# 3.0 desde 0.0 Parte 3, lo nuevo	5
Introducción	5
¿Por qué escribo un libro de C#?	5
¿A quién va dirigido este libro?	6
Material de apoyo.....	7
Agradecimientos	7
Contenido.....	9
Parte 1 El entorno de desarrollo	15
Capítulo 1 El entorno de Visual C# 2008	17
Introducción	17
Nombres y versiones	17
¿Visual C# 2008 o C# 3.0?.....	17
Las diferentes versiones de .NET	18
¿Cuáles son las diferencias entre estas versiones de .NET?	19
Las diferentes versiones de C#	20
El IDE de Visual C# 2008	22
Tipos de proyectos y versión de .NET Framework	22
Cambiar la versión de .NET una vez creado el proyecto	23
¿Se pueden usar aplicaciones creadas con versiones de .NET superiores a la instalada en el equipo del usuario?	25
Nuevo diseño del cuadro de diálogo de agregar elementos a un proyecto.....	25
El IDE de Visual Studio 2008 actualizado para el UAC de Windows Vista	27
Configuración del entorno para Visual C#.....	27
Parte 2 Características generales del lenguaje	31
Capítulo 2 Nuevas características de C# 3.0 (I).....	33
Introducción	33
Inferencia de tipos en variables locales	33
Inferencia de tipos en los arrays (matrices).....	34
¿Cuándo y dónde podemos usar la inferencia de tipos?	34
Inicialización de objetos y colecciones	36

Inicialización de objetos con clases que definan solo constructores con parámetros.....	37
Inicialización de objetos e inferencia de tipos.....	38
Inicializaciones de arrays y colecciones.....	38
Métodos parciales.....	40
Condiciones para los métodos parciales.....	42
Los métodos parciales para usar con delegados	42
El IDE de Visual C# 2008 y los métodos parciales	43
Capítulo 3 Nuevas características de C# 3.0 (II)	45
Introducción	45
Propiedades autoimplementadas	45
Capítulo 4 Nuevas características de C# 3.0 (III)	49
Introducción	49
Tipos anónimos	49
Definir un tipo anónimo	50
¿Cómo de anónimos son los tipos anónimos?	51
Comprobación de igualdad en los tipos anónimos	53
¿Son iguales dos instancias del mismo tipo anónimo con los mismos datos?	55
Los tipos anónimos solo los podemos usar a nivel local	56
Tipos anónimos que contienen otros tipos anónimos	56
Recomendaciones	58
Expresiones lambda.....	59
Entendiendo a los delegados	60
Definir una expresión lambda.....	61
Las expresiones lambda como parámetro de un método	62
Delegados genéricos	64
Ámbito en las expresiones lambda	65
¿Cómo clasificar una colección de tipos anónimos?	67
Consideraciones para las expresiones lambda.....	72
Capítulo 5 Características generales del lenguaje (IV)	75
Introducción	75
Métodos extensores	75
Los métodos extensores e IntelliSense.....	77
Desmitificando los métodos extensores	77
El ámbito de los métodos extensores	80

Precedencia en el ámbito de los métodos extensores	80
Definir las clases con métodos extensores en su propio espacio de nombres	83
Si la cosa puede empeorar, seguro que empeora	83
Conflictos con métodos existentes	84
Sobrecargas en los métodos extensores	84
Sobrecargas y array de parámetros en los métodos extensores	85
¿Qué tipos de datos podemos extender?	87
Reflexionando sobre los métodos extensores	90
Parte 3 C# y LINQ	91
Capítulo 6 C# y LINQ (I)	93
C# y LINQ	93
Un ejemplo de LINQ para ir abriendo boca	93
Los diferentes sabores de LINQ	97
LINQ to Objects	98
Elementos básicos de una consulta LINQ	98
Ejecución aplazada y ejecución inmediata	99
La cláusula select	100
Ordenar los elementos de la consulta	102
No es oro todo lo que reluce	102
Capítulo 7 C# y LINQ (II)	105
Instrucciones de C# para las consultas de LINQ	105
select	105
from y join	106
Funciones de agregado	108
Las funciones Average, Count, LongCount, Max, Min y Sum	110
Las funciones All y Any	112
Filtrar las consultas con where	113
Distinct	117
Indicar la ordenación de los elementos de una consulta	119
Invertir los datos con Reverse	120
Agrupaciones y combinaciones: group by y join into	121
group by (GroupBy)	121
join into (GroupJoin)	123
Divide y vencerás: Skip y Take	125

SkipWhile y TakeWhile.....	126
Vuelve un clásico de los tiempos de BASIC: let	128
Respuesta al ejercicio de la sección dedicada a where	130
Funciones de agregado personalizadas	131
Capítulo 8 C# y LINQ (III).....	135
Otros métodos extensores para consultas LINQ	135
Ordenar datos.....	135
Operaciones de conjuntos (Set)	136
Filtrar datos.....	137
Operaciones cuantificadoras.....	138
Operaciones de proyección.....	138
Particiones de datos.....	139
Operaciones de combinación.....	139
Agrupar datos	139
Operaciones de generación.....	140
Operaciones de igualdad	142
Operaciones de elementos	143
Convertir tipos de datos.....	148
Operaciones de concatenación.....	151
Operaciones de agregación.....	152
Capítulo 9 C# y LINQ (IV).....	157
Introducción	157
LINQ to XML	157
C# y LINQ to XML.....	157
Un AddIn para convertir código XML en código de C#	159
Clases más usuales de LINQ to XML.....	160
Una forma fácil de crear documentos XML en C#.....	162
Acceder al contenido XML en memoria	163
Acceder a un elemento de la consulta	165
Trabajar con espacios de nombres	166
Añadir nuevos elementos al código XML.....	168
Añadir nuevos elementos o atributos a un elemento	171
Modificar valores de los elementos	173
Validar los datos XML.....	174

LINQ to XML es LINQ	175
Capítulo 10 C# y LINQ (V)	177
Introducción	177
C# y LINQ to ADO.NET	177
LINQ to DataSet	178
Añadir un DataSet al proyecto.....	179
Crear un adaptador y llenar una tabla	179
Consultar los datos de una tabla	180
Consultas LINQ en DataSet no tipado.....	181
Añadir más elementos al DataSet	184
LINQ to SQL	186
Modelo de objetos de LINQ to SQL.....	186
Crear el código automáticamente	190
Crear las clases usando el diseñador relacional	190
LINQ to Entities	194
Crear un proyecto LINQ to Entities	194
Asociar una función con un procedimiento almacenado	199
Acceder a los objetos creados por el diseñador de Entity Data Model.....	202
Índice alfabético	205

(Esta página se ha dejado en blanco de forma intencionada)

Parte 1

El entorno de desarrollo

En la primera parte del libro veremos las novedades presentadas en el entorno de desarrollo de Visual Studio 2008, además de conocer qué versiones de .NET Framework podemos usar en nuestros proyectos.

También veremos algunos consejos para configurar el entorno de desarrollo con idea de mejorar la escritura de nuestro código con Visual C# 2008.

(Esta página se ha dejado en blanco de forma intencionada)

Capítulo 1

El entorno de Visual C# 2008

Introducción

En este primer capítulo del libro sobre las novedades de C# 3.0 (o Visual C# 2008, que es como se conoce comercialmente al entorno de desarrollo distribuido por Microsoft) veremos qué es lo que rodea al lenguaje (el entorno).

Empezaremos viendo qué versión o versiones de .NET podemos usar con el nuevo compilador de C#, y acabaremos conociendo algunos de los cambios realizados en el entorno de desarrollo (IDE) recomendado para sacarle el máximo provecho a esta nueva versión del lenguaje que nació con .NET Framework.

Nota

*Lo que veremos serán las cosas que han cambiado con respecto a la versión anterior, es decir, no veremos con detalle **todo** lo que concierne al entorno de desarrollo, solo las novedades aplicables a Visual Studio 2008 y en particular al editor de Visual C#.*

Nombres y versiones

¿Visual C# 2008 o C# 3.0?

Visual C# 2008 es el nombre comercial de la última versión de C# para crear aplicaciones bajo la tutela de .NET Framework.

El entorno de desarrollo de esta nueva versión se presenta (al igual que la versión anterior) en dos formas distintas: una de ellas, la comercial, se denomina **Visual Studio 2008**, con la que podemos crear aplicaciones escritas con cualquiera de los lenguajes de programación que incluye (entre los cuales, se encuentra Visual C#) y también nos permite utilizar otros “complementos” con los que podremos crear cualquier tipo de aplicación para .NET Framework; además de la versión comercial (de pago), podemos utilizar un entorno de desarrollo integrado que está especializado pura y exclusivamente en un lenguaje de programación o tecnología particular. Si ese lenguaje es C#, tendremos que elegir la versión que se conoce como **Visual C# 2008 Express Edition**. Las versiones Express de Visual Studio son totalmente gratuitas y operativas al 100%, es decir, no son versiones de prueba. Las características que ofrece (limitadas con respecto a la versión comercial, todo hay que decirlo)

están implementadas completamente, por tanto, lo que ofrece, lo ofrece al completo; más adelante, veremos algunas de las limitaciones de esta versión gratuita, pero antes sigamos hablando de los diferentes nombres de C#.

El nombre o la numeración 2008 es solo una forma comercial de llamar a este lenguaje (o si lo prefiere, al entorno integrado que usaremos para crear nuestras aplicaciones), ya que internamente se mantiene la numeración iniciada con la primera versión del lenguaje cuando hizo su aparición en el año 2002. Por tanto, el nombre interno del lenguaje es C# 3.0 que es la versión del compilador que se distribuye con .NET Framework 3.5.

Como seguramente ya sabrá el lector, el compilador de C# se distribuye de forma totalmente gratuita con el *runtime* de .NET. Y es ese compilador el que utilizan los entornos de desarrollo de Visual Studio 2008. Al estar incluido en el paquete de distribución del motor de ejecución (*runtime*) de .NET, podremos usar esta nueva versión del compilador desde otros entornos de desarrollo, al menos si están configurados para usar esta nueva versión de .NET. También podemos usar esta nueva versión del compilador de C# desde la línea de comandos, y por tanto, podremos compilar cualquier archivo sin necesidad de usar ningún entorno de desarrollo.

Por supuesto, lo recomendable es usar siempre alguna herramienta que nos facilite y acelere la creación de nuevos proyectos, y como veremos en este mismo capítulo, la opción más válida es usar Visual Studio 2008 o la versión gratuita del entorno.

Nota

El compilador de C# siempre estará disponible si tenemos el runtime de .NET instalado, y como resulta que sin dicho runtime no podremos usar ninguna aplicación de .NET, es evidente que todos los usuarios que tengan el runtime de .NET Framework instalado tienen la posibilidad de crear aplicaciones de C#.

Las diferentes versiones de .NET

Cuando hizo su aparición Visual Studio 2005, la versión de .NET Framework que necesitaba ese entorno de desarrollo era (y es) la versión 2.0. Por otro lado, Visual Studio 2008 requiere la versión 3.5 de .NET.

Sabiendo esto, es posible que nos preguntemos si existe algún Visual Studio que utilizara las versiones de .NET que hayan aparecido entre la 2.0 y la 3.5. La respuesta es no. De hecho, entre estas dos versiones de .NET solo ha habido una: la versión 3.0. Esa versión de .NET Framework es la que se distribuye con Windows Vista y Windows Server 2008, aunque también se puede instalar en otros sistemas operativos como puede ser Windows XP con Service Pack 2 o Windows Server 2003 con Service Pack 1.

En la figura 1.1 podemos ver las diferentes versiones de .NET Framework y la correspondencia con cada versión de C# y Visual Studio, así como el año en que hizo su aparición.

¿Cuáles son las diferencias entre estas versiones de .NET?

Estas tres últimas versiones de .NET (que son las que podemos elegir para crear nuestros proyectos con Visual C# 2008, tal como veremos en un momento) comparten una misma versión del motor en tiempo de ejecución (*runtime*) de .NET, al que llamaremos CLR 2.0, que son las siglas en inglés de *Common Language Runtime*.

El CLR 2.0 es el corazón de .NET, y todas las versiones de .NET Framework que han aparecido desde finales de 2005 hasta la actualidad (tomando como actualidad el año 2008) comparten o necesitan de esa versión del *runtime*.

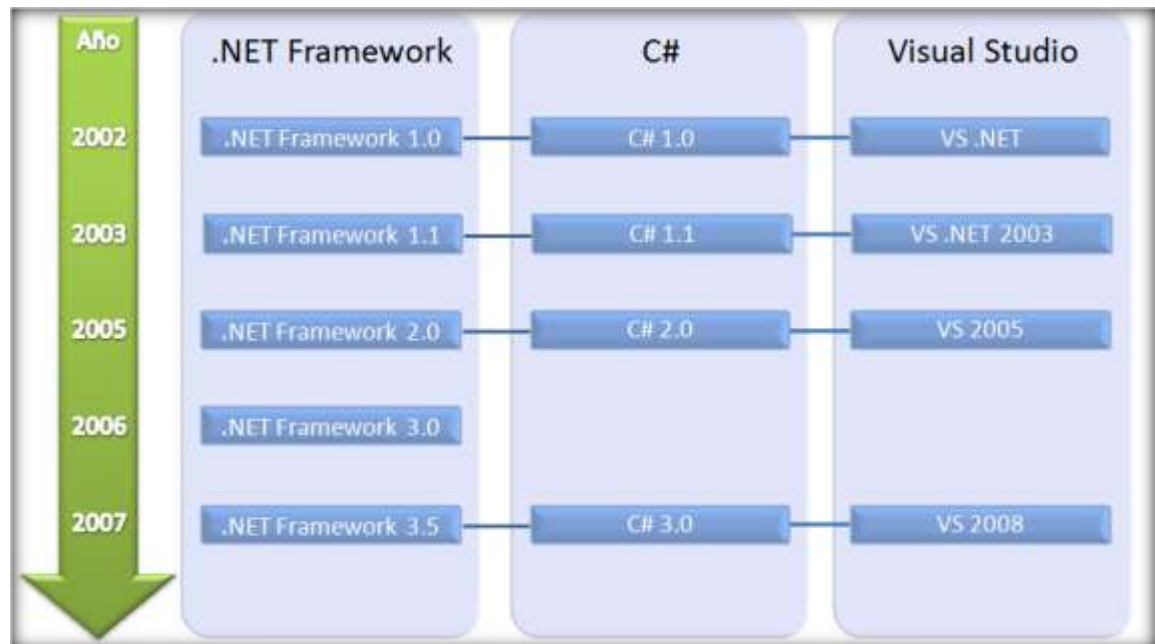


Figura 1.1. Las diferentes versiones de .NET Framework y su relación con Visual Studio

La diferencia entre las dos nuevas versiones de .NET (la 3.0 y la 3.5) con respecto a la 2.0, es que han añadido nuevos ensamblados que amplían la funcionalidad del corazón de .NET Framework. Esa nueva funcionalidad viene en forma de ensamblados (o bibliotecas), que se han incorporado al .NET Framework para permitir usar características como todo lo referente a *Windows Presentation Foundation* (WPF) y otros marcos de trabajo introducidos en la versión 3.0, o todo lo referente a LINQ que es prácticamente lo que trae de nuevo la versión 3.5.

Para tener una visión global del contenido de cada una de las versiones de .NET Framework que están relacionadas con la versión 2.0 del motor en tiempo de ejecución (CLR) y las diferentes tecnologías que incorporan, podemos ver el esquema de la figura 1.2.

¿Qué significa esto? Para nosotros, programadores o creadores de aplicaciones con Visual C# 2008, lo que significa es que la funcionalidad que tenemos al crear las aplicaciones programadas en C#, es la que nos ofrece la versión 2.0 del motor de ejecución de .NET, pero también podemos utilizar todo lo que nos ofrecen las bibliotecas añadidas en las nuevas versiones.

Las diferentes versiones de C#

No voy a hacer un repaso a la historia de C#, pero sí quiero aclarar un poco qué versiones son las que existen de este lenguaje para .NET Framework.

C# hizo su aparición con la primera versión de .NET Framework a principios del año 2002 junto con Visual Studio .NET. En esa primera versión de C#, el motor en tiempo de ejecución (CLR) es el conocido como .NET Framework 1.0. El nombre interno del compilador es C# 1.0 o también Visual C# 2002.

Un año después hizo su aparición Visual Studio .NET 2003 y con él llegó C# 1.1 o Visual C# 2003, la versión de .NET que incluía esa versión es la 1.1.

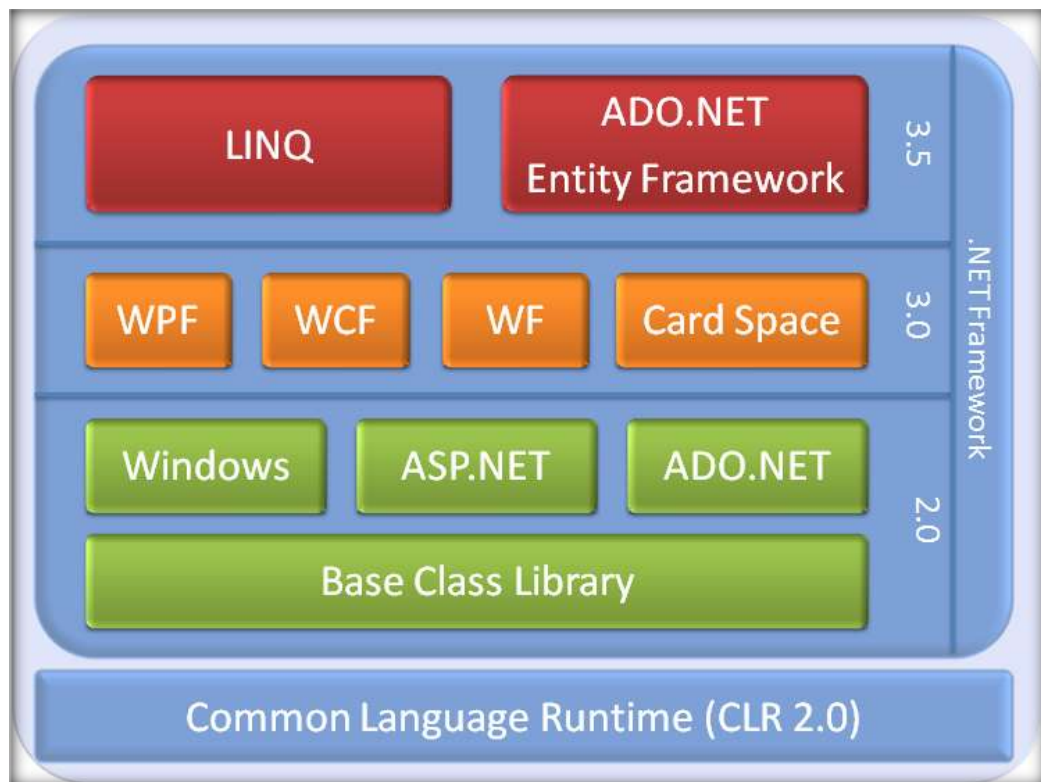


Figura 1.2. Las versiones de .NET Framework que están relacionadas con la versión 2.0 del motor en tiempo de ejecución (CLR)

A finales del año 2005 hubo una serie de cambios en la nomenclatura de Visual Studio, ya que dejó el “apéndice” .NET y empezó a llamarse simplemente Visual Studio 2005. La versión de .NET Framework también cambió de numeración, pasando a ser la versión 2.0. Y como estamos compro-

bando, la versión del compilador C# siempre va acorde al número de versión de .NET, por tanto, el nombre interno del compilador es C# 2.0, aunque comercialmente será Visual C# 2005.

En la versión 2.0 de .NET Framework hubo grandes cambios, tantos, que incluso dejó de ser compatible con las versiones anteriores; esto significa que con Visual Studio 2005 no se pueden crear aplicaciones que utilicen una versión anterior de .NET, por tanto, cualquier aplicación creada con el compilador de C# 2.0 solo se puede usar en máquinas que tengan instalado el *runtime* de la versión 2.0 de .NET Framework (en Visual Studio .NET 2003 se podía elegir la versión de .NET que queríamos usar, en esa ocasión cualquiera de las dos soportadas: la 1.0 o la 1.1).

En noviembre de 2006 apareció la versión 3.0 de .NET Framework, pero no hubo ningún Visual Studio que acompañara a esta nueva versión de .NET. En realidad, con esta versión empezaron los “líos”, ya que el nombre de .NET cambió de versión (de la 2.0 a la 3.0), pero la versión del motor en tiempo de ejecución (CLR) se mantuvo, es decir, seguía siendo la 2.0. Esa versión agregó nueva funcionalidad a .NET, pero no cambiaba la forma interna de trabajar del *runtime*, solo agregaba nueva funcionalidad en forma de ensamblados externos.

A finales del año 2007 apareció Visual Studio 2008 y otra nueva versión de .NET, la 3.5. Con esta versión se incluye el compilador de C# 3.0, que es el protagonista de este libro, y como podemos comprobar, con esta versión se cortó el paralelismo entre la numeración del compilador y la versión de .NET Framework, lo que si se mantiene es el nombre comercial que sigue ligado a la versión o numeración de Visual Studio. Al igual que ocurrió con la versión 3.0, esta nueva versión de .NET solo añade nueva funcionalidad al CLR; esa funcionalidad también viene en la forma de ensamblados externos, por tanto, el número de la versión del *runtime* de .NET (CLR) sigue siendo la 2.0.

El cambio en esta nueva versión del compilador (C# 3.0) es que, entre otras cosas, aprovecha los “añadidos” al .NET Framework, tanto de la versión 3.0 como de la versión 3.5, pero además se han agregado nuevas funcionalidades que son independientes de los ensamblados que acompañan a las diferentes (y nuevas) versiones de .NET Framework. Esos cambios los veremos en los próximos capítulos.

Para que quede claro, todo esto que estoy comentando sobre las versiones de .NET, de Visual Studio y de C#, es para indicar que, independientemente de la versión que elijamos para crear nuestros proyectos (en un momento lo veremos), las novedades incluidas en el compilador de C# 3.0 siempre estarán disponibles.

Para que quede más claro: si elegimos crear un proyecto para que use .NET Framework 2.0 (por tanto, no usará ninguno de los ensamblados añadidos en las versiones posteriores) podemos seguir usando las nuevas características añadidas al compilador de C#. Por supuesto, si algunas de esas novedades implican el uso de ensamblados, que solo se distribuyen con alguna versión posterior de .NET 2.0, es “físicamente” imposible aprovecharlas. En el repaso a las novedades del lenguaje indicaré cuáles de esas novedades podemos usar con cada una de las versiones de .NET.

Aunque en este primer capítulo veremos las novedades presentadas en el entorno de desarrollo, en los siguientes nos centraremos en las novedades propias del compilador y también en las novedades, que en cierta forma están relacionadas con los ensamblados que acompañan a la nueva versión de .NET Framework.

El IDE de Visual C# 2008

Tal como he comentado, la creación de programas en C# 3.0 la podemos realizar con cualquier editor y luego compilar desde la línea de comandos, pero lo habitual es que usemos algún entorno de desarrollo (IDE, *Integrated Development Environment*, entorno de desarrollo integrado). El compilador de C# 3.0 es el utilizado por las versiones de Visual Studio 2008, que es el IDE recomendado para crear aplicaciones de .NET; aunque existen otras ofertas de entornos de desarrollo, en este libro solo hablaremos de la oferta propuesta por Microsoft.

Como ya comenté, existen básicamente dos propuestas para el entorno de desarrollo que podemos usar con C#, uno es el entorno “completo” y comercial: Visual Studio 2008, que se ofrece en distintas versiones, según la cantidad de opciones que contemple. Visual Studio es un entorno multilenguaje, en el sentido de que desde el mismo entorno de desarrollo podemos crear aplicaciones para varios lenguajes de programación. Entre ellos está, como es de esperar, el lenguaje C#. Además, Visual Studio también ofrece un entorno de trabajo para las aplicaciones Web: Visual Web Developer (VWD). No es que sea algo diferente, sino que al trabajar con aplicaciones dirigidas al mundo Web es el entorno que se utiliza, ya que Visual Studio se acomoda al tipo de aplicación que estamos creando.

Las ediciones gratuitas de Visual Studio se conocen como Express Edition, y según el lenguaje para el que esté preparado tendrá un nombre diferente; en el caso de C#, la versión gratuita es Visual C# 2008 Express Edition y en el caso del IDE para la creación de aplicaciones Web es Visual Web Developer 2008 Express Edition. Esta última permite crear aplicaciones Web en cualquiera de los lenguajes soportados, entre los que está C#.

Nota

Todo lo comentado en esta sección (salvo cuando se indique expresamente) es aplicable tanto a la edición comercial (Visual Studio 2008) como a la versión gratuita del entorno integrado (Visual C# 2008 Express Edition).

Tipos de proyectos y versión de .NET Framework

Como ya he comentado, con C# 3.0 podemos crear proyectos para las tres versiones de .NET Framework basadas en la versión 2.0 del CLR.

Por tanto, es lógico pensar que al crear un nuevo proyecto de C# podamos elegir entre esas tres versiones de .NET, y precisamente ésta es una de las novedades que nos encontramos al crear un nuevo proyecto de C#.

Cuando indicamos que queremos crear un nuevo proyecto, el entorno de desarrollo nos muestra una ventana similar a la mostrada en la versión anterior, pero en esta ocasión también nos presenta una nueva lista desde la que podemos elegir para qué versión de .NET Framework queremos crear el

proyecto. En la figura 1.3 vemos esa lista desplegable y los tipos de proyectos que podemos crear desde la versión profesional de Visual Studio 2008.

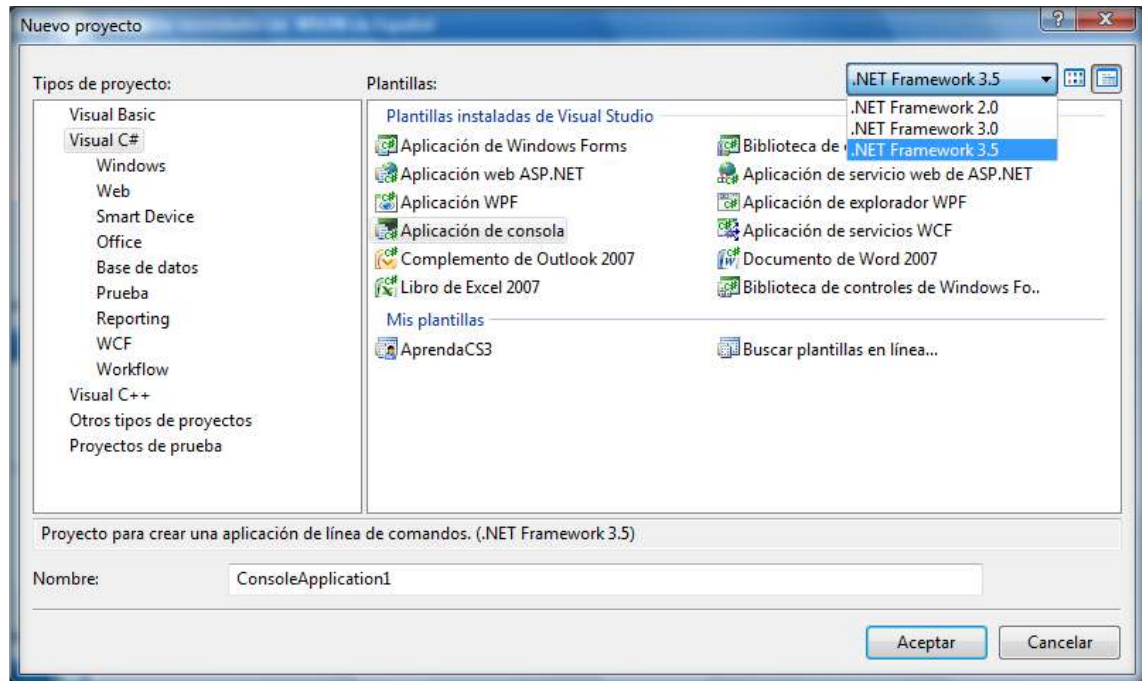


Figura 1.3. Los proyectos de Visual C# 2008 para .NET Framework 3.5

Debido a que cada versión de .NET permite crear diferentes tipos de proyectos, la lista de las plantillas que podemos usar con cada versión variará para ofrecer solo las que haya disponibles para trabajar con la versión de .NET que hayamos seleccionado de la lista.

Nota

En Visual C# 2008 Express Edition solo se muestran las plantillas para .NET Framework 3.5, pero como veremos en el siguiente punto, podremos cambiar la versión final de .NET que se utilizará para compilar el proyecto.

Cambiar la versión de .NET una vez creado el proyecto

Las plantillas de los tipos de proyectos que podemos crear están configuradas de forma que agregan las referencias a los ensamblados que se pueden usar en cada una de las combinaciones de la versión de .NET y el tipo de aplicación a crear. Debido a que esas referencias dependen de la versión de .NET, solo estarán disponibles cuando elijamos la plataforma adecuada. Este comentario, que puede parecer obvio, viene al caso para explicar que una vez seleccionada la versión de .NET que queremos usar en nuestra aplicación, podemos cambiar desde las propiedades del proyecto a otra versión diferente. Ese cambio lo haremos desde la ficha **Aplicación** (*Application*), particularmente desde la

lista desplegable que hay bajo **Marco de trabajo de destino** (*Target Framework*) en las propiedades del proyecto, y tal como vemos en la figura 1.4, podemos seleccionar la versión de .NET Framework que queremos usar en este proyecto en particular.

Nota

Como veremos a continuación, la selección de una versión de .NET Framework no obliga a que el equipo del usuario final de nuestra aplicación deba tener esa misma versión de .NET, pero si no la tiene no se podrán usar los nuevos ensamblados que se distribuyen con esa versión.

Nota

Ya sabemos que no todas las traducciones son perfectas, y en el caso de la ventana de propiedades de proyectos de Visual C# 2008 no es una excepción, si nos fijamos en la figura 1.4, veremos que la traducción original, en inglés, de "Target Framework" ha sido "Marco de trabajo de destino", sin embargo esa misma opción en el entorno de Visual Basic está traducida como "Versión de .NET Framework de destino" a pesar de que en la versión en inglés de Visual Studio tiene el mismo nombre tanto en Visual Basic como en Visual C#.

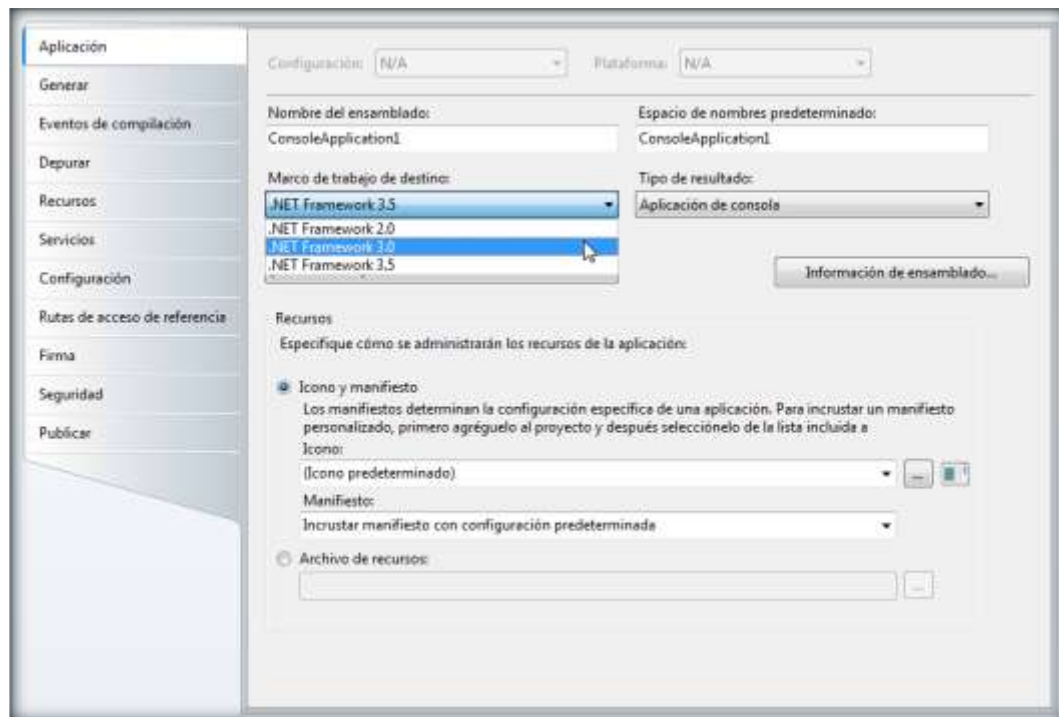


Figura 1.4. Cambiar la versión de .NET desde las propiedades del proyecto

¿Se pueden usar aplicaciones creadas con versiones de .NET superiores a la instalada en el equipo del usuario?

Esta puede ser una pregunta que algunos se harán, o simplemente ni se la plantearán dando un **no** por respuesta, incluso fundamentándolo en que si son diferentes versiones, también debe ser diferente el *runtime* usado. Pero la respuesta a esta pregunta es un **sí rotundo**. Por supuesto, un sí rotundo, pero con matices. Tal como he comentado al principio de este primer capítulo, la versión del *runtime* de .NET, que se utiliza en las tres variedades de .NET Framework para las que C# 3.0 puede crear aplicaciones, es en realidad la 2.0, ya que, lo que hace diferentes a las versiones de .NET, son los ensamblados que acompañan a dicho motor en tiempo de ejecución (CLR). Por tanto, la base de las diferentes versiones de .NET sigue siendo la misma: el CLR 2.0, los matices vienen de la mano de los ensamblados que acompañan a cada versión de .NET.

Aclarado esto, y para que no queden dudas, comentar que si creamos una aplicación en la que indicamos que la versión de .NET será la 3.5, pero nuestro código no utiliza ninguno de los ensamblados que se distribuyen con esa versión, el ejecutable resultante podrá utilizarse en equipos que tengan instalada una versión anterior de .NET Framework. Pero si esa aplicación usa algunas de las características de la versión 3.5, fallará, ya que el equipo no tiene los ensamblados adecuados.

Nuevo diseño del cuadro de diálogo de agregar elementos a un proyecto

Además del cambio en el cuadro de diálogo de creación de nuevos proyectos, el usado para agregar nuevos elementos a un proyecto existente también ha cambiado.

El cambio de este último cuadro de diálogo es que (tal como podemos ver en la figura 1.5) en la parte izquierda de la pantalla se muestra un árbol con diferentes opciones que nos facilita la elección o búsqueda del elemento que queremos añadir al proyecto.

Otra novedad de Visual C#/Studio 2008 en lo referente a los cuadros de diálogos de nuevos proyectos y elementos es que dicho cuadro de diálogo es *redimensionable*, por tanto, si se muestran muchos elementos en dicho cuadro de diálogo, podemos cambiar el tamaño para acomodarlo mejor a nuestras preferencias o simplemente para que se vean todos los elementos tal como ocurre en la figura 1.6.

Hay otras ventanas y cuadros de diálogo que han cambiado con respecto a la versión anterior de Visual Studio, pero, en la mayoría de los casos (o en todos), es solo para acomodar nuevas opciones, algunas de las cuales ya hemos visto. Y en otros casos, algunas de las opciones existentes están posicionadas en lugares diferentes de los mismos cuadros de diálogo, también para adaptarse al número de opciones que ahora debe poner a nuestra disposición.

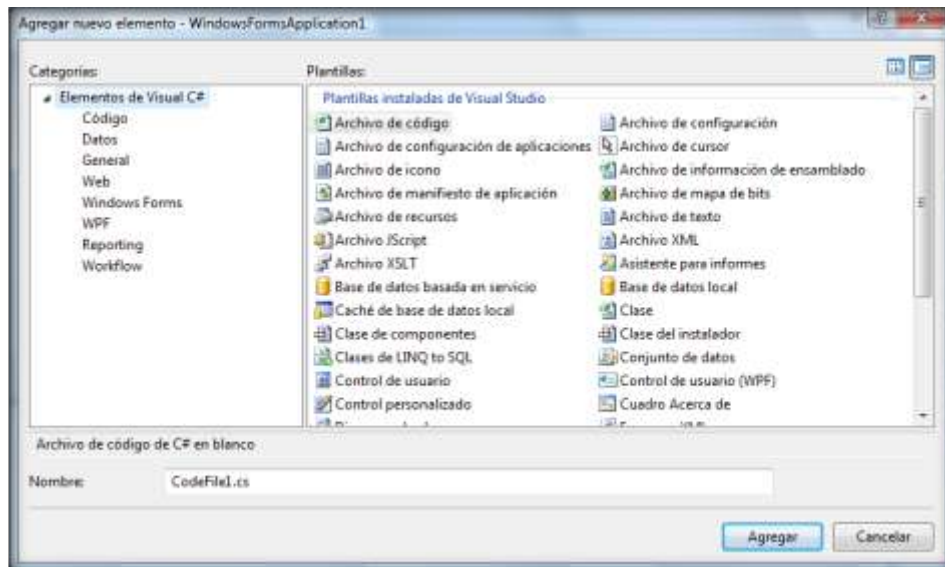


Figura 1.5. El cuadro de diálogo de añadir nuevos elementos al proyecto

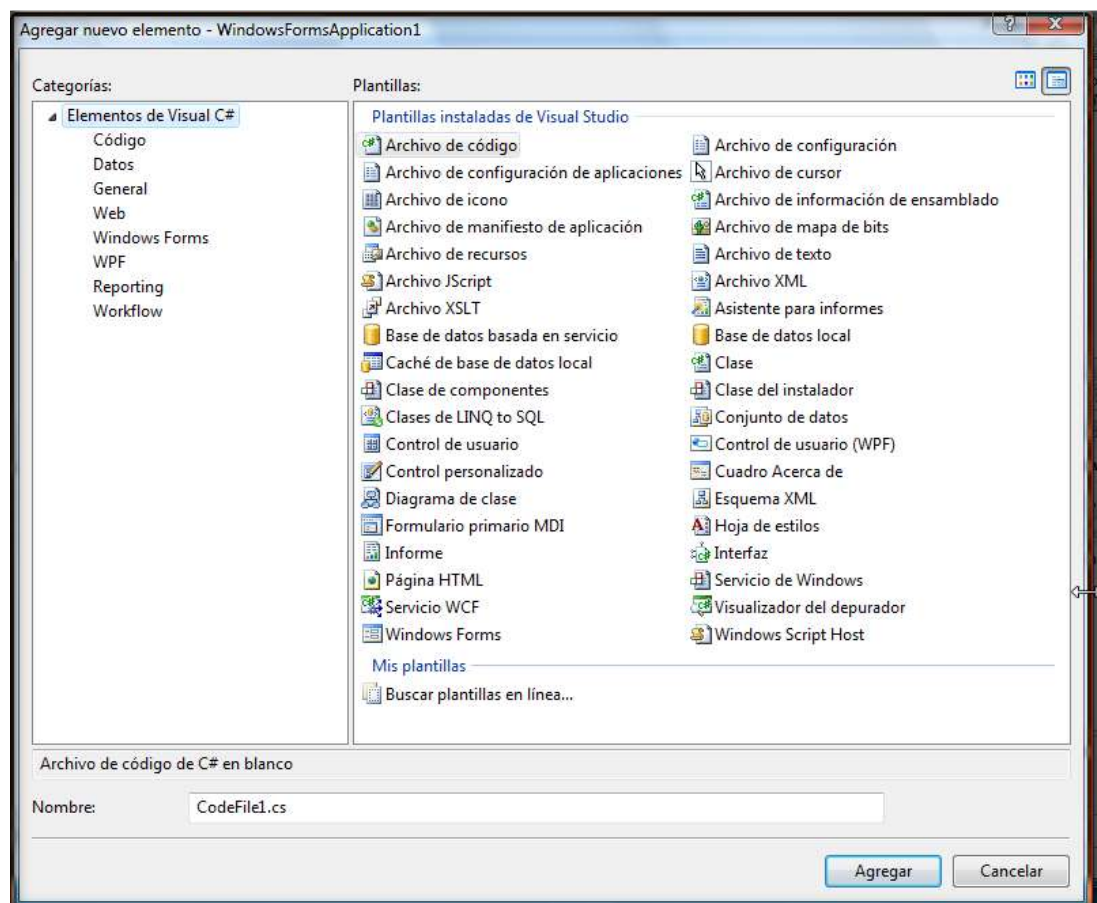


Figura 1.6. La ventana de agregar nuevos elementos con un tamaño adecuado para mostrar todas las opciones disponibles

El IDE de Visual Studio 2008 actualizado para el UAC de Windows Vista

Afortunadamente, Visual Studio 2008 ha aparecido después de Windows Vista, y digo “afortunadamente”, porque los que instalamos Visual Studio 2005 en Windows Vista tuvimos que esperar unos meses para que el IDE se adaptara al nuevo sistema operativo, e incluso después de la aparición del Service Pack 1 de Visual Studio 2005 y la actualización correspondiente para los usuarios de Windows Vista, el entorno de desarrollo funciona diferente según lo estemos usando como usuario normal o con los privilegios de administrador. En Visual Studio 2008 (y también en las versiones Express) esto ha cambiado a mejor, ya que ahora se tiene en cuenta esta característica de Windows Vista y ya no es necesario que ejecutemos el entorno de desarrollo como administrador para tener a nuestra disposición todas las características que más se veían afectadas, como es todo lo referente a la depuración. Aún hay cosas que necesitan privilegios de administrador, pero, en la mayoría de los casos, no será necesario hacer ningún cambio en la forma de iniciar el entorno de desarrollo para trabajar cómodamente y con todas las posibilidades de depuración. Por supuesto, si queremos tener a nuestra disposición todas las características de Visual Studio 2008, tendremos que ejecutar el IDE con los privilegios de administrador.

Ni qué decir tiene, que esto solo es aplicable a Windows Vista o Windows Server 2008 (el servidor basado en el “concepto” de Windows Vista) y siempre que tengamos activado el UAC (*User Account Control*). En otros sistemas operativos como Windows XP o Windows Server 2003 siempre se ejecutará según los privilegios que tenga el usuario desde el que iniciamos el entorno de desarrollo.

Configuración del entorno para Visual C#

Antes de dar por finalizado este capítulo sobre el entorno de desarrollo de Visual C# 2008, me gustaría comentar algo que, si bien no es nuevo, sí que considero importante que cualquier usuario de Visual Studio deba conocer. No es algo que afecte a nuestro trabajo, y posiblemente no nos ofrecerá ningún beneficio, pero yo siempre suelo hacer ciertos cambios en el entorno de desarrollo para que me resulte más cómodo trabajar, principalmente si utilizo otros lenguajes, como pueda ser Visual Basic o incluso el IDE para la creación de aplicaciones Web.

Cuando abrimos Visual Studio por primera vez (esto no es aplicable a la versión Express de Visual C#), nos pregunta qué tipo de entorno queremos utilizar, dándonos a escoger uno entre las posibilidades que tengamos según los lenguajes y otras opciones que hayamos instalado. En la figura 1.7 podemos ver las opciones en una instalación de Visual Studio 2008 Professional.

Si C# es el lenguaje con el que habitualmente trabajaremos, seguramente elegiremos la opción del entorno de desarrollo para programadores de Visual C# (yo siempre suelo seleccionar la opción general de desarrollo).

La diferencia, entre las distintas opciones ofrecidas al iniciar por primera vez el IDE de Visual Studio, es la forma de mostrar las ventanas, las opciones que encontraremos en los diferentes cuadros de diálogo y las combinaciones de teclas para acceder a diferentes opciones.

El que yo seleccione las opciones generales en lugar de la específica de un lenguaje, es más bien por costumbre de usar las combinaciones propias del IDE con el que trabajo. De hecho, yo particularmente suelo utilizar Visual Basic para crear la mayoría de mis proyectos, pero aún así siempre elijo la **Configuración general de desarrollo**, con idea de utilizar la configuración más habitual de Visual Studio. Cuando salió la primera versión de Visual Studio para .NET, sí que solía utilizar las combinaciones de teclas para Visual Basic, pero más que nada por costumbre, pues las usaba en Visual Basic 6.0 (y anteriores). El problema con el que me encontraba aparecía al trabajar con otra configuración diferente, ya que una vez que te acostumbras a usar ciertas teclas, algunas veces es complicado “cambiar el chip” y adaptarte a otras diferentes. Más allá de que sea una cuestión de gustos o costumbres, el usar unas combinaciones de teclas más generalizadas nos puede servir para comprender más rápido los trucos que nos encontremos en la red de redes o en otros libros, ya que muchas de esas combinaciones casi siempre suelen hacer referencia a las predeterminadas de Visual Studio.

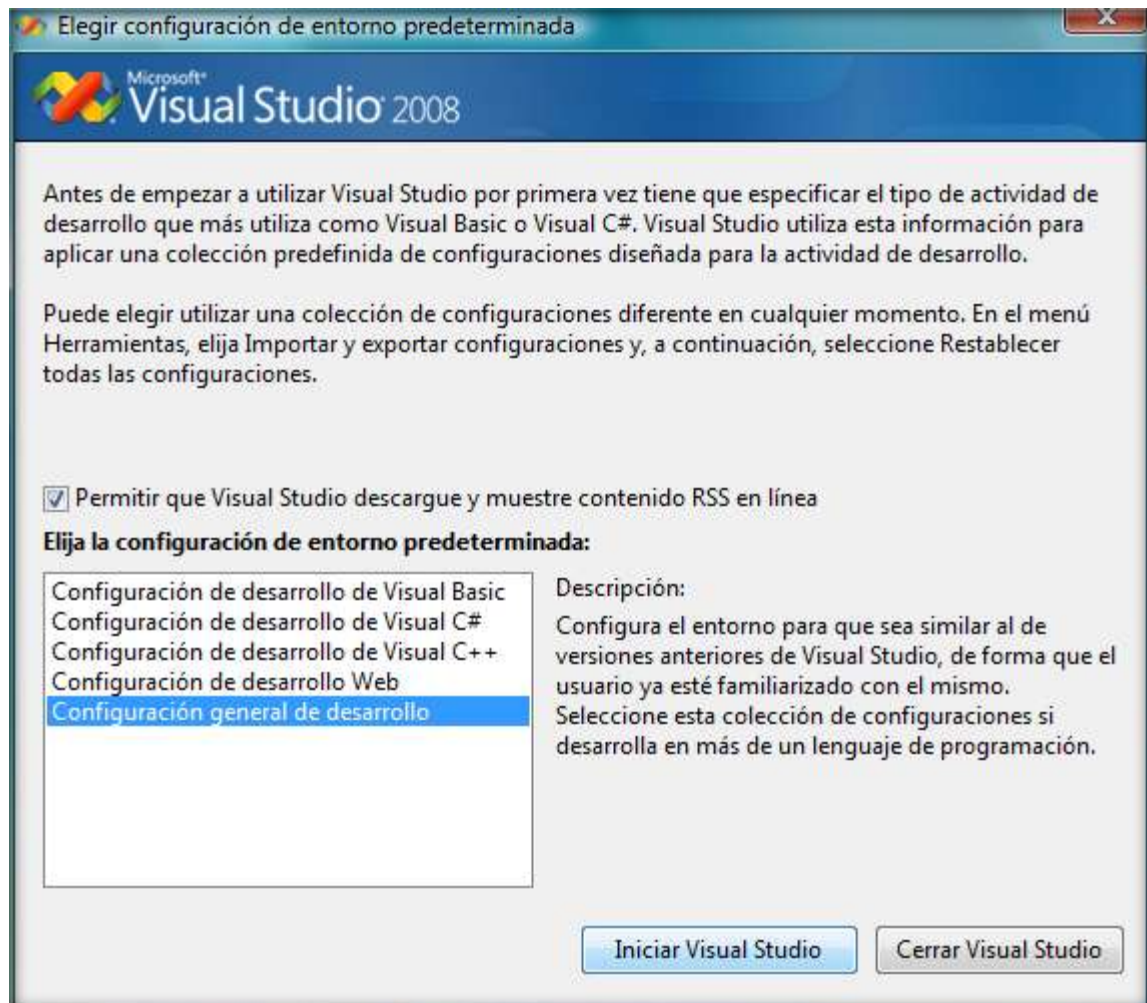


Figura 1.7. Al iniciar Visual Studio por primera vez podemos elegir la configuración del entorno

Pero no son solo las combinaciones de teclas las que cambian de una configuración a otra, también cambia las ventanas del IDE y en el caso de otros lenguajes, como Visual Basic, se ocultan ciertas características y opciones que es posible que sí utilicemos con más frecuencia de la que los que han decidido ocultarlas pensarán. En cualquier caso, mi recomendación es que elijamos la configuración general, particularmente si en nuestro equipo de desarrollo hay programadores que usan diferentes lenguajes de programación.

Independientemente del entorno que hayamos elegido al iniciar por primera vez Visual Studio (figura 1.7), podemos cambiarlo en cualquier momento desde la opción **Importar y exportar configuraciones** del menú **Herramientas**. Aunque no lo vamos a ver en detalle, en la figura 1.8 podemos ver que tenemos las mismas opciones que podemos elegir al iniciar Visual Studio.

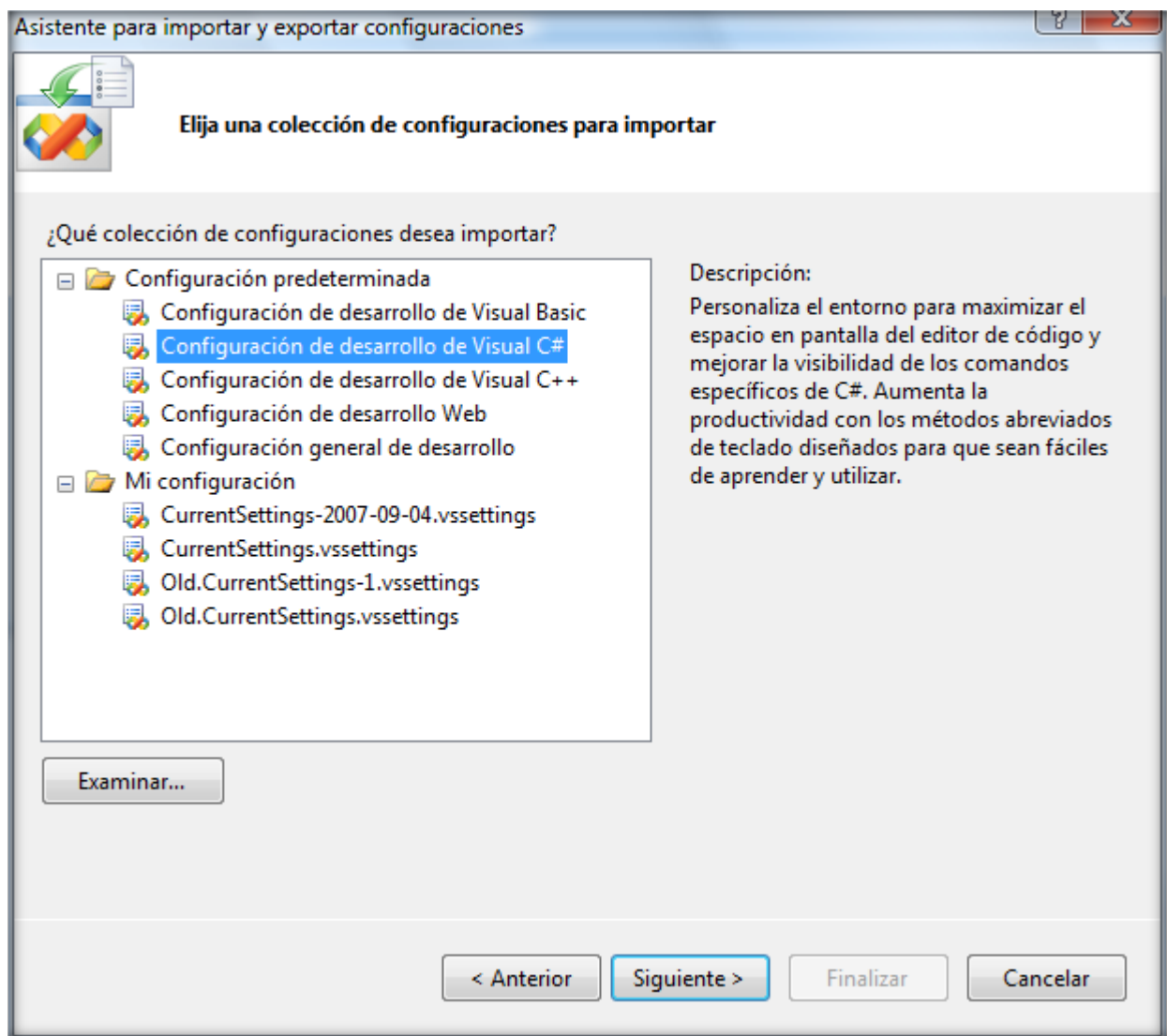


Figura 1.8. Las opciones del entorno en exportar e importar configuraciones

Nota

Estos cambios de las configuraciones del entorno de desarrollo solo los podemos hacer en la versión comercial, es decir, en Visual Studio. En las versiones gratuitas solo podemos usar la configuración propia del lenguaje de esa versión Express y lo único que podemos cambiar son las opciones normales del entorno.

Una vez hecho este repaso a las novedades referentes al entorno de desarrollo de Visual Studio 2008 (que en casi todos los casos es también aplicable a las versiones Express) pasemos a ver cuáles son las novedades en el lenguaje de la versión 3.0 de C#.

Nota

En realidad la numeración de las versiones de C# es algo más compleja. Cuando hablamos de la versión 3.0 nos estamos refiriendo a la versión incluida con .NET Framework 3.5, aunque el nombre interno del compilador distribuido con esa versión es: "Microsoft (R) Visual C# 2008 Compiler version 3.5.21022.8". De la misma forma, la versión del compilador que se distribuye con el runtime de .NET Framework 2.0 es: "Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.NNNN" las últimas cifras dependerán de la instalación que tengamos, por ejemplo, con el .NET 2.0 original mostrará .42, si la versión que tenemos es la 3.0 (la incluida en Windows Vista) ese número será .312 y la versión incluida en .NET 3.5 será .1434. Recordemos que cuando se instala el .NET Framework 3.5 se crea una carpeta especial para los compiladores de las versiones 2.0 y 3.5.

Para que no nos confundamos demasiado, y con idea de simplificar, llamaremos a las versiones con las numeraciones a las que estamos acostumbrados, por tanto la versión actual es la 3.0 y la anterior fue la 2.0.

Parte 2

Características generales del lenguaje

En esta segunda parte del libro, empezaremos viendo las novedades de C# 3.0, pero centrándonos más en el compilador, es decir, las novedades, que de alguna forma están más relacionadas con el compilador de C# 3.0 que con el resto de ensamblados que acompañan a .NET Framework 3.5.

Casi todas estas características (al igual que casi todas las novedades de C# 3.0) tienen su razón de ser en todo lo referente a LINQ.

En los primeros capítulos de esta parte nos centraremos en las características generales del lenguaje y después veremos otras novedades, que son más concretas para el uso y aprovechamiento de los nuevos ensamblados que acompañan a .NET Framework 3.5.

Nota

Al final de cada una de estas novedades indicaré con qué versión de .NET Framework se puede utilizar, ya que algunas de ellas solo estarán disponibles con ciertas versiones de .NET, pero otras las podremos usar con todas las versiones incluidas en la instalación de Visual Studio 2008 (o Visual C# 2008 Express Edition).

(Esta página se ha dejado en blanco de forma intencionada)

Capítulo 2

Nuevas características de C# 3.0 (I)

Introducción

En este segundo capítulo sobre las nuevas características de C# 3.0, nos centraremos principalmente en todas las novedades del lenguaje que se han incorporado en el compilador de C# que se distribuye con .NET Framework 3.5 y Visual Studio 2008 (o Visual C# Express), pero las que más directamente están relacionadas con el compilador en sí, aunque también hay cosas, que veremos en capítulos posteriores, que podrían haber entrado en esta clasificación, pero como tendremos ocasión de ver, esas otras novedades, en realidad están más relacionadas con las nuevas características que son necesarias o están más enfocadas en dar soporte a la tecnología LINQ, aunque a mor de ser sinceros, prácticamente todo lo nuevo que nos ofrece la tercera versión de C# tiene su razón de ser en dar soporte a LINQ, aún así, comprobaremos que sacando esas novedades del contexto del lenguaje de consultas integrado, también tienen su propia razón de ser.

En esta primera aproximación a las novedades de C# 3.0 veremos los siguientes temas:

- Inferencia de tipos en variables locales.
- Inicialización de objetos y colecciones.
- Métodos parciales.

Inferencia de tipos en variables locales

La inferencia automática de tipos en variables locales (que en la documentación de Visual Studio, concretamente en la guía de programación de C#, se denomina “Variables locales con asignación implícita de tipos”), es una característica de C# 3.0 por medio de la cual podemos dejar que sea el compilador el que se encargue de asignar el tipo que tendrá una variable a partir del valor que le estamos asignando. Pero debido a que C# es un lenguaje que requiere que todas las variables tengan un tipo de datos, tendremos que usar la nueva palabra clave **var** para que el compilador sepa que debe inferir el tipo que tendrá esa variable.

Esto supone que el compilador de C# será capaz de averiguar el tipo de una variable según el valor que estamos indicando en la parte derecha de la asignación, en ese caso, creará esa variable como si hubiésemos indicado el tipo de datos al declarar esa variable.

Por ejemplo, si tenemos esta asignación:

```
var nombre = "Guillermo";
```

El compilador (de forma interna) hará la declaración adecuada, que en este ejemplo, será:

```
string nombre = "Guillermo";
```

Es decir, averiguará qué tipo de datos estamos asignando a la variable y declarará esa variable con ese tipo de datos.

Todo esto es aplicable a cualquier tipo de variable, ya sean tipos por valor, tipos por referencia, tipos definidos en el propio .NET Framework o tipos que nosotros definamos. Y como veremos en el siguiente capítulo, hasta podrá inferir tipos anónimos.

Nota

La inferencia automática de tipos solo la podemos usar a nivel de variables locales, es decir, de las variables que usemos dentro de un método o propiedad, pero nunca a nivel de la clase o para el valor devuelto por una función o los parámetros de cualquier método o delegado.

Inferencia de tipos en los arrays (matrices)

En C# 3.0 también podemos inferir los tipos de datos que tendrán los *arrays*, al igual que en los tipos de datos simples, usaremos también la instrucción **var**, aunque en este caso, después del signo igual debemos indicar **new[]** seguido de los datos que queremos asignar al *array*. En el listado 2.1 vemos cómo definir un *array* usando la inferencia de tipos, en ese código la variable **nombres** es un *array* de tipo **string**.

¿Cuándo y dónde podemos usar la inferencia de tipos?

Como ya he comentado antes, la inferencia de tipos solo la podemos usar a nivel de procedimiento, es decir, con variables locales, aunque de esas variables locales debemos desechar las usadas como parámetros de esos procedimientos, entre otras cosas, porque al ser parámetros dejan de ser locales, aunque solo se usen localmente.

La inferencia automática de tipos puede llegar a convertirse en una práctica muy “adictiva”, ya que nos facilita la escritura de código, al no ser necesario indicar el tipo de las variables que declaremos. En este punto, como en todos los que se refieran a estilos de programación, siempre habrá quienes estén a favor o en contra.

Los que estén en contra seguramente apoyarán la idea de que si queremos escribir un código que sea fácilmente legible por cualquier programador (use el lenguaje que use), siempre se deberían indicar

los tipos de las variables que declaramos (debo reconocer que en un principio me “horrorizaba” la idea de declarar variables sin indicar el tipo de datos).

Por otro lado, son muchas las situaciones en las que podemos aprovechar que el propio compilador averigüe el tipo de datos de las variables. Y debido a que C# es un lenguaje que siempre necesita que todas las variables estén definidas con el tipo específico, y que la única forma de inferir el tipo de las variables es por medio de la palabra clave **var**, nos libera de los conflictos que pueden existir en otros lenguajes como en Visual Basic, en el que la sintaxis para declarar este tipo de variables automáticas, puede llevar a confusión frente a una programación *atipada*.

Y si estamos usando Visual Studio para crear nuestros proyectos, tenemos la ventaja de que en cualquier momento podemos saber qué tipo de datos tiene una de las variables en las que hayamos inferido el tipo, ya que en el editor de Visual C#, al pasar el puntero del ratón por una variable, nos indicará de qué tipo es (ver la figura 2.1).

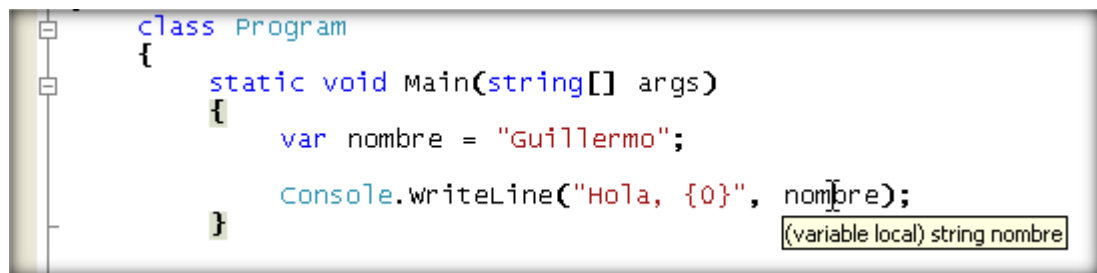


Figura 2.1. El IDE de Visual C# nos indica siempre el tipo de datos de las variables

Por otro lado, si esas variables las vamos a usar a nivel de procedimiento (es decir, son internas a un método o propiedad) no habrá problemas de que esa inferencia de tipos se propague fuera de ese procedimiento en el que usemos este nuevo automatismo que el compilador de C# 3.0 nos ofrece.

Un sitio propicio al uso de la inferencia de tipos es en los bucles, ya sean del tipo **for** o **foreach**, pues en ambos casos, el compilador “adivinará” siempre el tipo de la variable que usemos en el bucle, tal como vemos en el listado 2.1.

```
for(var i = 1; i < 11; i++)  
    Console.WriteLine(i);  
  
var nombres = new[] { "Pepe", "Luis", "Eva" };  
  
foreach(var s in nombres)  
    Console.WriteLine(s);
```

Listado 2.1. Inferencia de tipos en bucles

Lo mismo es aplicable a las variables que declaremos dentro de esos bucles (o en cualquier otra parte), además de que los tipos inferidos no solo son tipos “simples”, sino que esos tipos pueden ser de cualquier tipo de datos, al menos siempre que el compilador de C# sea capaz de averiguarlo, y si no es capaz, cuando compilemos el código nos avisará con un error.

En el listado 2.2 vemos cómo se puede inferir el tipo a partir de asignaciones que no usan tipos de datos en forma de constantes sino que se infieren a partir del tipo devuelto por una propiedad o un método.

```
var skey = TextBox2.Text;  
var tvn = TreeView1.Nodes.Find(skey, true);
```

Listado 2.2. Inferencia de tipos a partir de propiedades y métodos

En el primer caso, la propiedad **Text** del control **TextBox** devuelve un valor de tipo **string**, por tanto, la variable **sKey** es de tipo **string**. En la segunda línea de código, el método **Find** de la propiedad **Nodes** del control **TreeView** devuelve un *array* del tipo **TreeNode**, por tanto, el compilador de C# sabe que ése debe ser el tipo de datos de la variable **tvn**.

Cuando veamos las novedades de C# 3.0 referente a LINQ, tendremos más ocasiones de ver ejemplos de variables que “infieren” el tipo según el valor que asignemos. Ya que, en realidad, la razón de ser de esta característica ha sido principalmente dar soporte a todo lo relacionado con LINQ.

Versiones

Esta característica la podemos usar con cualquiera de las versiones de .NET Framework soportadas por Visual Studio 2008.

Inicialización de objetos y colecciones

La inicialización de objetos y colecciones son otras de las novedades incorporadas al lenguaje para facilitar la programación con LINQ, por eso hablaremos de esta característica en más de una ocasión.

¿Qué significa la inicialización de objetos y colecciones?

La inicialización de objetos nos permite crear nuevas instancias de objetos asignando valores a las propiedades de nuestra elección al mismo tiempo que creamos el objeto en cuestión.

Por ejemplo, si tenemos una clase llamada **Cliente** que tiene las propiedades habituales, para almacenar el nombre, los apellidos, etc., podemos crear nuevos objetos de ese tipo al estilo al que nos tiene acostumbrado C#, es decir, primero creamos la instancia de la clase y después asignamos los valores a las propiedades, tal como vemos en el listado 2.3.

```
cliente cli = new cliente();  
cli.Nombre = "Guillermo";
```



```
cli.Apellidos = "Som Cerezo";  
cli.Correo = "guillermo@nombres.com";
```

Listado 2.3. Creación y asignación de un objeto de la forma clásica

En C# 3.0 esta inicialización y asignación la podemos simplificar uniendo en una sola instrucción el código mostrado en el listado 2.3. Esta nueva forma es la que podemos ver en el código del listado 2.4.

```
cliente cli = new cliente  
{  
    Nombre = "Guillermo",  
    Apellidos = "Som Cerezo",  
    Correo = "guillermo@nombres.com"  
};
```

Listado 2.4. Creación y asignación de un objeto usando la inicialización de objetos de C# 3.0

Como vemos en el listado 2.4, lo que hacemos es una especie de mezcla del código mostrado en el listado 2.3, pero en lugar de usar la variable con la propiedad que queremos asignar, todo el código de asignación lo incluimos en un bloque, que por conveniencia (y porque Visual Studio le da ese formato) se ha separado en líneas independientes, pero en realidad lo podemos escribir todo en uno, siempre que lo hagamos de esa misma forma.

Las propiedades que podemos indicar después de las llaves pueden ser cualquiera de las que defina la clase y, por supuesto, no es necesario asignarlas todas, solo las que estimemos oportuno.

Inicialización de objetos con clases que definan solo constructores con parámetros

Si la clase que queremos inicializar de la forma que hemos visto en el listado 2.4 define constructores con parámetros, tendremos que respetar esa exigencia de la clase. En el ejemplo anterior, la clase *Cliente* define un constructor sin parámetros (en realidad el predeterminado), por tanto, podemos usarla tal como hemos visto. Si la clase define solo un constructor con parámetros, por ejemplo para asignar el número de cliente, esto supondría que es una condición sin la cual no podemos crear una nueva instancia de esa clase; es obvio que estamos obligados a usar ese constructor con parámetros para poder crear una instancia de dicha clase, ya sea usando la forma tradicional o la nueva. En este caso, el código del listado 2.5 sería la forma que tendríamos que usar para inicializar el objeto correctamente.

```
cliente2 cli2 = new cliente2("1234567890") { Nombre = "Guillermo" };
```

Listado 2.5. Inicialización de una clase que requiere el uso de un constructor con parámetros

Es decir, si la clase exige que la inicialicemos con un constructor que reciba algún argumento, debemos hacerlo.

Inicialización de objetos e inferencia de tipos

Una vez que ya sabemos cómo inicializar los objetos con C# 3.0, y sabiendo que el compilador también es capaz de “adivinar” el tipo de datos que tendrá una variable al asignarle un valor, podemos crear esos dos objetos (listados 2.4 y 2.5) tal como vemos en los listados 2.6 y 2.7.

```
var cli = new Cliente
{
    Nombre = "Guillermo",
    Apellidos = "Som Cerezo",
    Correo = "guillermo@nombres.com"
};
```

Listado 2.6. Creación y asignación de un objeto usando la inicialización de objetos y la inferencia de tipos de C# 3.0

```
var cli2 = new Cliente2("1234567890") { Nombre = "Guillermo" };
```

Listado 2.7. Inicialización de objetos e inferencia de tipos con tipo que define un constructor parametrizado

Evidentemente esto es posible porque el compilador sabe qué tipo de datos hay después del signo igual, por tanto, las variables que reciben esas asignaciones serán del tipo correcto.

Inicializaciones de arrays y colecciones

C# 3.0 soporta nativamente la inicialización de *arrays* y colecciones, es decir, nos permite asignar elementos a una variable que contendrá un *array* o una colección al mismo tiempo que la definimos.

En el listado 2.8 vemos cómo definir un e inicializar un *array* del tipo *Cliente*, y en el listado 2.9 declaramos e inicializamos una colección del tipo *List<Cliente>*. En esos dos listados, se supone que las variables *cli* y *cli2* ya están instanciadas con objetos del tipo *Cliente*.

```
Cliente[] clis = new[]
{
    cli, cli2,
    new Cliente { Nombre = "Juan" },
    new Cliente { Nombre = "Eva" }
};
```

Listado 2.8. Inicialización de un array

```
var clis = new List<Cliente>
{
    cli, cli2,
    new Cliente{ Nombre = "Juan"},
    new Cliente { Nombre="Eva"}
};
```

Listado 2.9. Inicialización de una colección con inferencia de tipos

Como vemos en el listado 2.9, la forma de inicializar una colección es muy parecida a cómo se inicializan los objetos, sólo que en lugar de asignar valores a propiedades, indicamos los elementos que queremos añadir. Por supuesto, si no queremos que el compilador infiera el tipo que tendrá la variable *clis*, siempre podemos usar la forma tradicional de declarar las variables, tal como vemos en el listado 2.10.

```
List<Cliente> clis = new List<Cliente>
{
    cli, cli2,
    new Cliente{ Nombre = "Juan"},
    new Cliente { Nombre="Eva"}
};
```

Listado 2.10. Inicialización de una colección sin usar inferencia de tipos

Sin embargo, en el caso de los *arrays* (ver listado 2.8), puede parecer que esto mismo es lo que podíamos hacer en las versiones anteriores, solo que sin necesidad de indicar el tipo de datos después del signo de asignación. Y en cierto modo así podría parecer, pero si además de la inicialización del *array* usamos la inferencia de tipos, sí que notamos un ahorro a la hora de escribir código, tal como podemos apreciar en el listado 2.11.

```
var clis = new[]
{
    cli, cli2,
    new Cliente{ Nombre = "Juan"},
    new Cliente { Nombre="Eva"}
};
```

Listado 2.11. Inicialización de arrays e inferencia de tipos

Versiones

Esta característica la podemos usar con cualquiera de las versiones de .NET Framework soportadas por Visual Studio 2008.

Métodos parciales

Los métodos parciales son una forma de definir prototipos de métodos, los cuales después podemos implementar en el código o simplemente no implementarlos. Si lo implementamos, se usará esa implementación y si no, el código en el que se haga la llamada a ese método parcial será totalmente ignorado.

Para comprender mejor esta característica veamos cómo tendríamos que hacerlo con las versiones de Visual C# anteriores a la versión 3.0.

En Visual C# 2005 o anterior, podemos crear métodos dentro de condicionales de compilación, es decir, si queremos que se incluya ese método en el código final de nuestra aplicación, la constante usada para la compilación condicional debe tener el valor indicado en la evaluación que usemos.

En el listado 2.12 vemos un trozo de código en el que definimos un método que está dentro de una condición usada por el compilador para saber si se incluye o no ese código.

```
#if DEBUG
    static void soloEnDebug()
    {
        // Lo que haga este método
    }
#endif
```

Listado 2.12. Un método que solo se compilará si la constante DEBUG está definida

En el código del listado 2.12 estoy usando una constante de compilación definida por el propio entorno de desarrollo cuando el proyecto está en modo depuración. Cuando esa constante no esté definida, ese código no se compilará, por tanto, es lógico pensar que si hacemos una llamada a ese método, dicha llamada también debería estar dentro de una comprobación similar, ya que solo debe usarse ese método cuando esté definida la constante **DEBUG**. El código del listado 2.13 muestra una forma de referirse a ese método.

```
#if DEBUG
    soloEnDebug();
#endif
```

Listado 2.13. Llamada al método sólo cuando estemos en la misma condición que al definirlo

Como podemos suponer, si esa llamada al método *soloEnDebug* la tenemos en varias partes de nuestro código, en todas y cada una de esas llamadas tendremos que hacerlo usando el condicional de compilación, y si no lo hiciéramos así, en el momento que no se cumpla la condición indicada para declarar el método, el propio compilador nos avisará de que dicho método no está definido.

Nota

*Recordemos que todo lo que escribamos entre **#if condición ... #endif** solo se incluirá en el código final si la condición indicada se cumple, si no se cumple, ese código será eliminado y por tanto, no incluido en el ensamblado final.*

Con los métodos parciales, todo este “lío” se simplifica de tal forma que podemos definir el prototipo del método, usando la construcción mostrada en el listado 2.14.

```
partial void unMetodoParcial();
```

Listado 2.14. Definición de un método parcial

La definición de un método parcial se hace usando la palabra reservada **partial**, y solo debemos definir la firma que tendrá el método, pero no el código a usar dentro de ese método. El código que dicho método usará lo definiremos en cualquier otra parte del código, y esa definición puede estar incluida dentro de alguna condición de compilación, es decir, la definición la podemos hacer como vemos en el listado 2.15, que es muy similar al que ya vimos en el listado 2.12.

```
#if DEBUG
    partial void unMetodoParcial()
    {
        // Lo que tenga que hacer este método
    }
#endif
```

Listado 2.15. Definición del código del método parcial

Como vemos, esto último no se diferencia del código que ya vimos para las versiones de C# que no soportan esta nueva característica de creación de métodos parciales.

Nota

Tanto la definición de la firma como la implementación del método parcial deben estar dentro de una clase o estructura parcial.

La diferencia está en que ahora, si queremos usar ese método parcial, no tenemos que incluir esas llamadas en ningún condicional, es decir, lo usaremos directamente, como cualquier otra llamada a un método:

```
unMetodoParcial();
```

Cuando se compile el proyecto, el compilador de C# comprobará si se ha definido el método (es decir, no solo está la definición del prototipo), y si ese método está definido, lo deja todo como está. Si el método no se ha definido, lo que hace es quitar todas las llamadas que haya, por tanto, el resultado es como si nunca hubiésemos escrito ese código.

Esto último es importante tenerlo en cuenta: se quitan todas las llamadas al método. Por tanto, si a ese método se le pasan algunos argumentos, y esos argumentos son llamadas a funciones o a cualquier otra parte de nuestro código que realice alguna tarea, simplemente se ignorará, ya que el código de esa llamada al método parcial que no se ha implementado no existe, no está, es como si nunca lo hubiésemos escrito.

Para entenderlo mejor, es como si todas las llamadas a los métodos parciales estuviesen dentro de un condicional de compilación.

Condiciones para los métodos parciales

Los métodos parciales son un poco restrictivos; veamos una lista de las condiciones que deben cumplir los métodos parciales:

- Solo pueden ser métodos de tipo **void** (no pueden devolver un valor).
- Hay que definirlos como privados (pero sin indicar el modificador **private**).
- Usaremos la instrucción **partial** tanto en el prototipo como en la implementación.
- Los métodos parciales pueden tener parámetros **ref** pero no **out**.
- No se puede crear un delegado para un método parcial.

Los métodos parciales para usar con delegados

Como vemos en la lista de condiciones para usar los métodos parciales, se indica expresamente que no se puede crear un delegado para usarlo con un método parcial, pero esto en realidad no es totalmente cierto, ya que sí se pueden usar, pero en ese caso la “magia” de los métodos parciales desaparece, por tanto, si esa es nuestra intención, dicha llamada o uso del método parcial con un delegado debemos realizarla solamente si el método está definido, y la mejor forma de asegurarnos de que eso sea así es incluyendo ese código en un condicional de compilación, que coincida con el que hayamos usado a la hora de definir el método. En cualquier caso, el compilador de C# nos avisará si no encuentra la definición del método.

En el código del listado 2.16 tenemos un ejemplo de esto que acabo de comentar. Con este código, el compilador nos avisará que el método indicado como argumento del delegado no está definido. Por tanto, en este caso especial de uso de un método parcial, lo más correcto sería definir el método *PruebaDelegado* dentro de un condicional como el que define el método *paraUnDelegado*, tal como se muestra en el listado 2.17. Por supuesto, si queremos usar ese código, la constante *PROBANDO* debe estar definida, además de que todas las llamadas al método *PruebaDelegado* también deben estar dentro de ese mismo condicional de compilación, es decir, el uso de métodos parciales no nos es de ninguna utilidad cuando éstos se usan como argumentos de un delegado.

```
        partial void paraUnDelegado();

#if PROBANDO
    partial void paraUnDelegado()
    {
        Console.WriteLine("Desde paraUnDelegado");
    }
#endif

    public void PruebaDelegado()
    {
        System.Threading.Thread t =
            new System.Threading.Thread(paraUnDelegado);
        t.Start();
    }
}
```

Listado 2.16. En ciertas circunstancias los métodos parciales pierden su magia

```
#if PROBANDO
    public void PruebaDelegado()
    {
        System.Threading.Thread t =
            new System.Threading.Thread(paraUnDelegado);
        t.Start();
    }
#endif
```

Listado 2.17. Si usamos un método parcial con un delegado, mejor ponerlo en un condicional de compilación

El IDE de Visual C# 2008 y los métodos parciales

Cuando tenemos prototipos de métodos parciales en una clase y escribimos la instrucción **partial** desde el entorno de desarrollo de Visual C# 2008, éste nos muestra los métodos parciales que podemos definir. En la figura 2.2 vemos una lista de los métodos parciales que tengo en el proyecto de ejemplo.

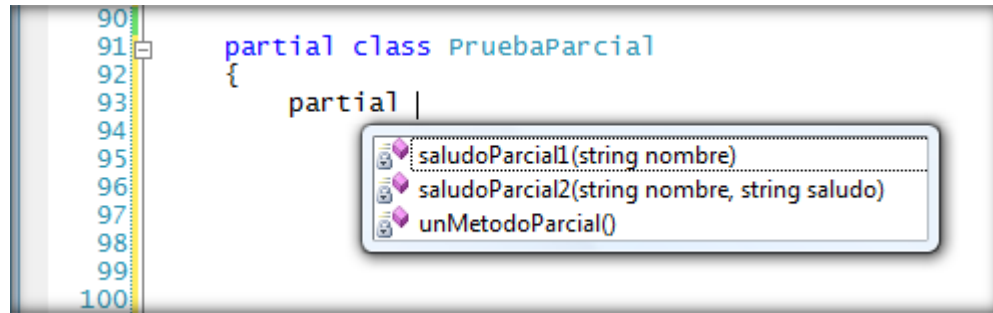


Figura 2.2. IntelliSense nos muestra los métodos parciales que podemos definir

La lista puede que no muestre todos los métodos parciales, ya que depende de dónde usemos la instrucción **partial**, por ejemplo, no incluirá los prototipos que tengamos definidos unas líneas más adelante o los que ya estén implementados en otra parte del código, al menos si están implementados en otra definición de la misma clase (o tipo) parcial o en algunas de las líneas anteriores de esa misma clase parcial. No es perfecto, pero nos ayuda a saber cuáles son los prototipos que tenemos definidos.

Versiones

Esta característica la podemos usar con cualquiera de las versiones de .NET Framework soportadas por Visual Studio 2008.

Capítulo 3

Nuevas características de C# 3.0 (II)

Introducción

En este capítulo veremos una de las novedades que se han incorporado en C# de forma exclusiva:

- Las propiedades autoimplementadas.

Propiedades autoimplementadas

Esta es una de esas novedades que están llamadas a pasar sin pena ni gloria, y no es porque no sea una característica que de por sí sea interesante, si no porque comparada con el resto de novedades que incorpora C# 3.0 es poco llamativa o al menos así lo han expresado algunos usuarios de C#, pero creo que lo mejor es que veamos qué nos ofrece y cómo usarla, y después decidamos si puede o no tener utilidad.

Las propiedades autoimplementadas nos permiten definir propiedades sin necesidad de utilizar el código completo al que estamos acostumbrados, por ejemplo, en el listado 3.1 tenemos la definición formal de una propiedad, como cualquier propiedad, existe un campo que está asociado a la misma y servirá para contener el valor de esa propiedad, también tiene los dos bloques habituales para leer el valor de la propiedad (bloque **get**) y para asignar un valor (bloque **set**).

```
private string m_Nombre;  
public string Nombre  
{  
    get { return m_Nombre; }  
    set { m_Nombre = value; }  
}
```

Listado 3.1. Definición de una propiedad al estilo clásico

La mayoría de las veces este será el código “típico” que usaremos para definir una propiedad. Y debido a que esa es la forma más habitual de hacerlo, en C# 3.0 se ha facilitado la creación de este tipo de propiedades, ahora podemos definir una propiedad sin necesidad de crear ese código completo, y la forma de hacerlo es al estilo de cómo definimos una propiedad en una interfaz, en el listado 3.2 vemos cómo definir una propiedad usando esta nueva característica.

```
public string Apellidos { get; set; }
```

Listado 3.2. Definición de una propiedad autoimplementada

Como podemos apreciar en el código del listado 3.2, la forma de definir las propiedades autoimplementadas es como en las interfaces, solo que en este caso, debemos indicar el ámbito que tendrá, al menos si no queremos aplicarle el predeterminado, que como ya sabemos sería **private**.

Al definir una propiedad de esta forma, en realidad el compilador la creará tal como vimos en el listado 3.1, es decir, creará un campo privado, que lo ocultará y por tanto no nos permitirá usarlo en el resto de la clase, también creará los dos bloques de código.

Este último comentario es importante, ya que **siempre** se crean los dos bloques (**get** y **set**), por tanto, no podremos definir propiedades de solo lectura (solo se define el bloque **get**) o de solo escritura (solo se define el bloque **set**). Pero no está todo perdido, ya que, como sabemos, podemos cambiar el ámbito de esos bloques, de forma que tengan uno de menor cobertura (o alcance) que el indicado al definir la propiedad. Por tanto, si queremos crear una variable autoimplementada que “simule” a una propiedad de solo lectura, lo haremos tal como vemos en el listado 3.3.

```
public string NIF { get; private set; }
```

Listado 3.3. Definición de una propiedad autoimplementada que simula a una de solo lectura

Como vemos, las propiedades autoimplementadas hacen más compacto el código.

El problema es que no podemos agregar código de validación (habitualmente en el bloque **set**), ni podremos aplicar atributos al campo privado (ya que no tenemos acceso al mismo), aunque sí a las propiedades autoimplementadas (contrariamente a lo que indica la documentación).

En cualquier caso, si necesitamos tener mayor control sobre las propiedades, siempre nos queda el recurso de definir las de la forma habitual, además usando las opciones de refactorización es fácil crear un campo y convertirlo en una propiedad, para ello solo tenemos que seleccionar el campo privado, seleccionar la opción **Encapsular campo** del menú **Refactorizar** y dicho campo se convertirá en una propiedad, tal como vemos en la figura 3.1.

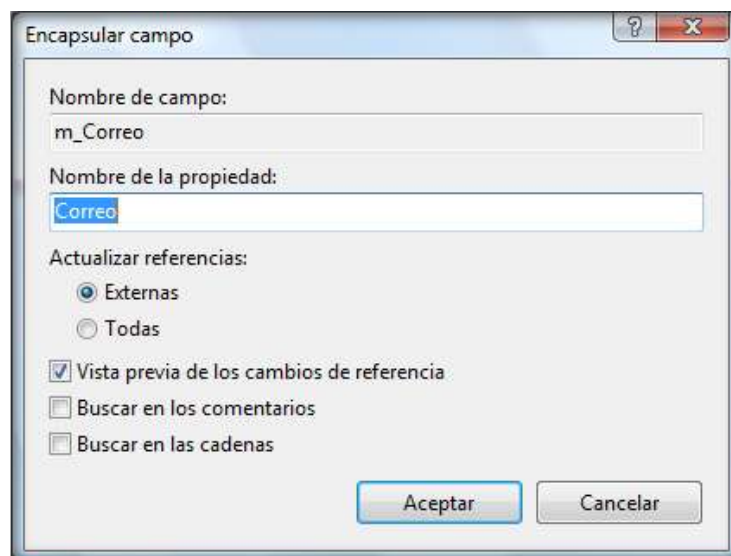


Figura 3.1. La opción de encapsular campo del menú Refactorizar

Versiones

Esta característica la podemos usar con cualquiera de las versiones de .NET Framework soportadas por Visual Studio 2008.

(Esta página se ha dejado en blanco de forma intencionada)

Capítulo 4

Nuevas características de C# 3.0 (III)

Introducción

En este capítulo continuamos con las novedades de C# 3.0, pero en esta ocasión solo las podremos usar con la versión 3.5 .NET Framework.

Estas características (al igual que casi todas las novedades de C# 3.0) tienen su razón de ser en todo lo referente a LINQ; pero antes de ver qué es LINQ y cómo usarlo desde nuestro lenguaje favorito, veámoslas con detalle para que nos resulte más fácil comprender lo que he dejado para el final de este repaso en profundidad a las novedades de C# 3.0.

Las novedades que veremos en este capítulo son:

- Los tipos anónimos.
- Las expresiones *lambda* o funciones anónimas.

Tipos anónimos

Esta característica nos permite crear nuevos tipos de datos (clases) sin necesidad de definirlos de forma separada, es decir, podemos crear tipos de datos “al vuelo” y usarlos, en lugar de crear una instancia de un tipo existente.

Esta peculiaridad nos va a permitir crear clases (los tipos anónimos deben ser tipos por referencia) cuando necesitemos crear un tipo de datos que tenga ciertas propiedades y que no necesiten estar relacionadas con una clase existente. En realidad, esto es particularmente útil para la creación de tipos de datos “personalizados” devueltos por una consulta de LINQ, aunque no necesitamos saber nada de LINQ para trabajar con estas clases especiales, por eso en este capítulo veremos cuáles son las peculiaridades de los tipos anónimos, pero usados de una forma más generalizada; cuando lleguemos al capítulo de LINQ, retomaremos este tema para entenderlos mejor en el contexto de LINQ.

Como sabemos, todos los tipos de datos pueden definir métodos, propiedades, eventos, etc. Sin embargo, en los tipos anónimos solo podemos definir propiedades. Las propiedades de los tipos anónimos pueden ser de solo lectura o de lectura/escritura. Este último aspecto, diferencia los tipos anónimos de Visual Basic de los que podemos crear con C#, ya que en C# las propiedades siempre son de solo lectura.

Los tipos anónimos, al igual que el resto de clases de .NET, se derivan directamente de **object**, por tanto, siempre tendrán los métodos que esa clase define, y éstos serán los únicos métodos que

tendrán. El compilador creará definiciones de esos métodos, que tendrán en cuenta las propiedades del tipo anónimo, pero como veremos en un momento, esas definiciones dependerán de cómo declaramos las propiedades.

Definir un tipo anónimo

Como su nombre indica, los tipos anónimos no están relacionados con ningún tipo de datos existente; éstos siempre se crean asignando el tipo a una variable y ésta infiere el tipo, que en realidad no es anónimo, ya que el compilador de C# le da un nombre; lo que ocurre es que no tenemos forma de saber cuál es ese nombre, por tanto, nunca podremos crear nuevas variables de un tipo anónimo de la forma tradicional. Para clarificar la forma de crear los tipos anónimos veamos un ejemplo.

En el listado 4.1 tenemos la definición de un tipo anónimo, el cual asignamos a una variable. Como vemos en ese código, para crear el tipo anónimo tenemos que usar la inicialización de objetos, pero a diferencia de usarla como vimos en el capítulo 2, no podemos indicar el tipo de datos, ya que es un tipo anónimo.

```
var ta1 = new { Nombre = "Guille", Edad = 51 };
```

Listado 4.1. Definición de un tipo anónimo

El código del listado 4.1 crea una clase anónima con dos propiedades; los tipos de datos de dichas propiedades se infieren según el valor que le asignemos.

La variable **ta1** es una instancia del tipo anónimo que acabamos de definir y tendrá dos propiedades, una llamada **Nombre** que es de tipo **string** y la otra, **Edad** que es de tipo **int**.

Cuando definimos un tipo anónimo en C#, las propiedades **siempre** son de solo lectura, por tanto, no podemos cambiar el contenido de las mismas, sabiendo esto, el código del listado 4.2 dará un error de compilación indicando que la propiedad **Edad** es de solo lectura (no porque el compilador sepa la edad de Guille).

```
ta1.Edad = 50;
```

Listado 4.2. Las propiedades de los tipos anónimos son de solo lectura

Como ya comenté, C# redefine los métodos heredados de **object** para adecuarlos al tipo de datos que acabamos de definir. De estos métodos, solo la implementación de **ToString** siempre tiene el mismo formato, que consiste en devolver una cadena cuyo contenido tendrá esta forma: **{ Propiedad = Valor[, Propiedad = Valor] }**, es decir, dentro de un par de llaves incluirá el nombre de la propiedad, un signo igual y el valor; si hay más de una propiedad, las restantes las mostrará separadas por una coma. Una llamada al método **ToString** de la variable **ta1** devolverá el siguiente valor:

```
{ Nombre = Guille, Edad = 51 }
```

¿Cómo de anónimos son los tipos anónimos?

En realidad, los tipos anónimos no son “totalmente” anónimos, al menos en el sentido de que sí tienen un nombre, lo que ocurre es que ese nombre no estará a nuestra disposición, ya que el compilador genera un nombre para uso interno. El tipo anónimo definido en el listado 4.1 tendrá el nombre `<>f__AnonymousType0`, pero ese nombre solo es para uso interno, por tanto, no podremos usarlo para crear nuevas clases de ese mismo tipo (incluso si pudiéramos definir variables con ese nombre, el compilador no lo permitiría, ya que se usan caracteres no válidos para los nombres de variables o tipos).

Pero si nuestra intención es crear más variables del mismo tipo anónimo, lo único que tendremos que hacer es definir otra variable usando un tipo anónimo con las mismas propiedades, de esa forma, el compilador reutilizará el tipo anónimo que previamente había definido.

En el código del listado 4.3 la variable `ta2` también tendrá una instancia de un tipo anónimo, pero debido a que las propiedades se llaman igual, son del mismo tipo y están en el mismo orden que el definido en el listado 4.1, el compilador no creará otro tipo anónimo, sino que reutilizará el que definió para la variable `ta1`.

```
var ta2 = new { Nombre = "David", Edad = 28 };
```

Listado 4.3. Este tipo anónimo es el mismo que el del listado 4.1

Cada vez que el compilador se encuentra con un tipo anónimo que tiene las mismas propiedades (nombre, tipo y orden de declaración) de otro definido previamente en el mismo ensamblado, usará el que ya tiene definido, pero si cualquiera de esas tres condiciones cambia, creará un nuevo tipo.

Por ejemplo, el compilador creará un nuevo tipo para el que definimos en el listado 4.4, porque aunque tiene dos propiedades que se llaman igual y son del mismo tipo que el de los listados anteriores, el orden en que están definidas esas propiedades es diferente, por tanto, lo considera como otro tipo de datos.

```
var ta3 = new { Edad = 24, Nombre = "Guille" };
```

Listado 4.4. Si el orden de las propiedades es diferente, se crea otro tipo anónimo

La explicación a esta extraña forma de actuar es porque, en realidad, el compilador define los tipos anónimos como tipos *generic* y en estos dos casos concretos, la definición de los dos tipos es como vemos en el listado 4.5. Ese código es el generado por el compilador, que como sabemos lo genera usando el lenguaje intermedio de Microsoft (MSIL, *Microsoft Intermediate Language* o IL) que es el que finalmente compila el CLR cuando ejecutamos la aplicación, y que tiene una sintaxis muy

parecida a C#; en el listado 4.6 tenemos el equivalente al estilo de cómo lo definiríamos con instrucciones propias de C#.

```
class <>f__AnonymousType0`2<string, int32>
class <>f__AnonymousType1`2<int32, string>
```

Listado 4.5. Definición de los dos tipos anónimos en lenguaje IL

```
class f__AnonymousType0<string, int> { ... }
class f__AnonymousType1<int, string> { ... }
```

Listado 4.6. Definición de los dos tipos anónimos al estilo de C#

En realidad, la definición de los tipos anónimos no usan los tipos de datos explícitos de las propiedades, ya que el compilador define esos tipos como *generic*. Lo que en realidad ocurre, es que, aunque las propiedades se llamen igual, al estar definidas en un orden diferente, la asignación de los tipos *generic* usados se invertirían y el resultado no sería el deseado.

Para que lo entendamos mejor, el pseudocódigo al estilo de C# de la definición de esos dos tipos sería como el mostrado en el listado 4.7, y como vemos, la propiedad **Edad** de la primera clase es del tipo **T0**, mientras que en la segunda clase es del tipo **T1**, por tanto, si se usara un solo tipo anónimo, los tipos usados no coincidirían en las dos declaraciones.

```
class f__AnonymousType0<T0, T1>
{
    public T0 Nombre;
    public T1 Edad;
}

class f__AnonymousType1<T0, T1>
{
    public T0 Edad;
    public T1 Nombre;
}
```

Listado 4.7. Los tipos anónimos se definen como tipos generic y cada propiedad es del tipo indicado según el orden de las declaraciones

Si solo tuviéramos la primera clase (**f__AnonymousType0**) y definiéramos dos variables, pero en la segunda declaración invertimos los tipos de los parámetros, el compilador nos daría un error, tal como vemos en la figura 4.1, ya que los valores asignados a las propiedades de la variable **ta5** no son de los tipos correctos.

Como truco nemónico, yo pienso en los tipos anónimos como si fueran sobrecargas de un mismo método, si se puede crear una nueva sobrecarga, es que el tipo anónimo será diferente; y si puedo usar una de las sobrecargas existentes es que ya existe un tipo anónimo que se puede reutilizar (ya sabemos que en las sobrecargas lo que se tiene en cuenta son el número de parámetros y los tipos de

esos parámetros, y solo se permiten sobrecargas cuando la firma no coincide con una existente), pero como veremos en la siguiente sección, al evaluar esas “sobrecargas” debemos tener en cuenta otras consideraciones, entre las que se incluye que el nombre de los parámetros también sea el mismo (y esté definido en el mismo orden).

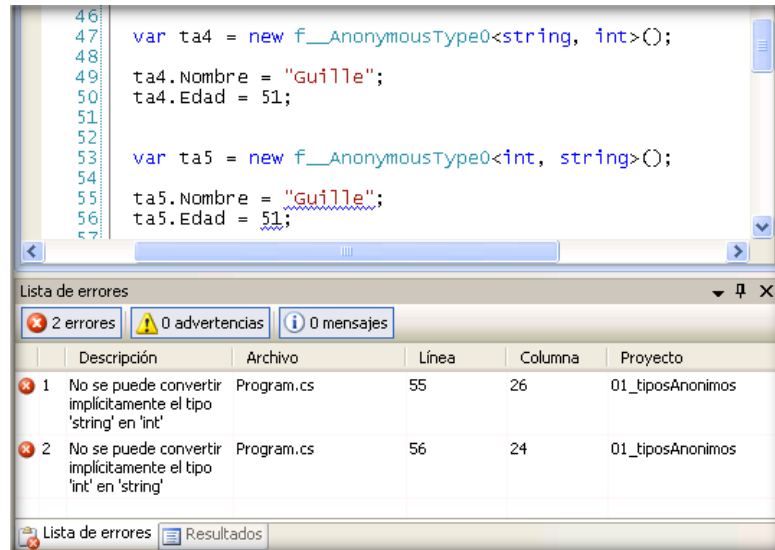


Figura 4.1. Error al asignar los valores a las propiedades del supuesto tipo anónimo

Comprobación de igualdad en los tipos anónimos

Los tipos anónimos se derivan directamente de **object**, por tanto definen los métodos de este tipo elemental, entre los métodos heredados tenemos dos que se utilizan de forma especial a la hora de comparar dos instancias de dos tipos anónimos: **Equals** y **GetHashCode**.

El primero servirá para comparar dos instancias de tipos anónimos, la comparación de esas instancias se realizará comprobando solo el contenido de las propiedades de solo lectura (que como sabemos son las únicas que podemos crear con C#). Para realizar esa comparación se utiliza el valor generado por el método **GetHashCode** en cada una de las propiedades de solo lectura.

En la comprobación de igualdad de dos tipos anónimos siempre se tendrán en cuenta los contenidos de todas las propiedades de esos dos tipos anónimos y dicha comprobación la podemos hacer tanto usando el método **Equals** como usando el operador de igualdad (**==**), al menos si estamos comparando dos objetos creados usando el mismo tipo anónimo, ya que si los tipos anónimos usados para crear esas dos instancias que queremos comparar son diferentes, solo podremos usar el método **Equals**, y siempre devolverá un valor falso.

En el listado 4.8 tenemos algunas definiciones de tipos anónimos y cómo realizar una comparación de igualdad en las instancias creadas, en estos ejemplos el valor devuelto en la comparación siempre será **false**, ya que ninguna de las tres instancias son exactamente iguales, además de que el tipo anónimo usados para crear las variables **ta6** y **ta7** es diferente al tipo anónimo de la variable **ta8**, por tanto, para comprar si el contenido de **ta6** y **ta8** son iguales debemos usar el método **Equals**, ya que al usar el operador de igualdad produciría un error de compilación indicando que: “El operador ‘==’ no se puede aplicar a operandos del tipo ‘AnonymousType#1’ y ‘AnonymousType#2’”.

```
var ta6 = new { ID = "003", Nombre = "C#" };
var ta7 = new { ID = "003", Nombre = "Visual C#" };

// La comparación de igualdad
// la podemos hacer de dos formas
// (siempre que sean instancias del mismo tipo anónimo)
var sonIguales = ta6.Equals(ta7);
Console.WriteLine(sonIguales);

sonIguales = (ta6 == ta7);
Console.WriteLine(sonIguales);

var ta8 = new { ID = 3, Nombre = "C#" };

// esto dará error
// sonIguales = (ta6 == ta8);

sonIguales = ta6.Equals(ta8);
Console.WriteLine(sonIguales);
```

Listado 4.8. Comparar instancias de tipos anónimos

Al ver ese error, es posible que nos parezca que no se puede usar el operador de igualdad porque son tipos anónimos diferentes, pero no es así, ya que las tres variables son del mismo tipo anónimo, lo que ocurre es que las instancias tienen diferentes tipos de datos para las propiedades, ya que en el fondo, los tipos anónimos son clases *generic*.

Usando pseudocódigo de C#, el tipo anónimo se definiría tal como vemos en el listado 4.9. Como vemos, es un tipo *generic* y el constructor recibe como parámetros los valores a asignar a las propiedades de solo lectura, los tipos de datos de esos parámetros serán de los tipos de datos que indiquemos en el constructor del tipo anónimo, por tanto, aunque en realidad la clase sea la misma, al instanciar los objetos lo haremos usando los tipos de datos que se indiquen en la creación de cada una de esas variables. Sabiendo esto, en el pseudocódigo del listado 4.10 vemos que los tipos de datos usados en las variables **ta6** y **ta7** son **<string, string>**, mientras que la variable **ta8** se crea usando **<int, string>**, por tanto, las instancias creadas en memoria manejan tipos de datos diferentes en las propiedades de ese tipo *seudo-anónimo*, de ahí que no podamos comparar con el operador de igualdad dos instancias del *supuestamente* mismo tipo anónimo. Aclarar que ese error al comparar dos tipos *generic* que usan tipos de datos diferentes en las propiedades es algo que no es exclusivo de los tipos anónimos, ya que si definimos el tipo de datos del listado 4.9 y lo usamos tal como vemos en el listado 4.10, tampoco permitirá comparar con el operador **==** las variables **ta6** y **ta8**.

```
internal sealed class f__AnonymousType0<T0_ID, T1_Nombre>
{
    // Los campos privados y las propiedades
    private readonly T0_ID m_ID;
    private readonly T1_Nombre m_Nombre;

    public T0_ID ID
    {
        get { return m_ID; }
    }
    public T1_Nombre Nombre
    {
        get { return m_Nombre; }
    }

    // El constructor
    public f__AnonymousType0(T0_ID ID, T1_Nombre Nombre)
    {
        this.m_ID = ID;
        this.m_Nombre = Nombre;
    }

    // Los métodos
    // Redefine ToString, Equals y GetHashCode
}
```

Listado 4.9. Seudocódigo del tipo anónimo usado en el listado 4.8

```
f__AnonymousType0<string, string> ta6;
ta6 = new f__AnonymousType0<string, string>("003", "C#");

f__AnonymousType0<string, string> ta7;
ta7 = new f__AnonymousType0<string, string>("003", "visual C#");

f__AnonymousType0<int, string> ta8;
ta8 = new f__AnonymousType0<int, string>(3, "C#");
```

Listado 4.10. Seudocódigo que usa el tipo definido en el listado 4.9

Como vemos, el compilador usará el mismo tipo anónimo siempre que las propiedades tengan los mismos nombres, pero las instancias finales dependerán del tipo de datos que usemos al crearlos.

¿Son iguales dos instancias del mismo tipo anónimo con los mismos datos?

Al hacer las comparaciones con instancias de los mismos tipos anónimos, hay que tener en cuenta que si el contenido de las propiedades es exactamente el mismo, el uso del método **Equals** producirá un resultado diferente al operador de igualdad. Esto es así porque en realidad al comparar tipos por referencia (los tipos anónimos siempre son tipos por referencia), el operador de igualdad comprueba si las dos instancias son la misma, mientras que la sobrecarga del método **Equals** comprueba el contenido de cada una de las propiedades de los dos tipos anónimos a comparar.

El código del listado 4.11 demuestra esto que acabo de comentar.

```
var ta6 = new { ID = "003", Nombre = "C#" };
var ta7 = new { ID = "003", Nombre = "C#" };

// Equals comprueba los contenidos de las propiedades
var sonIguales = ta6.Equals(ta7);
Console.WriteLine("Usando Equals: {0}", sonIguales);

// El operador == comprueba si las dos instancias son iguales
sonIguales = (ta6 == ta7);
Console.WriteLine("Usando ==: {0}", sonIguales);
```

Listado 4.11. Comprobar el contenido de dos instancias de tipos anónimos

Independientemente de todo lo que el compilador haga para que podamos usar los tipos anónimos, no cabe duda de que en muchas situaciones nos resultarán de utilidad, particularmente cuando los utilicemos con todo lo relacionado con LINQ.

Pero aún nos quedan otros detalles que debemos conocer sobre los tipos anónimos, tanto para sacarles el máximo rendimiento, como para decidir si utilizar este tipo de clases o usar clases definidas por nosotros.

Los tipos anónimos solo los podemos usar a nivel local

Sabiendo que la inferencia de tipos siempre actúa en la definición de los tipos anónimos, es evidente que su uso solo puede ser a nivel local, es decir, dentro de un procedimiento. Por tanto, no podremos usar los tipos anónimos como parámetro de un método, como valor devuelto por una función o propiedad, ni para definir variables a nivel de tipo (accesible a todo el tipo).

Para que nos sirva de recordatorio, los tipos anónimos solo los podremos definir donde podamos inferir automáticamente los tipos de las variables.

Esto también nos sirve para saber que aunque podamos crear *arrays* o colecciones de tipos anónimos, esas colecciones no podremos utilizarlas fuera del procedimiento en el que lo definimos, salvo que lo almacenemos como **object**, pero en ese caso perderemos el tipo de los elementos de esa colección o *array*.

En estos casos, en los que necesitamos usar esos objetos de tipo anónimo fuera del procedimiento en el que los definimos (aunque parezca contradictorio), lo mejor es usar tipos que no sean anónimos.

Tipos anónimos que contienen otros tipos anónimos

Como ya comenté, un tipo anónimo lo podemos usar en cualquier sitio en el que podamos inferir una variable, por tanto, también podemos definir tipos anónimos como valor de una propiedad de un

tipo anónimo. En el listado 4.12 vemos un ejemplo en el que la propiedad *Artículo* es a su vez un tipo anónimo.

```
var ta9 = new
{
    ID = 1,
    Artículo = new { Cantidad = 12 },
    Descripción = "Prueba 9"
};
```

Listado 4.12. Un tipo anónimo puede contener otros tipos anónimos

Y si necesitamos que una de esas propiedades sea un *array*, podemos hacerlo como vemos en el código del listado 4.13 en el que la propiedad *Artículos* es en realidad un *array* de tipos anónimos (el mismo que el usado en la propiedad *Artículo* del listado 4.12).

```
var ta10 = new
{
    ID = 2,
    Artículos =
        new [] {
            new { Cantidad = 6 },
            new { Cantidad = 24 },
            new { Cantidad = 10 }
        },
    Descripción = "Prueba 10"
};
```

Listado 4.13. Las propiedades pueden ser arrays de tipos anónimos

Si quisiéramos que la propiedad *Artículos* fuese una colección en lugar de un *array*, tendríamos que crear un método que convierta ese *array* en una colección o bien usar el método extensor *ToList* en el *array* con los tipos anónimos que representan a cada artículo, tal como vemos en el listado 4.14.

```
var ta11 = new
{
    ID = 3,
    Artículos =
        (new [] {
            new { Cantidad = 6 },
            new { Cantidad = 24 },
            new { Cantidad = 10 }
        }).ToList(),
    Descripción = "Prueba 11"
};
```

Listado 4.14. Si queremos que una propiedad sea una colección en lugar de un array podemos usar el método extensor ToList

Nota

*En el proyecto de prueba de este capítulo he definido un método que crea una colección a partir de un array de parámetros opcionales (**params**).*

Recomendaciones

Para finalizar esta primera aproximación a los tipos anónimos, veamos algunas recomendaciones de cómo y cuándo debemos usar los tipos anónimos.

En realidad, los tipos anónimos no están pensados para usarlos como una forma de sustituir los tipos normales, y casi con toda seguridad la mayor utilidad que le veremos a este tipo de datos es cuando los usemos con LINQ, y como de LINQ nos ocuparemos en un próximo capítulo, cuando le llegue el turno comprobaremos que ahí sí que tienen utilidad.

Mientras llega el capítulo de LINQ, repasemos un poco lo que hemos tratado de esta nueva característica de C#; espero que la siguiente relación le sirva al lector para tener una visión rápida de cómo y cuándo utilizar los tipos anónimos.

- Los tipos anónimos solo los podemos usar a nivel de procedimiento (localmente).
- Siempre los crearemos usando la característica conocida como inicialización de objetos.
- Las propiedades siempre son de solo lectura.
- El compilador creará un nuevo tipo anónimo si no existe uno a nivel de ensamblado que tenga la misma firma.
- En la firma de un tipo anónimo se tendrá en cuenta el número de propiedades y los nombres de éstas (se hace diferencia entre mayúsculas y minúsculas), además del orden en el que estén declaradas esas propiedades. Los tipos inferidos en las propiedades no influyen en la firma del tipo anónimo.
- Al comparar dos instancias de tipos anónimos (si son exactamente del mismo tipo), el método **Equals** comprobará el contenido de cada una de las propiedades de las dos instancias, mientras que el operador de igualdad solo comprobará si las dos instancias son la misma.
- El método **Equals** permite comprar dos instancias de tipos anónimos diferentes, pero siempre devolverá un valor falso.

Versiones

Esta característica solo la podemos usar con .NET Framework 3.5 y no es necesario añadir ninguna importación especial.

Expresiones lambda

Según nos dice la documentación de Visual Studio 2008 (en la guía de programación de C#): *"Una expresión lambda es una función anónima que puede contener expresiones e instrucciones y se puede utilizar para crear delegados o tipos de árboles de expresión"*.

El lector que conozca algo de C# seguramente esta definición le recordará a los métodos anónimos que introdujo ese lenguaje en la versión 2.0, pues algo parecido, pero no exactamente lo mismo.

En C# 3.0 las expresiones *lambda* (o funciones anónimas o funciones en línea) son funciones que pueden tener una instrucción (o un bloque de instrucciones) que será la encargada de devolver el valor de retorno de esa función (en las funciones anónimas no se usa la instrucción **return** para devolver los valores).

En realidad, el contenido de las funciones anónimas se reduce a una sola expresión que devuelve un valor, pero si necesitamos usar más de una expresión podemos incluirlo dentro de un bloque de código encerrado entre un par de llaves, en ese caso tendremos que usar la instrucción **return** para indicar el valor devuelto por la función anónima.

Antes de ver ejemplos de estos dos casos, es conveniente explicar cómo se definen las expresiones *lambda*.

Para definir una expresión *lambda* se usa un nuevo operador llamado operador *lambda* (\Rightarrow), a la izquierda de este operador se indicarán las variables que se usarán en el código (o expresión) que evaluará y a la derecha del operador se indicará la expresión (o bloque de instrucciones) que contendrá esta función en línea.

En C# podemos asignar este tipo de expresiones a una variable que represente a un delegado que tenga la misma firma que la función anónima (tipos de parámetros y valor devuelto), por ejemplo, en el código del listado 4.15 definimos un delegado que posteriormente lo usamos en el listado 4.16 para crear una función anónima que devolverá la cadena **Par** o **Impar** dependiendo del valor recibido como argumento.

```
delegate string EsParCallback(int n);
```

Listado 4.15. Definición de un delegado para usar con una expresión lambda

```
EsParCallback ep = n => n % 2 == 0 ? "Par": "Impar";  
Console.WriteLine("{0} es par: {1}", 3, ep(3));  
Console.WriteLine("{0} es par: {1}", 4, ep(4));
```

Listado 4.16. Ejemplo de expresión lambda de una sola expresión

Si quisiéramos usar más de una instrucción para crear nuestra función anónima, podemos hacerlo de forma parecida al código del listado 4.17 que es equivalente al mostrado en el listado 4.16.

```
EsParCallback ep = n => {  
    string s;  
    s = n % 2 == 0 ? "Par" : "Impar";  
    return s;  
};
```

Listado 4.17. Ejemplo de expresión lambda con un bloque de código

Nota

A lo largo de este texto (y lo que resta del libro) usaré los términos “expresiones lambda”, “funciones anónimas” o “funciones en línea” para referirme a esta característica de C# 3.0, aunque el nombre más acertado (o conveniente) es el de expresión lambda, por la razón de que el operador usado para definir este tipo de expresiones es el “operador lambda” (=>).

Entendiendo a los delegados

Como ya he comentado, las expresiones *lambda* se pueden usar en cualquier lugar en el que podamos usar un delegado, pero esto necesita de un poco de clarificación para comprender mejor esta característica.

Un delegado define la firma que debe tener el método al que queremos invocar desde un objeto del tipo de ese delegado. El caso habitual en el que se usan los delegados es en los eventos, una clase define un evento (que en realidad es una variable del tipo de un delegado) y cuando quiere informar que ese evento se ha producido, hace una llamada a cada uno de los métodos que se han suscrito a ese evento. En C#, para indicar el método que queremos suscribir a un evento, debemos usar el operador += seguido de una llamada al constructor del delegado relacionado con el evento, al que le pasaremos como argumento el nombre del método que usaremos para interceptarlo; aunque desde la versión 2.0 de C# podemos usar una forma directa en la que simplemente indicamos el nombre del método a usar en ese evento. Por ejemplo, si tenemos una aplicación de **Windows.Forms** y queremos interceptar el evento **Click** de un botón llamado *button1*, tendremos que definir un método que tenga la misma firma que el delegado asociado a ese evento (en nuestro ejemplo, el método se llama *button1_Click*), y para asociar ese evento con ese método lo haremos tal como vemos en el listado 4.18, en el cual mostramos las dos formas de asociar el evento con el método, según usemos la forma tradicional o la que se incluyó en la versión 2.0 del lenguaje (*covarianza*).

```
// Asignación del método del evento usando el constructor del delegado  
this.button1.Click += new System.EventHandler(this.button1_Click);  
  
// Asignación del método usando la covarianza  
this.button1.Click += this.button1_Click;
```

Listado 4.18. Dos formas de asignar un método a un evento

Todo esto es para que comprendamos mejor la parte de la definición de las expresiones *lambda* en la que se indica que se pueden usar para crear delegados. De hecho, si el método *button1_Click* define el código mostrado en el listado 4.19, podríamos asociar el evento **Click** del botón tal como vemos en el código del listado 4.20, ya que lo que generamos con esa expresión *lambda* es un delegado, el cual podemos asignar directamente al evento **Click** del botón. En el código del listado 4.20 los tipos de los datos de los parámetros se infieren de forma automática.

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Has pulsado en el botón");
}
```

Listado 4.19. Definición del método a usar en las definiciones del listado 4.17

```
this.button1.Click +=
    (sender, e) => MessageBox.Show("Has pulsado en el botón");
```

Listado 4.20. Las expresiones *lambda* se pueden usar donde esté permitido usar un delegado

Definir una expresión *lambda*

En C# las expresiones *lambda* las tenemos que definir usando el operador `=>` (según la documentación de Visual Studio 2008, este operador se lee como: *va a*). Y tal como comenté al principio de esta sección, antes de ese operador indicaremos los parámetros que se usará en la función anónima. Si el cuerpo de esa función está formado por una sola expresión, no es necesario usar la instrucción **return** para devolver el valor, simplemente se usa el resultado de esa expresión como valor a devolver. Sin embargo, si el cuerpo de la expresión *lambda* está formado por un bloque de código, sí que debemos usar la instrucción **return** seguida del valor que queremos devolver, al menos si esa función devuelve un valor, ya que si el delegado representado es de tipo **void**, evidentemente no devolverá nada, tal como vimos en el código del listado 4.20.

Es importante saber que tanto los tipos de datos de los parámetros como el valor devuelto se infieren, por tanto no es necesario indicarlos de forma expresa, pero sí que deben coincidir con el delegado al que queremos relacionarlo. De hecho, en la mayoría de las ocasiones en las que usamos las expresiones *lambda* (normalmente para todo lo relacionado con LINQ) no tendremos que crear un delegado de forma expresa, ya que este tipo de expresiones las usaremos cuando se espere un delegado como parámetro. Si nuestra intención es usar este tipo de funciones anónimas de forma independiente, por ejemplo para asignarlas a una variable, sí que tendremos que definir el delegado que usaremos para el tipo de variable que recibirá la expresión *lambda* como sustituto de un delegado. El código mostrado en los listados 4.15 a 4.17 demuestra esta forma de usar las expresiones *lambda*.

Si el lector utiliza también Visual Basic, debería saber que ese lenguaje permite la asignación de una expresión *lambda* a una variable sin necesidad de crear un delegado intermedio que la represente. Ese es el caso del código del listado 4.21, que para usarlo en C# tendríamos que definir un delegado que tenga la misma firma que la expresión *lambda* asignada a la variable *delSuma*, tal como vemos en el código del listado 4.22 y para instanciarlo lo haremos tal como se muestra en el listado 4.23.

```
Dim delSuma = Function(n1 As Integer, n2 As Integer) n1 + n2
```

Listado 4.21. Definición de una expresión lambda en Visual Basic

```
delegate int SumaCallback(int n1, int n2);
```

Listado 4.22. Definición del delegado a usar en el listado 4.23

```
SumaCallback delSuma = (n1, n2) => n1 + n2;  
int i = delSuma(3, 5);
```

Listado 4.23. Uso de una expresión lambda para el delegado del listado 4.22

Las expresiones lambda como parámetro de un método

Cuando queremos pasar por argumento a un método otro método, el parámetro que recibirá la dirección de memoria de ese método debe ser un delegado y como las expresiones *lambda* las podemos usar en cualquier sitio en el que se pueda usar un delegado, también podremos usar las funciones anónimas para realizar esta tarea.

Para comprenderlo mejor, veamos primero cómo lo haríamos con delegados pero al estilo de la versión anterior de C#, de esta forma tendremos más claro todo lo relacionado a las expresiones *lambda* y la facilidad que tenemos ahora para hacer ciertas tareas que antes suponía tener que escribir más código y hacer más pasos intermedios.

El ejemplo consiste en definir un método que reciba dos valores de tipo entero y un tercer parámetro que será un delegado. Éste lo definimos como una función que recibe dos parámetros de tipo entero y devuelve otro entero. Para llamar a este método, tendremos que indicar dos números y la dirección de memoria de una función con la misma firma del delegado; esa función simplemente sumará los dos valores.

En los siguientes listados tenemos el código que hace todo esto que acabo de comentar, el delegado usado es el mostrado anteriormente en el listado 4.22.

```
static int sumar(int x, int y)
{
    return x + y;
}
```

Listado 4.24. La función con la misma firma que el delegado

```
static void conDelegado(int x, int y, SumaCallback d)
{
    Console.WriteLine("x = {0}, y = {1}", x, y);
    Console.WriteLine("d(x,y) = {0}", d(x, y));
}
```

Listado 4.25. Un método que recibe un delegado como parámetro

```
conDelegado(10, 20, sumar);
```

Listado 4.26. Una de las formas de usar el método del listado 4.25

Para usar el método del listado 4.25 tenemos que pasarle dos valores de tipo entero y la dirección de memoria del método que se usará cuando se llame al delegado. La forma más sencilla de hacerlo es tal como vemos en el listado 4.26, pero también podríamos crear una variable del tipo del delegado y usar esa variable como argumento del tercer parámetro, tal como vemos en el listado 4.27.

```
SumaCallback sumaDel = sumar;
conDelegado(10, 20, sumaDel);
```

Listado 4.27. Otra forma de utilizar el método del listado 4.25

El método *conDelegado* espera recibir un delegado en el tercer parámetro, y como sabemos, las expresiones *lambda*... ¡efectivamente! *las podemos usar en cualquier sitio en el que se pueda usar un delegado*, por tanto, el código del listado 4.28 también sería válido.

```
conDelegado(10, 20, (n1, n2) => n1 + n2);
```

Listado 4.28. El método del listado 4.25 lo podemos usar con una expresión lambda

La ventaja de usar el código del listado 4.28 es que nos libra de tener que definir una función que tenga la firma del delegado, ya que al usar la función anónima obtenemos el mismo resultado.

Otra ventaja es que podríamos querer usar ese mismo método con otra función que, en lugar de sumar los parámetros hiciera otra operación, con las expresiones *lambda* sería una tarea fácil, ya que solo tendríamos que cambiar la expresión que devuelve el valor, como en el listado 4.29 en el que efectuamos una resta.

```
conDelegado(10, 20, (n1, n2) => n1 - n2);
```

Listado 4.29. Al ser una función en línea, podemos cambiar el resultado a devolver

Si esto mismo quisiéramos hacerlo con versiones anteriores de C# nos veríamos obligados a definir otra función que hiciera esa operación de restar los dos valores y llamar a este método pasándole la dirección de memoria de esa función.

Delegados genéricos

Tal como hemos visto hasta ahora, cada vez que queramos usar una expresión *lambda* debe existir un delegado adecuado a dicha expresión. Para solventar en parte esta obligación de que exista un delegado con la firma adecuada antes de usar una expresión *lambda* (al menos cuando la queremos usar para asignarla a una variable), en .NET Framework 3.5 existen una serie de delegados *generic* que podemos usar en diferentes situaciones. En el listado 4.30 definimos una variable del tipo de uno de estos delegados *generic*, le asignamos una expresión *lambda* y obtendremos el mismo resultado que ya vimos en el listado 4.23.

```
Func<int, int, int> de1Suma2 = (n1, n2) => n1 + n2;  
int j = de1Suma2(3, 5);
```

Listado 4.30. .NET Framework define una serie de delegados *generic* que podemos usar con las expresiones *lambda*

En particular existen dos tipos de delegados *generic* que podemos usar para estos casos, según devuelvan o no un valor. De estos delegados existen varias sobrecargas, según el número de parámetros que acepten. Los delegados que devuelven valores tienen el nombre **Func**, mientras que los que no devuelven nada (el tipo de valor devuelto es **void**) se llaman **Action**.

Debido a que ya existen estos tipos de delegados *generic*, podríamos cambiar el código de la definición del método *conDelegado* mostrado en el listado 4.25 para que en lugar de definir el tercer parámetro del tipo *SumaCallback* (que nos permite pasar la dirección de memoria de una función al estilo de la definida en el listado 4.24) podemos hacer que sea del tipo de una de las sobrecargas del delegado **Func**. En el listado 4.31 vemos la definición de ese nuevo método y en el listado 4.32 vemos cómo usarlo, en este caso para pasarle una expresión *lambda* que realiza una multiplicación entre los dos valores recibidos.

```
static void conDelegado2(int x, int y, Func<int, int, int> d)
{
    Console.WriteLine("x = {0}, y = {1}", x, y);
    Console.WriteLine("d(x,y) = {0}", d(x, y));
}
```

Listado 4.31. Nueva definición del método que recibe un delegado en el tercer parámetro

```
conDelegado2(50, 20, (x, y) => x * y);
```

Listado 4.32. Llamada al método que define el delegado Func

.NET Framework 3.5 define 5 sobrecargas del delegado *generic* **Func** según el número de parámetros que se utilicen (usando siempre el último para indicar el valor devuelto), si necesitamos un delegado que no devuelva nada, podemos usar el delegado *generic* **Action**, que al igual que **Func** tiene 5 sobrecargas según los parámetros que reciba.

Todos estos delegados, al definir los parámetros de tipo *generic*, se pueden usar para cualquier tipo de datos, en el ejemplo del listado 4.31, los tres parámetros usados son de tipo **int**.

Debido a que en C# las expresiones *lambda* siempre se deben usar si existe un delegado adecuado a los parámetros y el tipo de datos que devuelve, no es necesario indicar de qué tipo de datos son esos parámetros, pero si lo hacemos, debemos usar los tipos adecuados. En el listado 4.33 vemos cómo usar el mismo código del listado 4.32, pero indicando de qué tipos son esos parámetros.

```
conDelegado2(50, 20, (int x, int y) => x * y);
```

Listado 4.33. Podemos indicar de forma explícita los tipos de los parámetros de las expresiones lambda

Si nos decidimos por indicar los tipos de datos de los parámetros de las expresiones *lambda*, debemos indicarlos todos, por tanto el código del listado 4.34 dará error ya que hay una inconsistencia en los parámetros de la expresión *lambda*.

```
conDelegado2(50, 20, (int x, y) => x * y);
```

Listado 4.34. Si indicamos el tipo de un parámetro, debemos indicarlo todos

Ámbito en las expresiones lambda

Viendo el código de los ejemplos mostrados hasta ahora es posible que nos llevemos la impresión de que en la expresión de una función anónima siempre usaremos las variables indicadas en los parámetros.

Pero eso no es así, en esa expresión podemos usar cualquier constante o variable que esté en el mismo ámbito de la función anónima, es decir, además de los parámetros, también podemos usar las variables que estén definidas en el mismo procedimiento en el que estemos usando la expresión *lambda* o cualquier otra variable definida en cualquier otra parte, pero siempre que esté accesible, por ejemplo, las variables definidas a nivel de tipo.

En esto no cambia con respecto a las funciones normales, la diferencia es que las expresiones *lambda* siempre las usaremos desde “dentro” de un procedimiento, por tanto, todas las variables locales de ese procedimiento también las podremos utilizar para evaluar la expresión que devolverá.

En el listado 4.35 vemos un ejemplo en el usamos variables definidas fuera de la expresión *lambda*.

```
int i1 = 10;
double d1 = 22.5;

Func<int, double> d = n => (n * i1) + d1;

Console.WriteLine(d(12));
```

Listado 4.35. En las expresiones lambda podemos usar variables que estén en ámbito

Si usamos variables externas dentro de la expresión de una función anónima, y esas variables tienen una vida más corta que la propia función, no habrá problemas de que el recolector de basura las elimine cuando pierdan el ámbito, ya que al estar siendo usadas, la vida de las mismas se mantiene.

Por ejemplo, en el listado 4.36 tenemos la definición de un método que devuelve un delegado como el que hemos estado usando en los listados anteriores. El valor devuelto por ese método es una expresión *lambda*, que utiliza una variable local (*k*) para efectuar el cálculo que debe devolver cuando se utilice.

```
static Func<int, int, int> conDelegado3(int x, int y)
{
    var k = (x + y) / 2;
    Console.WriteLine("k = {0}", k);

    return (n1, n2) => n1 * k + n2 / k;
}
```

Listado 4.36. Un método que devuelve una expresión lambda

En el código del listado 4.37 usamos ese método, y guardamos una referencia a la función anónima creada dentro del método, después usamos en varias ocasiones esa variable, y como podemos imaginar, cada vez que utilizamos esa variable estamos invocando a la función anónima, que aún sigue estando activa y conservando el valor que tenía la variable *k*. Ese valor no se perderá aunque el método (*conDelegado3*) ya haya terminado y, por tanto, todo el contenido del mismo se haya desechado. Si nos sirve de comparación, esto es similar a lo que ocurre cuando creamos un formulario dentro de un método y aunque el método finalice, el formulario sigue estando operativo hasta que finalmente lo cerremos.

```
var d2 = conDelegado3(7, 4);  
  
Console.WriteLine("Llamadas a d2");  
var s = d2(3, 3);  
Console.WriteLine(s);  
  
s = d2(5, 7);  
Console.WriteLine(s);
```

Listado 4.37. Mientras el delegado devuelto por la función siga activo, la variable local usada para efectuar los cálculos estará “viva”

Otro ejemplo parecido es si usamos una expresión *lambda* para asignarla a un manejador de eventos, es decir, usar una función anónima en lugar de un método, tal como vimos en el listado 4.20.

¿Cómo clasificar una colección de tipos anónimos?

Como sabemos, para poder clasificar los elementos de una colección, los objetos almacenados en la colección deben implementar la interfaz **IComparable**, y las clases anónimas no implementan esa interfaz, además de que aunque pudiéramos implementarla, tampoco podríamos escribir un método para que realice las comparaciones, ya que los tipos anónimos no nos permiten definir métodos.

Pero todos los métodos de clasificación permiten que indiquemos una clase basada en **IComparer** que sea la que se encargue de realizar las comparaciones de los elementos que no implementan directamente la interfaz que define el método **CompareTo**. El problema es que esa clase necesita conocer los tipos de datos con los que debe trabajar, por tanto, no sería la solución, ya que, como vimos anteriormente, no tenemos forma de acceder al tipo interno que el compilador usa para los tipos anónimos.

Una de las sobrecargas del método **Sort** de la clase genérica **List<T>** permite que indiquemos un método que sea el encargado de comparar dos elementos de la colección y devolver el valor adecuado a la comparación. Nuevamente nos encontramos con el problema de que, para que ese método funcione, la comparación se debe hacer con dos elementos del tipo que queremos clasificar, y si esos tipos son anónimos ¿cómo los indicamos? Y aquí es donde entran en escena las extensiones de .NET Framework 3.5, particularmente las funciones anónimas y la contravarianza (usar un tipo en un parámetro que se derive del tipo indicado en el delegado), ya que ese método que espera **Sort**, lo podemos indicar como una expresión *lambda*, de forma que reciba dos objetos de los contenidos en la colección y hagamos la comparación que creamos conveniente. Mejor lo vemos con un ejemplo.

En el listado 4.38 tenemos una adaptación del código mostrado en el listado 4.14, en el que creamos un tipo anónimo, que a su vez contiene una colección de tipos anónimos, y para clasificar esa colección usaremos el código del listado 4.39 en el que usamos una función en línea para realizar la comparación de dos elementos. En este ejemplo tenemos dos propiedades en los elementos “anónimos” de la colección, pero usaremos solo una de esas propiedades para realizar la clasificación.

```
var ta13 = new
{
    ID = 17,
    Artículos =
        (new[] {
            new { ID = 95, Cantidad = 6 },
            new { ID = 22, Cantidad = 24 },
            new { ID = 95, Cantidad = 1 },
            new { ID = 39, Cantidad = 10 }
        }).ToList(),
    Descripción = "Prueba 13"
};
```

Listado 4.38. Un tipo anónimo que tiene una colección de tipos anónimos

```
ta13.Artículos.Sort((x, y) => x.ID.CompareTo(y.ID));
```

Listado 4.39. Usamos una expresión lambda como función a usar con el método Sort

En el código del listado 4.39, la sobrecarga que usamos del método **Sort** es la que utiliza el delegado **Comparison<T>**, éste recibe dos parámetros, los compara y devuelve un valor cero si son iguales, menor de cero si el primer argumento es menor que el segundo, o mayor de cero si el segundo es mayor (es decir, los valores típicos para las comparaciones). En nuestro ejemplo, el delegado (el método que hará las comparaciones), es una función *lambda* que recibe dos objetos, y gracias a la inferencia automática de tipos no es necesario indicar el tipo de los mismos, ya que el compilador sabe que son del tipo anónimo que hemos almacenado en esa colección, por eso nos permite acceder a las propiedades de esos parámetros, que en este ejemplo hemos usado el método **CompareTo** de la propiedad **ID**. Ese método anónimo se usará cada vez que el *runtime* de .NET necesite comparar dos elementos de la colección, es decir, clasificará el contenido de la colección **Artículos** por el valor de la propiedad **ID**.

En el listado 4.38, cada uno de los elementos de la colección **Artículos** es de tipo anónimo con dos propiedades en lugar de una, que es como lo teníamos en el listado 4.14, pero esto solo lo he hecho para usar un código diferente, no porque sea necesario que el tipo de datos a comparar tenga una propiedad llamada **ID**.

En el código del listado 4.38 **Artículos** es una colección, si nos decidimos por dejarla como un *array*, tendríamos que cambiar la forma de clasificar esos elementos. En los listados 4.40 y 4.41 se muestra cómo crear el tipo anónimo con una propiedad que es un *array* de elementos anónimos y la forma de llamar a la sobrecarga adecuada del método estático **Sort** definido en la clase **Array**.

```
var ta14 = new
{
    ID = 18,
    Artículos =
        new[] {
```



```
        new { ID = 95, Cantidad = 6 },  
        new { ID = 22, Cantidad = 24 },  
        new { ID = 95, Cantidad = 1 },  
        new { ID = 39, Cantidad = 10 }  
    },  
    Description = "Prueba 14"  
};
```

Listado 4.40. Un tipo anónimo con una propiedad que es un array de tipos anónimos

```
Array.Sort(ta14.Artículos, (x, y) => x.ID.CompareTo(y.ID));
```

Listado 4.41. Uso de una sobrecarga del método Sort de la clase Array con una expresión lambda

En la comparación que hacemos en el cuerpo de la función anónima podemos comparar lo que queremos, no solo los valores solitarios de las propiedades de esa clase anónima. Por ejemplo, en el listado 4.42 tenemos una comparación algo más complicada, esto es posible porque esa función debe devolver un valor entero indicando el orden (0, 0 < o > 0) y la forma de conseguir ese valor no le preocupa al CLR de .NET.

```
ta13.Artículos.Sort((x, y) =>  
    (x.ID.ToString("00") + x.Cantidad.ToString("000"))  
    .CompareTo  
    (y.ID.ToString("00") + y.Cantidad.ToString("000"))  
);
```

Listado 4.42. Modificación de la función anónima para clasificar los elementos usando las dos propiedades del tipo anónimo

Como sabemos, en la expresión usada en el cuerpo de la función anónima, podemos usar cualquier código que sea válido en una expresión y siempre que devuelva un valor del tipo esperado (**int** en nuestro ejemplo), por tanto, podríamos crear una función con nombre que sea la encargada de hacer las comprobaciones necesarias para clasificar esos dos elementos. En el listado 4.43 tenemos el equivalente al listado 4.42, pero usando una función externa, que es la definida en el listado 4.44.

```
ta13.Artículos.Sort((x, y) =>  
    comparaEnteros(x.ID, x.Cantidad, y.ID, y.Cantidad));
```

Listado 4.43. La expresión usada en la función anónima usa una función externa

```
static int comparaEnteros(int x1, int x2, int y1, int y2)  
{  
    string x = x1.ToString("00") + x2.ToString("000");  
    string y = y1.ToString("00") + y2.ToString("000");  
    return x.CompareTo(y);  
}
```

Listado 4.44. La función usada por la expresión lambda del listado 4.43

En los listados 4.42 y 4.44 realizamos una comparación de forma que se clasifiquen los elementos por el valor de la propiedad **ID** teniendo en cuenta el valor de la cantidad, de forma que si hay dos elementos con el mismo **ID** se muestren clasificados por la cantidad además del identificador. Usando los elementos del listado 4.40, y mostrándolos como vemos en el listado 4.45, el resultado sería el mostrado en la figura 4.2.

```
Console.WriteLine("ID = {0}, Descripcion = {1}",  
    ta13.ID, ta13.Descripcion);  
  
foreach(var a in ta13.Artículos)  
{  
    Console.WriteLine(" {0}", a);  
}
```

Listado 4.45. Mostrar el contenido del tipo anónimo y de la colección Artículos

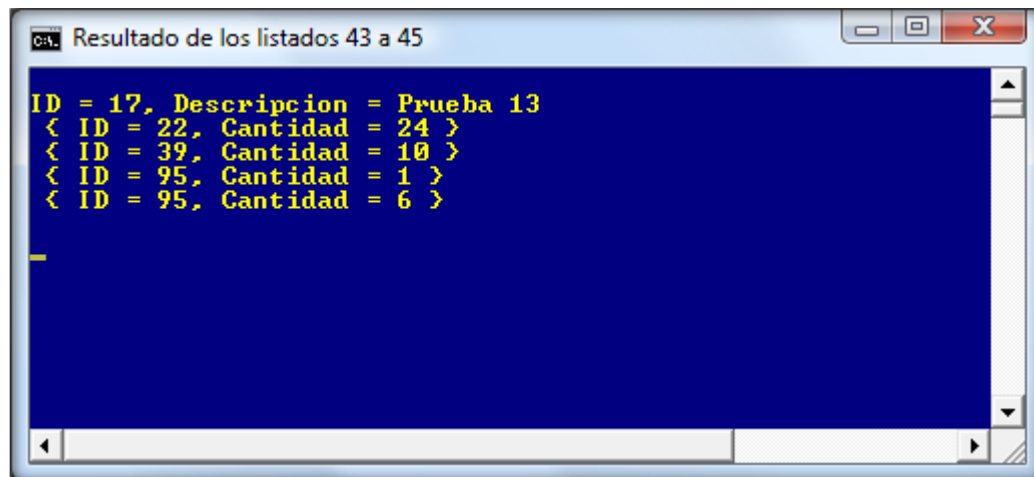


Figura 4.2. Resultado de mostrar los datos usando el listado 4.45

Para realizar todas estas comparaciones, lo ideal es que tuviésemos acceso al tipo anónimo usado como elemento de la colección **Artículos** y de esa forma poder pasar esos dos elementos a una función que se encargue de hacer las comparaciones oportunas, pero accediendo a las propiedades directamente. Esto no es posible, ya que no podemos acceder a los tipos anónimos; si pudiéramos, dejarían de ser anónimos.

Me va a permitir el lector que mire a otro lado, porque no quiero que se me relacione con lo que voy a decir a continuación.

Como sabemos, en C# no podemos hacer lo que se conoce como *late binding* o compilación tardía, es decir, acceder a miembros de un tipo sin saber si ese tipo los define o no. En la próxima versión de C# esto será posible gracias a la ejecución dinámica, pero actualmente (y por fortuna) C# no permite este tipo de “barbaridades”. Sin embargo, Visual Basic sí que permite hacer estas cosas, al

menos si tenemos desactivada la comprobación estricta del código (**Option Strict Off**), por tanto, podríamos crear un ensamblado en Visual Basic en el que definamos un método que reciba dos objetos por parámetro y acceder a las propiedades que “supuestamente” define esos objetos. En el código del listado 4.46 vemos la definición completa de ese código de Visual Basic, el cual tendremos que compilar y agregar una referencia a nuestro proyecto de C# para poder acceder a esa clase y al método estático (o compartido que es como se conocen a estos métodos en Visual Basic) que usaremos en nuestra expresión *lambda* para realizar la clasificación de los elementos, tal como podemos ver en el listado 4.47.

```

Option Strict Off
Option Infer On

Imports System

Public Class Class1
    Public Shared Function ComparaArticulos(ByVal x As Object, _
                                           ByVal y As Object) As Integer

        Dim a1 = x.ID.ToString("00") & x.Cantidad.ToString("000")
        Dim a2 = y.ID.ToString("00") & y.Cantidad.ToString("000")

        Return a1.CompareTo(a2)
    End Function
End Class

```

Listado 4.46. Una función de VB que utiliza late binding para acceder a las propiedades de dos objetos

```

ta13.Articulos.Sort((x, y) => Class1.ComparaArticulos(x, y));

```

Listado 4.47. Clasificamos los elementos de la colección Artículos pasándole dos objetos del tipo anónimo a una función definida en un ensamblado de Visual Basic

Ni que decir tiene que si ese tipo anónimo no define esas dos propiedades a las que accedemos en la función del listado 4.46 ese código fallará, ya que el compilador no comprueba (entre otras cosas, porque no puede) si esos objetos definen esas dos propiedades que usamos, por tanto, hasta que no se ejecute ese código no se sabrá si es correcto o no. En nuestro ejemplo, ese código funcionará, pero ya sabemos que si el tipo de datos que pasamos como argumento de esa función no definiera esas dos propiedades, recibiríamos una excepción en tiempo de ejecución, echando al traste la aplicación. Estos son los riesgos de la compilación tardía (o dinámica), que no hay forma de asegurarnos de que el código siempre funcionará.

Moraleja: Si necesitamos hacer cosas que dependan de ciertas propiedades de los elementos de una colección o cualquier otro tipo de datos, es preferible usar tipos “normales” y dejar los anónimos para cuando no tengamos otra posibilidad.

Consideraciones para las expresiones lambda

Para finalizar este tema de las expresiones *lambda* (al menos hasta que veamos todo lo referente a LINQ, ya que este tipo de funciones se utilizan bastante con esa tecnología), repasemos un poco la forma de definir y usar este tipo de funciones anónimas.

En la siguiente lista vemos una serie de condiciones o temas que debemos tener en cuenta a la hora de definir una expresión *lambda*.

- Las expresiones *lambda* se definen usando el operador *lambda* =>.
- Para devolver el valor no se utiliza la instrucción **return**, (salvo que el contenido de la expresión *lambda* sea un bloque encerrado entre llaves). El valor devuelto es la expresión que indicamos después del operador =>.
- Solo podemos usar una expresión como valor a devolver, aunque esa expresión puede contener otras funciones anónimas anidadas e incluso usar funciones y variables externas a las indicadas como parámetro.
- Las expresiones *lambda* se pueden usar con parámetros, los tipos de esos parámetros siempre se infieren, ya que siempre habrá algún delegado asociado con el uso de la expresión *lambda*.
- Si se indica el tipo de uno de los parámetros, debemos indicarlos todos, independientemente de que se pueda inferir el tipo.
- Los parámetros de la expresión *lambda* no pueden ser opcionales, por tanto, no podemos usar **params**.
- Los parámetros no pueden ser de tipo **ref** u **out**.
- No podemos indicar parámetros *generic*.
- Los nombres de los parámetros no pueden ser los mismos que los de otros elementos que estén en el mismo ámbito.
- Los parámetros de una expresión *lambda* son solo visibles dentro de esa función.
- Podemos usar una expresión *lambda* como valor devuelto por una función con nombre, pero el tipo del valor devuelto por ésta debe ser del tipo de un delegado.
- Las funciones anónimas las podemos usar para cualquier tipo de delegado, ya sean definidos por nosotros o los que el propio .NET define.

En los próximos capítulos veremos otros usos de este tipo de funciones, particularmente en otros contextos, como el que nos permiten las expresiones de consulta (LINQ).

Versiones

Esta característica solo la podemos usar con .NET Framework 3.5 y no es necesario añadir ninguna importación especial.

(Esta página se ha dejado en blanco de forma intencionada)

Capítulo 5

Características generales del lenguaje (IV)

Introducción

En este capítulo continuamos con las novedades de C#, que solo podemos usar si nos apoyamos en los ensamblados que acompañan a .NET Framework 3.5. En esta ocasión le toca el turno a una nueva característica que, al igual que la inferencia automática de tipos, ha levantado un poco de polémica, sobre todo porque desvirtúa todo lo referente a la programación orientada a objetos y puede ser causante de “malos hábitos”, pero que por otro lado, su aparición es casi necesaria para poder utilizar todas las extensiones introducidas en el *framework* para dar soporte a LINQ. Pero mejor que sea el lector el que decida si ésta u otras de las novedades que incorpora la versión 3.5 de .NET Framework son “convenientes” o no.

Métodos extensores

Los métodos extensores (o métodos de extensión) son una nueva propuesta de .NET Framework 3.5 para ampliar la funcionalidad de las clases (o tipos) existentes sin necesidad de tener acceso al código fuente de las mismas.

Por medio de los métodos extensores podemos agregar nuevas funciones (o métodos) a cualquier clase (o tipo), de forma que “extendemos” su funcionalidad.

Nota

Solo podemos definir métodos extensores pero no podemos crear extensiones en forma de propiedades o cualquier otro elemento que no sea un método de instancia.

Y como veremos más adelante, podemos crear métodos extensores que sean sobrecargas de métodos existentes, pero no podremos crear una sobrecarga de una propiedad.

No confundamos la extensibilidad conseguida por medio de esta característica con la conseguida por medio de la herencia o la implementación de interfaces, ya que no es lo mismo, al menos en el sentido de que no modificamos la clase que queremos extender ni la usamos como base de una nueva clase, simplemente agregamos nueva funcionalidad a las clases existentes.

Para extender el funcionamiento de una clase (o tipo) existente necesitamos crear nuestro proyecto para que utilice los ensamblados de .NET Framework 3.5 (en particular **System.Core.dll**). Además, la definición de métodos extensores también deben cumplir otros requisitos, entre los más importantes y sin los cuales no podríamos “extender” nada, es que el método extensor lo definamos en una clase estática (por tanto, el método también será estático), que dicho método sea accesible desde otras partes de nuestro código (es decir, que no sea privado) y que en el primer parámetro de ese método indiquemos el tipo de datos que vamos a extender precedido de la palabra clave **this**.

El hecho de declarar el método extensor en una clase estática es por la necesidad de que el método extensor siempre esté disponible, es decir, que no dependa de ninguna instancia de ninguna clase, y como sabemos, cualquier cosa que definamos en una clase estática siempre estará accesible, ya que los métodos definidos en ese tipo de clase también deben ser estáticos (o compartidos).

Pero lo más importante es que definamos el primer parámetro del método usando la palabra clave **this**. Tan importante es la forma de definir ese primer parámetro del método extensor, que si no lo hacemos bien, el resto da igual que los tengamos en cuenta. Pero si ese método no lo definimos en una clase estática, tampoco servirá de nada. De hecho, si definimos un método estático en una clase no estática y en el que usamos la palabra clave **this** en el primero parámetro, recibiremos un mensaje de error al compilar, indicando que los métodos extensores debemos crearlos en una clase estática no genérica. Y si ese método no tiene un ámbito diferente a privado no lo podremos usar desde fuera de la propia clase en el que se define.

Dejemos las cavilaciones y vayamos al grano. Empecemos viendo un ejemplo de un método extensor. En este caso, vamos a definir un método que nos devuelva la longitud de una cadena. El código que define este método extensor es el mostrado en el listado 5.1.

```
static class Extensiones
{
    public static int Len(this string str)
    {
        if(string.IsNullOrEmpty(str))
            return 0;

        return str.Length;
    }
}
```

Listado 5.1. Un método extensor para el tipo string

El primer parámetro del método extensor **Len** es del tipo de datos que queremos extender. Este primer parámetro representa al objeto al que queremos aplicar la extensión, y la forma de usarlo es como vemos en el listado 5.2.

```
string s1 = "Hola extensiones";
int i1 = s1.Len();
```

Listado 5.2. Uso del método extensor definido en el listado 5.1

Cuando el compilador de C# ve que usamos el método *Len* aplicado a una cadena (en este ejemplo es a una variable, pero también lo podemos usar con una constante) busca ese método entre las extensiones que hemos definido y sustituye el primer parámetro por la cadena a la que lo aplicamos, y el valor que devuelve es el indicado por el código de ese método, que en este caso es la longitud de la cadena en cuestión.

El método extensor lo podemos aplicar a cualquier cadena, incluso a cadenas que tengan un valor nulo (**null**). Si éste es el caso (que lo apliquemos a una cadena sin contenido válido), esta función no fallará (como le ocurre a la propiedad **Length**), ya que en el código de la función tenemos una comprobación que tiene en cuenta ese detalle, por tanto, el código mostrado en el listado 5.3 funcionará sin problemas (sin producir una excepción). En realidad, lo que hace este método extensor es casi lo mismo que hace la función homónima de Visual Basic, de forma que siempre podremos usarla con cualquier cadena, sin importarnos si contiene un valor nulo o no.

```
string s2 = null;  
int i2 = s2.Len();
```

Listado 5.3. Los métodos extensores los podemos aplicar incluso a objetos con valores nulos

Del código del listado 5.3, destacar que debido a que queremos asignar un valor nulo a la variable *s2*, tenemos que definirla indicando que es del tipo **string**, ya que si quisiéramos que la inferencia de tipos sea la que asigne el tipo de datos de esa variable, el tipo sería **object**, y como nuestro método extensor solo es aplicable al tipo **string**, no podríamos usar ese método en un objeto de tipo diferente para el que lo hemos definido.

Los métodos extensores e IntelliSense

Debido a que los métodos extensores pueden estar definidos por nosotros o por otros programadores, cuando escribimos el código en el IDE de Visual Studio, éste tiene en cuenta que esas extensiones pueden estar o no disponibles en un tipo, es decir, los métodos extensores no forman parte del tipo, en el sentido de que siempre están presentes, por tanto, al mostrarlos por medio de IntelliSense lo hace de una forma especial. Tal como vemos en la figura 5.1, por un lado usa un icono diferente (con una flechita azul) y por otro, en la descripción emergente (*tooltip*) muestra que es una extensión (esto es aplicable tanto a los métodos extensores definidos con C# como a los definidos en otros lenguajes o en el propio .NET).

Desmitificando los métodos extensores

No es que los métodos extensores sean mitos, pero toda la funcionalidad y la “magia” que tienen la pierden un poco si vemos cómo se compilan en el ensamblado final.

Aunque esa magia y extensibilidad que nos ofrece siempre están presentes, sobre todo si utilizamos el IDE de Visual Studio 2008 para escribir nuestro código, ya que, como hemos visto en el apartado anterior, el propio entorno de desarrollo por medio de IntelliSense nos muestra cuales son los méto-

dos extensores que podemos usar en cada momento, además de permitirnos usarlos de una forma, digamos, más contextual, ya que de cada tipo de datos que usemos nos mostrará los métodos extensores que hay disponibles.

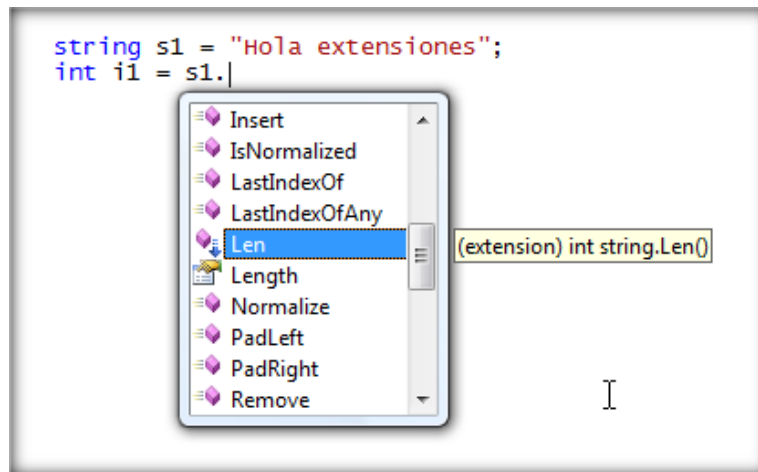


Figura 5.1. IntelliSense y los métodos extensores

Pero en el fondo, los métodos extensores solo son eso: métodos. Un poco especiales, sí, pero métodos normales, al menos si usamos C# para definirlos, ya que excepto por la salvedad de que deben usar la palabra clave **this** delante del primer parámetro y estar definidos en una clase estática, son métodos como el resto, (en Visual Basic los métodos extensores no utilizan una palabra clave para definir el primero parámetro, pero deben tener aplicado el atributo **Extension**, además de estar definidos en un **Module**).

En el listado 5.4 tenemos la definición de un método como el mostrado en el listado 5.1, la diferencia de éste con aquél es que el último no extiende nada, simplemente es un método definido en el mismo archivo en el que está todo el código de prueba (en estos ejemplos estoy usando un proyecto de tipo consola en el que siempre hay un tipo con el método **Main**). Sí, resulta que es idéntico al método extensor del listado 5.1, pero para usarlo lo haremos como con cualquier otro método, es decir, si queremos saber la longitud de una cadena, ésta la tendremos que indicar como argumento al llamar a este método, tal como vemos en el código del listado 5.5.

```
static class Extensiones2
{
    public static int Len(string str)
    {
        if(string.IsNullOrEmpty(str))
            return 0;

        return str.Length;
    }
}
```

Listado 5.4. Un método normal, que es idéntico al método extensor del listado 5.1

```
i1 = Extensiones2.Len(s1);
```

Listado 5.5. Uso habitual de un método

La forma habitual de usar el método *Len* es como vemos en el listado 5.5. De hecho, incluso el método extensor lo podemos usar de esa misma forma, es decir, usando el nombre del tipo en el que está definido, tal como vemos en el listado 5.6.

```
i1 = Extensiones.Len(s1);
```

Listado 5.6. El método extensor también lo podemos usar como un método normal

Todo esto lo estamos viendo simplemente para que sepamos que los métodos extensores son métodos normales, pero con un “toque” especial, el que le da el uso de la palabra clave *this* delante del primer parámetro, que cuando los usamos desde el entorno integrado resulta más evidente esa especialización (porque IntelliSense lo reconoce, etc.). Por supuesto, también podemos usar los métodos extensores en cualquier archivo de código, incluso aunque no los escribamos con el IDE de Visual C# 2008, ya que es el compilador de C# el que reconoce que es un método extensor y sabe qué tiene que hacer con él.

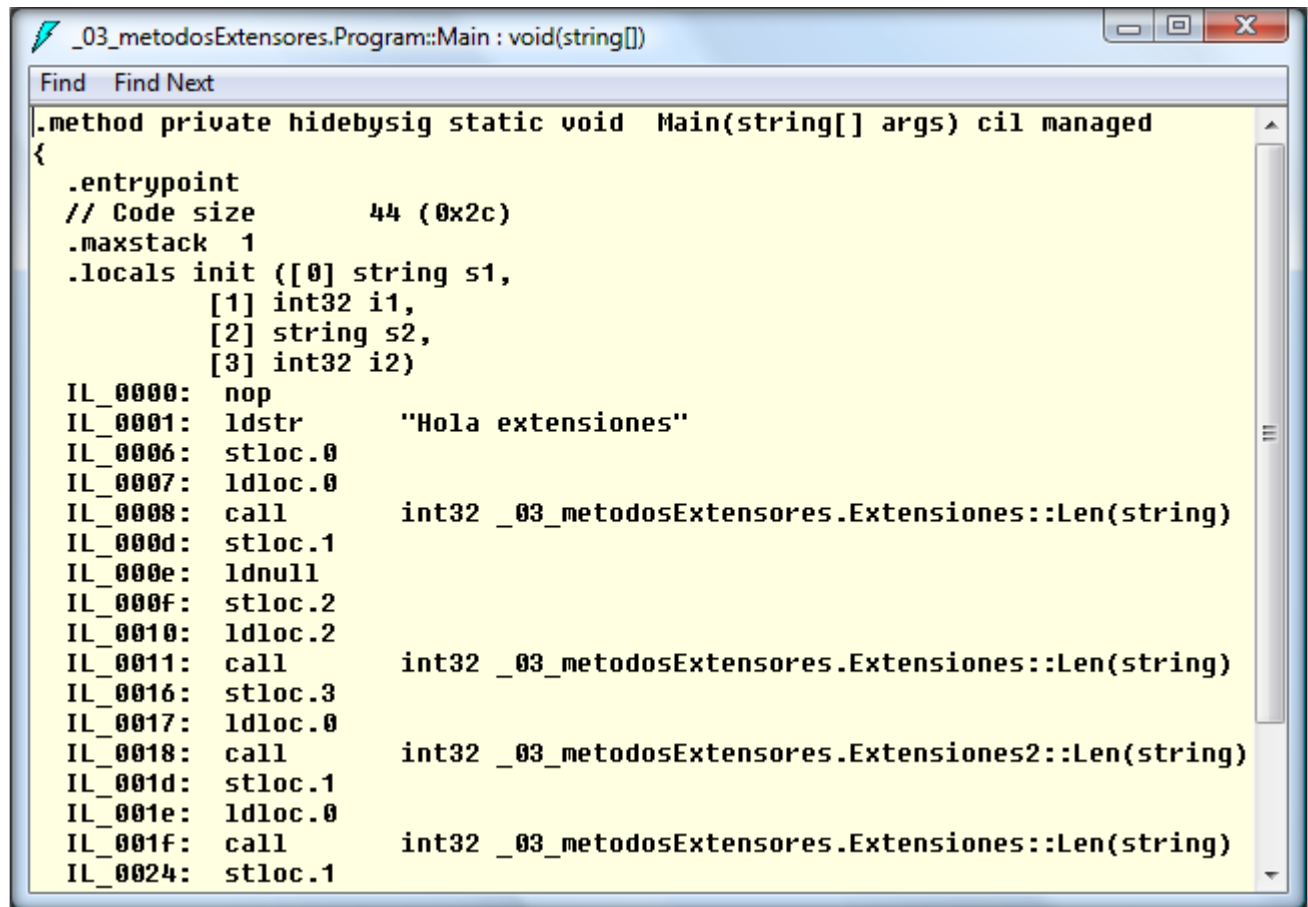
Y precisamente lo que hace el compilador cuando se encuentra con un método extensor es a lo que me refería al decir que los “desmitifica”.

Como ya indiqué al principio del libro, la versión de .NET 3.5 en realidad utiliza el *runtime* de .NET Framework 2.0 (el CLR 2.0), pero define otros ensamblados que le dan nueva funcionalidad, es decir, todo lo que los compiladores generen debe ser “reconocible” por el motor en tiempo de ejecución de la versión 2.0 de .NET. Por tanto, y para hacer las cosas de la forma correcta, el compilador de C# debe generar código compatible, y cuando se encuentra con un método extensor, simplemente lo usa como un método normal y corriente.

Si miramos el código IL generado por el código de estos ejemplos, veremos que la llamada al método extensor del listado 5.2 (o del listado 5.3) no se diferencia en nada de los mostrados en los listados 5.5 y 5.6.

En la figura 5.2 vemos parte del código IL generado por el compilador de Visual C# 2008, y podemos comprobar que la forma de llamar al método *Len* es la misma, independientemente de que lo usemos como extensión o como un método normal.

Para aquellos que no comprendan el código mostrado en la figura 5.2, indicar que el código mostrado en la línea *IL_0008* corresponde a la llamada del método extensor del listado 5.2, la línea *IL_0018* es el listado 5.5 y la línea *IL_001f* corresponde al listado 5.6. Como vemos, en el código intermedio (IL) siempre se usan los métodos de la forma tradicional.



```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size          44 (0x2c)
    .maxstack 1
    .locals init ([0] string s1,
                  [1] int32 i1,
                  [2] string s2,
                  [3] int32 i2)
    IL_0000: nop
    IL_0001: ldstr      "Hola extensiones"
    IL_0006: stloc.0
    IL_0007: ldloc.0
    IL_0008: call       int32 _03_metodosExtensores.Extensiones::Len(string)
    IL_000d: stloc.1
    IL_000e: ldnull
    IL_000f: stloc.2
    IL_0010: ldloc.2
    IL_0011: call       int32 _03_metodosExtensores.Extensiones::Len(string)
    IL_0016: stloc.3
    IL_0017: ldloc.0
    IL_0018: call       int32 _03_metodosExtensores.Extensiones2::Len(string)
    IL_001d: stloc.1
    IL_001e: ldloc.0
    IL_001f: call       int32 _03_metodosExtensores.Extensiones::Len(string)
    IL_0024: stloc.1
}
```

Figura 5.2. Los métodos extensores son métodos normales

El ámbito de los métodos extensores

En C#, los métodos extensores se definen en una clase estática y por tanto deben ser estáticos, es decir, para usarlos no hay que crear ninguna instancia de una clase. Esto es algo que ya sabemos, pero al estar definidos en clases estáticas, todos los que tengamos definidos en el mismo espacio de nombres tendrán el mismo ámbito, por tanto, no podremos definir dos métodos extensores que tengan la misma sobrecarga (se llamen igual, reciban los mismos parámetros y amplíen la funcionalidad de la misma clase) si ambos están definidos en el mismo espacio de nombres. Si lo hacemos, el propio IDE de Visual C# 2008 nos avisará de este hecho, incluso antes de compilar el código (mostrando el error subrayado con los típicos dientes de sierra en rojo).

Precedencia en el ámbito de los métodos extensores

Cuando definamos métodos extensores en C# debemos tener en cuenta que el ámbito de éstos dependerán de dónde estén declarados, teniendo preferencia los que estén definidos en el mismo espacio de nombres desde dónde se utilizan.

Por ejemplo, si tenemos dos clases estáticas que tienen el mismo nombre, pero cada una de ellas está definida en un espacio de nombres distinto (esto es evidente, ya que no podemos tener dos clases

que se llamen igual en el mismo espacio de nombres), al usar los métodos estáticos de la forma habitual (es decir, no usándolos como métodos de extensión), solo tendremos acceso a los que estén definidos en la clase del mismo espacio de nombres. Esto es independiente de que tengamos o no una importación al otro espacio de nombres, ya que al usar el nombre de la clase, el compilador solo verá la definida en el propio espacio de nombres desde el que se utiliza. Sin embargo, si no hay conflictos de nombres en los métodos, no habrá problemas de que esas clases se llamen de la misma forma y por tanto se podrán usar métodos extensores de ambas clases.

Aclaremos esto con un ejemplo.

En el código del listado 5.7 tenemos una clase que define un método extensor para la clase **string**. Esa clase se llama **Extensiones** y como vemos, podemos usar el método **Len** tanto como una extensión de la clase **string** (asignación a la variable **i1**) o como un método normal (asignación a **i3**), en este último caso debemos indicar el nombre de la clase en el que está definido.

En ese mismo código vemos que tenemos una importación a otro espacio de nombres, el código de ese otro espacio de nombres es el mostrado en el listado 5.8. Como vemos, la clase que define los métodos extensores también se llama **Extensiones** y además de definir otro método extensor (**LenTrim**, usado en el listado 5.7 para asignar la variable **i2**), podemos comprobar que también define un método extensor para la clase **string** que tiene el mismo nombre que el definido en el código del listado 5.7 (**Len**).

Cuando se llama al método **Len** (ya sea directamente o como método extensor), aunque haya dos definiciones en espacios de nombres distintos, se utiliza el definido en el mismo espacio de nombres desde el que se utiliza. Sin embargo, al usar el método extensor **LenTrim** se usará el definido en el listado 5.8 (entre otras cosas porque es el único que existe con ese nombre).

De acuerdo, esto es evidente, pero solo es para aclarar de que aunque tengamos la importación al espacio de nombre **_05_metodosExtensores**, la clase **Extensiones** definida en ese espacio de nombres no la podremos usar directamente desde el método **Main**, ya que al llamarse de la misma forma que la definida en el espacio de nombres **_04_metodosExtensores** (que es el espacio de nombres que contiene a la clase que define el método **Main**), la definida en este espacio de nombres tendrá preferencia sobre la definida en el otro, y de hecho el compilador actuará como si solo existiera la que está en el mismo espacio de nombres, sin embargo a la hora de usar los métodos extensores, si se tiene en cuenta que existe esa clase y que está en ámbito (siempre que exista una importación al otro espacio de nombres).

Todo esto nos sirve para aclarar una serie de detalles que debemos tener en cuenta y que solo a la hora de usar los métodos extensores es cuando tienen su importancia, ya que si las tuviéramos que usar de forma normal, es evidente de que a la hora de resolver los conflictos de nombres, debemos hacerlo usando el nombre completo de la clase, es decir, indicando también el espacio de nombres.

```
using _05_metodosExtensores;

namespace _04_metodosExtensores
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = " Hola, Mundo ";

            int i1 = s1.Len();
            Console.WriteLine("{0}.Len = {1}\n", s1, i1);

            int i2 = s1.LenTrim();
            Console.WriteLine("{0}.LenTrim= {1}\n", s1, i2);

            int i3 = Extensiones.Len(s1);
            Console.WriteLine("Extensiones.Len({0})= {1}\n", s1, i3);

            Console.ReadLine();
        }
    }

    static class Extensiones
    {
        public static int Len(this string str)
        {
            if(string.IsNullOrEmpty(str))
                return 0;

            return str.Length;
        }
    }
}
```

Listado 5.7. Los conflictos se resuelven de forma diferente según donde se declaren los métodos extensores

```
namespace _05_metodosExtensores
{
    public static class Extensiones
    {
        public static int Len(this string str)
        {
            if(string.IsNullOrEmpty(str))
                return 0;

            return str.Length;
        }

        public static int LenTrim(this string str)
        {
            if(string.IsNullOrEmpty(str))
                return 0;

            return str.Trim().Length;
        }
    }
}
```

Listado 5.8. Métodos extensores definidos en otro espacio de nombres

Definir las clases con métodos extensores en su propio espacio de nombres

Para evitar los conflictos con métodos extensores que tienen el mismo nombre, se recomienda que las clases que definen métodos extensores estén en su propio espacio de nombres, de esta forma, podemos solucionar muchos de los conflictos añadiendo o quitando importaciones de espacios de nombres.

Por supuesto, esos conflictos de nombres en los métodos extensores no ocurrirán porque nosotros mismos hemos definido varias veces un método extensor en varias partes (es posible que ocurra, pero no será lo habitual). El problema es que podemos tener referencias a otros ensamblados que también definan un método extensor con el mismo nombre que nosotros hemos definido o bien es posible que dos ensamblados de los que tenemos en las referencias los definan, ya que ése es uno de los mayores problemas que nos podemos encontrar, y es que no hay nada que evite que dos programadores le den el mismo nombre a un método extensor.

Si los métodos extensores los creamos en el mismo proyecto en el que los vamos a usar, deberíamos definir un espacio de nombres adicional para contener el tipo en el que definimos los métodos extensores. De esta forma, tendríamos “aislados” (o casi) los métodos extensores del resto del código de nuestro proyecto.

Si la cosa puede empeorar, seguro que empeora

Pues sí, pero esto no es necesario que nos lo diga Murphy con sus leyes para que sepamos que puede ocurrir. Y con los métodos extensores seguro que alguna que otra vez nos encontraremos con conflictos de nombres y por tanto no podremos usarlos, salvo que el programador que los haya desarrollado lo haya hecho con cierto planteamiento y sabiendo que estos conflictos de nombres pueden existir, independientemente del nombre que tenga la clase que contiene los métodos extensores.

Por supuesto, estos conflictos se darán porque existan dos (o más) métodos extensores que tengan los mismos nombres (y se apliquen al mismo tipo de datos). En estos casos conflictivos no podemos hacer mucho, salvo decidir qué métodos extensores queremos usar, y en ese caso, agregar solo una importación a los espacios de nombres que queramos tener en ámbito para usar los métodos extensores que nos convenga.

Por esa razón, es importante definir las clases con métodos extensores en su propio espacio de nombres, de forma que le resulte cómodo al programador que use nuestros ensamblados decidir si añade o no una importación a esos espacios de nombres.

Aún así, los conflictos de nombres seguirán existiendo, al menos si hay varios métodos extensores con el mismo nombre (aunque estén definidos en espacios de nombres distintos). Y lo peor de todo es que si en esos espacios de nombres hay otros métodos extensores que nos interesan usar, tendremos que decidir cuáles vamos a usar en nuestro código, ya que si en un espacio de nombres se definen unos y en otros se definen otros distintos, pero sigue habiendo uno que tenga el mismo nombre, no podremos utilizarlos todos, por tanto, debemos plantearnos muy bien si necesitamos usar esos métodos extensores, y de hacerlo, tendremos que sopesar cuáles nos interesa usar en cada momento.

Nota

En el código de ejemplo que acompaña al libro, vemos estos conflictos de nombres al usar varias definiciones de métodos extensores que se aplican a la misma clase y tienen el mismo nombre y cómo resolver algunos de ellos al definirlos en espacios de nombres diferentes.

Conflictos con métodos existentes

Como estamos viendo, los métodos extensores sirven para añadir funcionalidad a clases existentes, pero ¿qué ocurre si definimos un método que ya existe en la clase que queremos extender?

Este conflicto lo podemos tener en dos situaciones: una es que creamos un método que inicialmente no esté definido en la clase que estamos extendiendo, pero que en una nueva versión de esa clase, el autor de la clase decida añadir un nuevo método con la misma firma del que nosotros definimos anteriormente; la otra es que ese método ya esté definido, y simplemente no nos hayamos dado cuenta de su existencia (este caso sería extraño, pero puede ocurrirle a algún programador despistado). Pero como nosotros no somos programadores despistados, la situación que nos puede llevar a declarar un método exactamente igual a uno que ya existe en la clase original, es la de querer darle una funcionalidad mejorada a la que la propia clase le da a ese método.

En estos casos, siempre gana la definición que haya en la clase que queremos extender.

De hecho, no se producirá ningún error (ni advertencia) si definimos un método extensor que tiene la misma firma que uno existente en la clase.

Por ejemplo, si definimos un método extensor para la clase **string** llamado **Trim** que no reciba ningún parámetro, el compilador usará la definición que ya existe en la clase, y nuestro método extensor, simplemente lo ignorará.

Sobrecargas en los métodos extensores

Como acabamos de ver, no tiene ninguna utilidad práctica crear métodos extensores que tengan la misma firma que un método de instancia definido en la clase que queremos extender, lo que sí podemos hacer es crear sobrecargas de métodos existentes. En este caso, debemos tener en cuenta todo lo que ya sabemos sobre las sobrecargas, por tanto, las nuevas sobrecargas deben diferenciarse de los métodos existentes en que tengan parámetros de tipos diferentes a los definidos en el tipo de datos que queremos extender o que el número de parámetros sea diferente. Y como ya comenté al principio, solo podemos crear sobrecargas de métodos de instancia, no de propiedades, incluso aunque los métodos extensores tengan firmas diferentes a las propiedades.

Nota

Solo podemos crear sobrecargas de los métodos de instancia, no de los que sean estáticos (compartidos), ya que los métodos estáticos siempre se deben usar indicando el tipo de datos que los define.

Sabiendo esto, podremos crear sobrecargas de cualquiera de los métodos de instancia que estén definidos en un tipo de datos, pero recordando que solo se usarán aquellos que realmente estén creando una sobrecarga; los que tengan la misma firma, simplemente se ignorarán.

Nota

Aunque no tenga ningún efecto práctico, podemos crear un método extensor que sobrecargue a una propiedad, pero no lo podremos usar como miembro del tipo que estamos extendiendo, aunque sí lo podremos usar como un método “normal”, es decir, sin ninguna relación directa con la clase que queremos extender.

Sobrecargas y array de parámetros en los métodos extensores

Como sabemos, en C# podemos definir un *array* de parámetros en los métodos usando la palabra clave **params**. Esta forma de definir parámetros nos permite hacer que en realidad se conviertan en parámetros (o argumentos) opcionales, ya que podemos indicar cualquier cantidad (incluso no indicar ninguno) a la hora de llamar al método. Esto es muy parecido a los parámetros opcionales de Visual Basic, aunque no exactamente lo mismo, ya que Visual Basic permite indicar parámetros opcionales independientes y sin necesidad de estar en un *array*, y por tanto todos deben ser del mismo tipo.

En los métodos extensores también podemos definir un parámetro de este tipo, pero debemos tener en cuenta que el primero de los parámetros de un método extensor **no** puede ser opcional, ya que ese primer parámetro le sirve al compilador para saber el tipo de datos que queremos extender. Sin embargo, los demás parámetros sí que pueden ser opcionales y funcionarán de la misma forma a la que estamos acostumbrados.

Nota

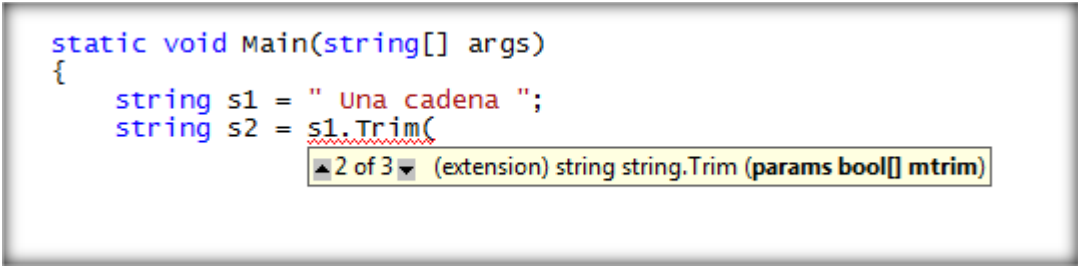
En C# el concepto de parámetro opcional no es el mismo que en Visual Basic, ya que en C# solo se puede usar un array de un tipo concreto, pero al permitir indicar uno o más valores de ese tipo, lo podemos considerar como parámetros opcionales. Aunque en otros libros de C# a esta característica se suele referir como “array de argumentos variables” (o simplemente array de parámetros) yo prefiero llamarla array de parámetros opcionales, ya que considero que es una forma más directa de llamar a esta característica, seguramente porque uso con bastante asiduidad Visual Basic.

Debemos tener en cuenta que el uso de un *array* de parámetros opcionales, en realidad, es como si creáramos sobrecargas, por tanto, pueden entrar en conflicto con otros métodos que ya estén definidos en el tipo que estamos extendiendo. Si esto ocurre, que al usar los parámetros opcionales se cree un método que ya existe, esa sobrecarga se ignorará, pero no la que en realidad agregue parámetros que no existen en la clase que estamos extendiendo. Con un ejemplo lo veremos más claro.

En el listado 5.9 vemos una sobrecarga del método **Trim** de la clase **string** que usa un *array* de parámetros opcionales. Al usar este método extensor, IntelliSense nos mostrará ese método de la forma habitual en este tipo de parámetros (ver la figura 5.3), pero en realidad solo llamará a nuestro método extensor en la sobrecarga que no entre en conflicto con la definida en la clase **string**; en este ejemplo será cuando se indique un valor de tipo **bool** en el segundo argumento. Ese valor puede ser verdadero o falso, pero debemos indicarlo expresamente para que se utilice nuestro método extensor, ya que si no indicamos nada, la definición del método **Trim** que existe en la clase **string** tendrá preferencia con respecto a nuestra definición.

```
public static string Trim(this string str, params bool[] mtrim)
{
    if(mtrim[0])
        return TrimMid(str);
    return str.Trim();
}
```

Listado 5.9. Método extensor con un array de parámetros opcionales



```
static void Main(string[] args)
{
    string s1 = " Una cadena ";
    string s2 = s1.Trim(
        ▲ 2 of 3 ▼ (extension) string string.Trim (params bool[] mtrim)
```

Figura 5.3. Sobrecarga con array de argumentos variables (parámetros opcionales)

¿Qué tipos de datos podemos extender?

Podemos crear métodos extensores para cualquier tipo de datos: clases, estructuras, interfaces e incluso delegados. Lo que siempre debemos recordar es que esos métodos extensores serán tratados como métodos de instancia y desde ellos solo podemos acceder a los miembros públicos del tipo que extendemos.

Cuando creamos un método extensor para una interfaz, ese método se puede usar en cualquier clase que implemente la interfaz a la que le damos nueva funcionalidad, ni qué decir tiene, que desde ese método extensor solo podremos acceder a los miembros que la interfaz definida. La ventaja de los métodos extensores aplicados a las interfaces es que los podremos usar en todas las clases que implementen directa o indirectamente esa interfaz.

Si una clase que implementa esa interfaz define un método que coincide con el que hemos definido como método extensor de la interfaz, al usar la llamada desde una instancia de la clase, siempre tendrá preferencia el método definido en la propia clase; pero si esa interfaz la obtenemos a partir de esa instancia que define métodos con la misma firma que los métodos extensores que hemos definido para la interfaz, siempre se usarán los métodos extensores ya que la extensión la estamos haciendo a la interfaz y la interfaz no sabe nada de los métodos que una clase define de forma independiente.

En el código del listado 5.10 vemos la definición de una interfaz, una clase que la implementa directamente y otra clase que se deriva de esa clase. En ésta última (que también hereda la implementación de la interfaz) definimos un método que coincide con el método extensor definido en el listado 5.11. La forma de usar estos métodos extensores la vemos en el listado 5.12, y como podemos comprobar, el método *Imprimir* lo podemos usar en todas las instancias, pero cuando lo usamos directamente en un objeto del tipo *OtraPrueba*, el método que se usa es el de instancia, sin embargo, cuando ese objeto lo asignamos a uno del tipo de la interfaz (variable *ipr2*), el método al que se llama es al extensor.

```
public interface IPrueba
{
    string Mostrar();
}

public class Prueba : IPrueba
{
    public string Contenido;
    public string Mostrar()
    {
        return Contenido;
    }
}

public class OtraPrueba : Prueba
{
    public string Imprimir()
    {
        return Contenido + " en OtraPrueba";
    }
}
```

Listado 5.10. Interfaz y clases que usan la interfaz

```
public static class Extensiones
{
    public static string Imprimir(this IPrueba ipr)
    {
        return ipr.Mostrar();
    }
}
```

Listado 5.11. Un método extensor para la interfaz IPrueba

```
var pr1 = new Prueba { Contenido = "Prueba 1" };
var pr2 = new OtraPrueba { Contenido = "Prueba 2" };

// Se usa la extensión
var s1 = pr1.Imprimir();
Console.WriteLine(s1);

// se usa el método de instancia
s1 = pr2.Imprimir();
Console.WriteLine(s1);

IPrueba ipr = pr1;
// Se usa la extensión
s1 = ipr.Imprimir();
Console.WriteLine(s1);

IPrueba ipr2 = pr2;
// Se usa la extensión
// ya que aunque el objeto pr2 define ese método,
// la interfaz no lo define
s1 = ipr2.Imprimir();
Console.WriteLine(s1);
```

Listado 5.12. Ejemplos de uso del método extensor de la interfaz

En cuanto a crear métodos extensores para los delegados, debemos tener en cuenta que ese método siempre lo usaremos con una variable del tipo del delegado que queremos extender. La ventaja es que el método extensor no tendrá conflictos con ningún método del tipo, ya que los delegados no pueden definir métodos.

El delegado que usaremos en el método extensor (el indicado en el primer parámetro del método) hará referencia al método que tenemos asociado con ese delegado, y como veremos en este mismo capítulo, los métodos que podemos usar con un delegado pueden ser métodos anónimos, por tanto, el método extensor siempre llamará al método asociado con el delegado que reciba en el primer argumento. Esto es evidente, pero es bueno tenerlo claro, ya que el valor que usemos en el método extensor en realidad depende de lo que haga el delegado, aunque tampoco es algo de lo que debemos preocuparnos, ya que en teoría para el código del método extensor eso es algo transparente y no debe ser un impedimento para usar el delegado.

Todo este galimatías es porque la única forma práctica de extender los delegados es si ese delegado recibe algún parámetro con el que poder trabajar, o bien porque el delegado devuelva algún valor, ya que si el delegado ni define parámetros ni devuelve algo, la verdad es que poco podemos hacer en los métodos extensores, en el sentido de que eso que hagamos tenga alguna relación con el método que el delegado tenga asociado.

En el código del listado 5.13 tenemos la definición de un delegado que recibe un parámetro de tipo **string** y devuelve un valor de ese mismo tipo. En el listado 5.14 tenemos la definición de un método extensor para ese delegado, el cual usaremos con dos argumentos de tipo cadena. En el código del listado 5.15 definimos un método que tiene la misma firma que el delegado, y finalmente, en el listado 5.16 vemos cómo podemos usar ese delegado y el método extensor.

```
delegate string PruebaDelegado(string str);
```

Listado 5.13. Definición de un delegado para el que crearemos un método extensor

```
static class Extensiones
{
    public static string Saludo(this PruebaDelegado deleg,
                                string str,
                                string msg)
    {
        return deleg(msg + " " + str);
    }
}
```

Listado 5.14. Un método extensor para el delegado del listado 5.13

```
static string pruebaDelegado1(string str)
{
    return str;
}
```

Listado 5.15. Un método con la misma firma del delegado del listado 5.13

```
PruebaDelegado pd1 = pruebaDelegado1;
var s5 = pd1("Mundo");
Console.WriteLine(s5);

s5 = pd1.Saludo("Guille", "Hola");
Console.WriteLine(s5);
```

Listado 5.16. Ejemplo de uso del delegado y el método extensor

En el listado 5.17 vemos cómo usar el método extensor definido en el listado 5.14, pero en lugar de usar el delegado con un método existente, lo definimos de forma que utilice un método anónimo (o expresión *lambda*).

```
PruebaDelegado pd3 = str => "Hola, '" + str + "'";  
Console.WriteLine(pd3("Guille"));  
  
s5 = pd3.Saludo("Guille", "Hola");  
Console.WriteLine(s5);
```

Listado 5.17. Asignación de una expresión lambda con la firma del delegado y ejemplo de uso

Reflexionando sobre los métodos extensores

Con el título no me refiero a usar reflexión (*reflection*) con los métodos extensores, sino a que antes de crear un método extensor debemos reflexionar sobre si debemos crearlo.

La primera recomendación es que solo definamos métodos extensores cuando realmente sea necesario, es decir, para clases que o bien no hemos creado nosotros o que no permitan crear clases derivadas (estén marcadas como selladas, **sealed** en C# o **NotInheritable** en Visual Basic).

En la medida de lo posible, es más efectivo crear un nuevo tipo derivado, que añadir un método extensor, entre otras cosas, porque los métodos de instancia (los definidos en las clases) siempre tienen preferencia sobre los métodos extensores.

Si vamos a definir métodos extensores, hacerlo en su propio espacio de nombres, y si esos métodos extensores solo los usaremos en la aplicación actual, y con la intención de mejorar o ampliar algunas clases usadas en ese proyecto, lo ideal es que esos métodos extensores solo sean visibles dentro de ese proyecto, es decir, no declararlos en una clase pública.

No voy a entrar en la polémica de si es conveniente crear este tipo de métodos, ya que, en esto (como en todo), dependerá de lo que necesitemos hacer, y si un método extensor nos soluciona el problema, ¿por qué no vamos a definirlo? Pero no caigamos en la trampa de que es muy fácil definir nuevos métodos de esta forma sin necesidad de modificar las clases que estamos ampliando (particularmente si somos los autores de esas clases), ya que siempre será más correcto ampliar la funcionalidad de nuestras clases que crear métodos extensores, porque, como hemos podido comprobar, nos podemos encontrar con problemas de ámbito y conflictos de nombres.

Pero como siempre, el lector tiene la última palabra. Y si, además, tiene la información suficiente para trabajar con los métodos extensores, con más fundamento puede elegir lo que estime más oportuno.

Versiones

Esta característica solo la podemos usar con .NET Framework 3.5.

Parte 3

C# y LINQ

Esta es la parte final del libro en la que a lo largo de cinco capítulos veremos todas las novedades de C# directamente relacionadas con LINQ.

Para utilizar las consultas de LINQ necesitaremos los nuevos ensamblados de .NET Framework 3.5 y para utilizar todo lo referente a *LINQ to SQL* además debemos agregar una referencia al ensamblado **System.Data.Linq.dll**.

Con el Service Pack 1 para .NET Framework 3.5 (y Visual Studio 2008) tenemos a nuestra disposición una nueva forma de acceso a datos: *ADO.NET Entity Framework*, que, entre otras cosas, incluye *LINQ to Entities*, de este último nos ocuparemos al final del último capítulo.

(Esta página se ha dejado en blanco de forma intencionada)

Capítulo 6

C# y LINQ (I)

C# y LINQ

El tema de este capítulo del libro de novedades de C#, sin lugar a dudas es el que más atención y hasta polémica ha levantado. Y para ser sinceros, casi todas las novedades que se han introducido en .NET Framework 3.5 (y por extensión en C#) están relacionadas directa o indirectamente con esta nueva tecnología que se conoce como *Language INtegrated Query* (lenguaje de consulta integrado).

LINQ (que son las siglas en inglés de esta novedad de .NET 3.5) es la tecnología que nos permite usar instrucciones al estilo de las consultas de SQL (*Structured Query Language*, lenguaje de consulta estructurado) en nuestro código de C#, o al menos esa es la forma más simple de presentar esta tecnología, pero como veremos, hay muchas más cosas detrás de LINQ que simples instrucciones para hacer consultas.

En este primer capítulo dedicado a LINQ, nos ocuparemos de las diferentes tecnologías relacionadas con LINQ. En los siguientes veremos las diferentes instrucciones que se incluyen en C# para dar soporte a estas consultas integradas en el lenguaje. Aunque antes de entrar en detalles, veamos un ejemplo de una consulta LINQ usando las instrucciones propias de Visual C# 2008.

Un ejemplo de LINQ para ir abriendo boca

A lo largo de este capítulo, iremos viendo las nuevas instrucciones que se han añadido a C# para dar soporte a todo lo relacionado con LINQ, pero para ir calentando motores, veamos un ejemplo para que sepamos lo que nos vamos a ir encontrando en el resto de éste y siguientes capítulos.

El código que veremos a continuación va a extraer de una colección de objetos de una clase llamada *Artículo* todos los elementos cuyo código empiece por cero. Esa colección la podemos crear usando cualquiera de las dos formas que nos permite C# para inicializar *arrays* y colecciones. Si queremos tener la posibilidad de clasificar los elementos de la lista generada, usaremos una colección del tipo *List<T>*, pero como veremos en un momento, con las nuevas instrucciones que LINQ nos ofrece, podremos generar los datos clasificados en el orden que deseemos. En este ejemplo, he optado por crear una colección del tipo *List<Artículo>* más que nada para que resulte más fácil agregar nuevos elementos a la colección, aunque la verdad es que si esos datos solo los vamos a utilizar en las consultas de LINQ, da igual que sea de un tipo o de otro, incluso podíamos crear un *array*, ya que la sintaxis de C# nos permite crear colecciones a partir de un *array* de una forma muy simple, particularmente apoyándonos en el método extensor *ToList*.

En el listado 6.1 vemos cómo generar esa colección usando la inicialización de objetos de cada uno de los artículos que queremos agregar a la lista.

```
var artículos = new List<Artículo>() {
    new Artículo("01")
        {Descripción = "Refresco lima",
         PrecioVenta = 0.92M},
    new Artículo("02")
        {Descripción = "Refresco naranja",
         PrecioVenta = 0.9M},
    new Artículo("00")
        {Descripción = "Refresco cola",
         PrecioVenta = 0.95M},
    new Artículo("11")
        {Descripción = "vino tinto brick",
         PrecioVenta = 0.55M},
    new Artículo("21")
        {Descripción = "Rioja 0.75 ml",
         PrecioVenta = 1.2M}
};
```

Listado 6.1. Una colección de elementos del tipo Artículo

La clase *Artículo* usada en estos ejemplos es la mostrada en el listado 6.2 en el que las propiedades de esa clase las he definido usando la auto implementación de propiedades, salvo en el caso de la propiedad *IVA*, ya que le he querido dar un valor predeterminado.

```
class Artículo
{
    public string Código { get; set; }
    public string Descripción { get; set; }
    public decimal PrecioVenta { get; set; }

    protected decimal m_IVA = 16M;
    public decimal IVA
    {
        get { return m_IVA; }
        set { m_IVA = value; }
    }

    public Artículo()
    {
    }

    public Artículo(string código)
    {
        this.Código = código;
    }

    public override string ToString()
    {
        return String.Format(
            "{0}, {1}, {2}",
            Código, Descripción, PrecioVenta);
    }
}
```

Listado 6.2. La clase Artículo usada en el ejemplo

Nota

La clase Artículo está simplificada para dar una idea de qué propiedades define. En el sitio Web del libro está el código completo junto a otras clases utilizadas a lo largo de esta parte del libro dedicada a LINQ.

Lo siguiente que vamos a hacer es extraer de la colección **artículos** los elementos cuya propiedad **Código** empiece por cero. Esos datos los devolveremos ordenados de forma ascendente por el contenido de la propiedad **PrecioVenta**. Si la colección **artículos** fuese una tabla de una base de datos, el código de una consulta SQL que utilizaríamos sería similar al mostrado en el listado 6.3.

```
SELECT [Descripción],[PrecioVenta]
FROM [Articulos]
WHERE LEFT(Código, 1)='0'
ORDER BY PrecioVenta ASC
```

Listado 6.3. Código SQL de una consulta

En el listado 6.4 vemos cómo extraer los datos usando las instrucciones de consulta integradas en el lenguaje (LINQ), que como podemos comprobar, es muy parecido al código del listado 6.3, solo que en esta ocasión utilizamos instrucciones de C#. En la comparación de la cláusula **where** he usado el método **StartsWith** de la clase **string** que es el que más se asemeja a la función **LEFT** del código de T-SQL.

```
var refrescos = from d in artículos
                 where d.Código.StartsWith("0")
                 orderby d.PrecioVenta ascending
                 select new { d.Descripción, d.PrecioVenta };
```

Listado 6.4. Consulta de LINQ para extraer los datos de una colección

Como vemos en el listado 6.4, en las consultas de LINQ la instrucción **select** se indica al final, pero básicamente conseguimos lo mismo que con el código de SQL, que son todos los artículos cuyo código empiecen por cero, pero en lugar de obtener todos los campos de la “tabla” solo se devuelven los que hemos indicado después de **select**, en estos casos, un tipo anónimo con la descripción y el precio de venta.

Para mostrar los datos que contiene la variable **refrescos** lo podemos hacer como vemos en el listado 6.5. En ese código debemos fijarnos en dos cosas: una de ellas es que la variable **r** (cada uno de los elementos de la colección **refrescos**) tiene dos propiedades: **Descripción** y **PrecioVenta**, en realidad cada elemento de la colección es un tipo anónimo con esas dos propiedades; la segunda cosa que debe llamar nuestra atención es el uso de un método extensor (**PadFillLeft**) que yo he programado para la clase **string** (ver el listado 6.6), de forma que me devuelva la cantidad de caracteres indicados en el parámetro (en este ejemplo, 15), con idea de que se muestre el texto alineado, tal como vemos en el resultado mostrado en la figura 6.1. La razón de crear este método extensor y no usar

uno de los existentes en la clase **string**, o usar las opciones de formato en el método **WriteLine** de la clase **Console**, es porque si la cadena tiene más de 15 caracteres no se recortaría, como es en este caso, ya que todas las funciones de formato siempre devuelven como mínimo la cantidad que indicamos, pero si el original tiene más caracteres (como en este caso en que el primer elemento mostrado tiene 16) se mostrarán todos los caracteres. Pero no mezclemos temas y sigamos con todas las novedades de C# relacionadas con LINQ.

```
foreach(var r in refrescos)
    Console.WriteLine("{0} {1:0.00}",
        r.Descripción.PadFillLeft(15),
        r.PrecioVenta);
```

Listado 6.5. Mostrar los datos del resultado de la consulta del listado 6.4

```
static class Extensiones
{
    public static string PadFillLeft(this string str, int total)
    {
        return (str + new string(' ', total)).Substring(0, total);
    }
}
```

Listado 6.6. El método PadFillLeft para ajustar el ancho de las cadenas al número de caracteres indicados

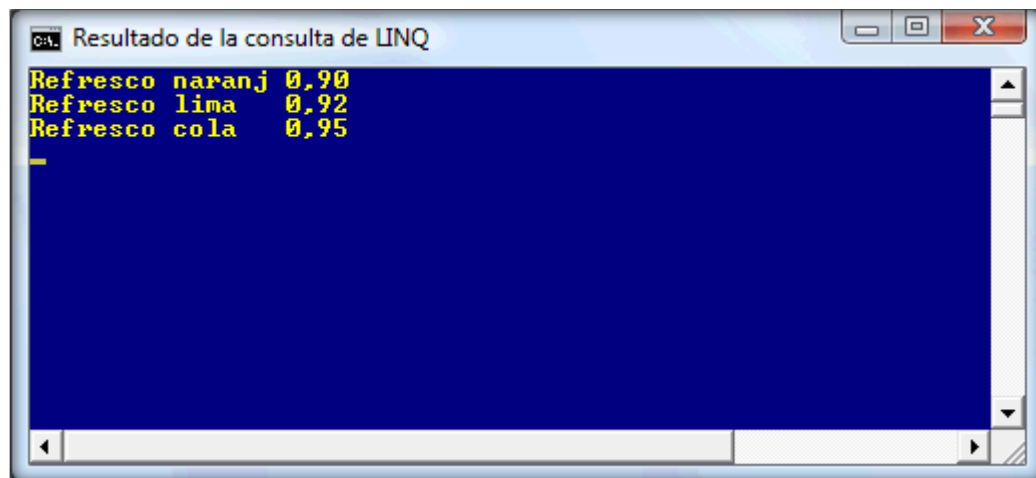


Figura 6.1. Resultado de la consulta LINQ

Si el lector no tiene muy claro qué es lo que hace todo el código que hemos visto, le pido que no se preocupe, que este ejemplo simplemente lo he usado para ir abriendo boca, como suelen decir en la tele, para que no haga *zapping* y cambie de canal. En un momento veremos todos los detalles de las nuevas instrucciones que C# incluye para dar soporte a LINQ. Pero antes veamos un poco de teoría sobre LINQ y las diferentes versiones que .NET Framework 3.5 pone a nuestra disposición.

Los diferentes sabores de LINQ

Antes de adentrarnos en los detalles de LINQ y la integración en C# del lenguaje de consultas, veamos qué variantes (o proveedores) de LINQ nos ofrece .NET Framework 3.5 y podemos usar desde C#.

Básicamente hay tres tecnologías basadas en LINQ:

- **LINQ to Objects**, permite utilizar las instrucciones de consultas en objetos que están en la memoria.
- **LINQ to XML**, es la parte que nos permite acceder a documentos XML, descubriremos todas las clases que necesitamos para consultar este tipo de datos.
- **LINQ to ADO.NET**, nos permite utilizar las instrucciones de consultas en objetos de bases de datos. Esta característica se divide a su vez en otras dos tecnologías:
 - **LINQ to DataSet** proporciona mejoras en el acceso a datos con los ya conocidos **DataSet**, de forma que facilita el acceso por medio de las instrucciones de consultas.
 - **LINQ to SQL** por su parte proporciona acceso a bases de datos relacionales de forma que nosotros trabajemos con elementos de programación como clases para representar tablas o métodos para representar procedimientos almacenados o funciones.

De estas formas en que se presenta LINQ, nos centraremos principalmente en la primera: *LINQ to Objects*, aunque de las otras también veremos ejemplos de cómo usarlas, pero sin profundizar demasiado, ya que otros compañeros de **Solid Quality Mentors** (más expertos que yo en todo lo relacionado con acceso a datos) publicarán en breve libros electrónicos que profundizarán en esas tecnologías de acceso a datos. Pero eso no significa que no veremos en este libro cómo utilizar esas tecnologías desde C#, ya que sí veremos ejemplos de acceso a datos, usando *LINQ to DataSet* y *LINQ to SQL*, y el nuevo diseñador de consultas incluido en Visual Studio 2008: *O/R Designer* (Diseñador R/O) o **diseñador relacional de objetos**. Pero todo esto será después, antes centrémonos en conocer qué es LINQ y cómo utilizar esa integración en Visual C# 2008.

Nota

Con la aparición del Service Pack 1 de .NET Framework (y Visual Studio 2008), se ha incluido LINQ to Entities que está basado en ADO.NET Entity Framework y permite el uso de LINQ en bases de datos relacionales diferentes a SQL Server.

LINQ to Objects

Esta es la denominación de la tecnología LINQ que nos permite trabajar con los datos que tenemos en memoria, podemos aplicar las consultas LINQ a cualquier objeto que implemente las interfaces **IEnumerable** o **IEnumerable<T>**, es decir, todos los *arrays* o colecciones, ya sean *generic* o normales.

Esto nos permite utilizar las instrucciones de consultas con prácticamente cualquier objeto que pueda contener más de un elemento. En el listado 6.4 ya vimos un ejemplo de cómo realizar una consulta a una colección, pero mejor veamos algo más sencillo para que entendamos mejor cómo funcionan estas nuevas instrucciones de C# para poder usar todo lo referente a LINQ.

Elementos básicos de una consulta LINQ

Empecemos viendo un ejemplo muy básico, que iremos ampliando para ver las distintas posibilidades que nos ofrece el lenguaje de consultas integrado en C#.

En el listado 6.7 definimos un *array* numérico y creamos un objeto que contiene una consulta LINQ. En ese código se seleccionan todos los valores numéricos del *array* que sean mayores de 4. La variable **res** contiene la expresión de la consulta, es decir, contiene las instrucciones necesarias para realizar las operaciones que hemos indicado, en este caso, esas operaciones serán: todos los valores que haya en el *array* **nums** cuyo valor sea superior a 4.

```
var nums = new[] { 1, 9, 8, 2, 5, 7, 4, 3, 6 };  
var res = from n in nums where n > 4 select n;  
foreach(var v in res)  
    Console.WriteLine(v);
```

Listado 6.7. Una consulta LINQ con un array numérico

Cuando usamos esa variable en el bucle **foreach** es cuando se pone en marcha toda la maquinaria de LINQ para hacer efectivo ese código, dando como resultado una colección del tipo **IEnumerable<int>** con los valores 9, 8, 5, 7 y 6 que son los que cumplen la condición indicada por la cláusula **where**.

Antes de entrar en detalles, analicemos la expresión de consulta y veamos lo que hace LINQ.

from n in nums, esto lo podemos leer como: asigna a la variable indicada después de **from** cada uno de los valores de la colección indicada después de la cláusula **in**.

where n > 4, toma solo los elementos que coincidan con esta expresión.

select n, le indica al compilador qué elemento queremos incluir en el resultado de la consulta, en este caso un valor simple, el de cada número que cumpla la condición de la cláusula **where**.

La variable que recibe esa expresión de consulta es del tipo **IEnumerable<T>**, siendo **T** el tipo devuelto por **select**. En este ejemplo un tipo **int**. En esa colección estarán solo los valores que cumplan la condición indicada, es decir, los valores superiores a 4.

Ejecución aplazada y ejecución inmediata

Cuando el compilador se encuentra con esta expresión de consulta no ejecuta ese código de forma inmediata, simplemente prepara la variable (**res** en nuestro ejemplo del listado 6.7) para contener esa expresión, esto es lo que se conoce como consulta de ejecución aplazada (o ejecución diferida), es decir, cada vez que utilicemos esa variable, es cuando se ejecuta la consulta.

Esto significa, que si los elementos de la variable **nums** cambian, también cambiará el resultado final. Por ejemplo, el primer elemento del *array* es 1, si lo cambiamos a 10 y volvemos a ejecutar el bucle **foreach**, veremos que también se muestra ese nuevo valor, ya que cumple la condición usada con **where**. El listado 6.8 tiene ese cambio y en la figura 6.2 vemos el resultado de ejecutar los dos bucles.

```
nums[0] = 10;  
Console.WriteLine("Después de hacer nums(0) = 10:");  
foreach(var v in res)  
    Console.WriteLine(v);
```

Listado 6.8. Si cambiamos un valor del array y coincide con la condición, se incluirá en el resultado de la consulta

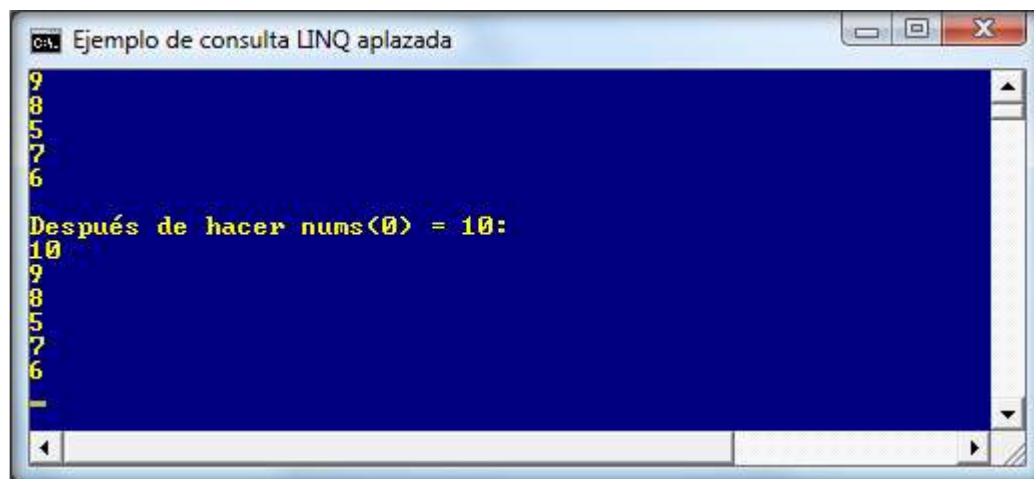


Figura 6.2. Resultado de ejecutar el código de los listados 6.7 y 6.8

Por tanto, cualquier cambio que hagamos a los valores del *array* se verá reflejado al utilizar la variable con la consulta. Al menos si el número de elementos de ese *array* no cambia, ya que si añadimos o quitamos elementos, la consulta usará los últimos valores que obtuvo antes del cambio de tamaño del *array*.

Esto último lo podemos comprobar añadiendo al código actual el mostrado en el listado 6.9, veremos que se muestra lo mismo que vemos al final de la figura 6.2.

```
// Quitamos un elemento del array
Array.Resize(ref nums, nums.Length - 1);
// Asignamos al primer elemento el valor 1
nums[0] = 1;

// Se mostrará lo que había antes del cambio
foreach(var v in res)
    Console.WriteLine(v);
```

Listado 6.9. Si cambiamos el número de elementos del array usado en la consulta, se mantienen los últimos valores que hubiera en la consulta

La mayoría de las consultas LINQ son de tipo aplazada (se realizan cuando se utiliza la variable que contiene la consulta), aunque hay ciertas funciones que pueden obligar a que la consulta se haga de forma inmediata, por ejemplo, si usamos funciones agregadas o bien convertimos el resultado de la consulta en una lista (colección de tipo **List<T>**) o un *array* por medio de los métodos **ToList** o **ToArray** respectivamente.

En el listado 6.10 tenemos estos dos casos comentados: en el primero, asignamos a la variable **res2** el resultado de la suma de los elementos que devuelve la consulta, debido a que la única forma de efectuar esa suma es sabiendo los elementos que contiene, la consulta debe ejecutarse de forma inmediata; en el segundo, al convertir esa colección en una lista también debe ejecutarse la consulta y una vez creada la lista devuelta por el método **ToList**, cualquier cambio que hagamos al *array* no afectará al resultado. En realidad, ambas consultas se ejecutan porque usamos un método que trabaja con el resultado de la consulta, y por tanto, lo que se asigna a las variables es lo que se conoce como un valor *singleton*, es decir, un valor directo en lugar de una expresión de consulta.

```
var res2 = (from n in nums where n > 4 select n).Sum();
Console.WriteLine(res2);

Console.WriteLine();

var res3 = (from n in nums where n > 4 select n).ToList();
foreach(var v in res3)
    Console.WriteLine(v);

Console.WriteLine();

nums[0] = 5;
foreach(var v in res3)
    Console.WriteLine(v);
```

Listado 6.10. Consultas de ejecución inmediata

La cláusula select

En todos estos ejemplos vemos cómo se utilizan las novedades que hemos estado examinando en capítulos anteriores. La principal de ellas es la inferencia de tipos, ya que el compilador infiere el tipo que deben tener las variables que reciben los resultados de las consultas, además de los tipos usados en la misma, por ejemplo, no es necesario indicar el tipo de la variable después de la cláusula

from. También estamos utilizando métodos de extensión, ya que tanto **Sum** como **ToList** son métodos que extienden la funcionalidad de la interfaz **IEnumerable<T>**. Y aunque no lo veamos en este ejemplo, la creación de tipos anónimos también será utilizada en la mayoría de las expresiones de consulta que creemos. En un momento veremos un ejemplo.

Normalmente en las expresiones de consultas, LINQ siempre se utiliza la cláusula **select** para indicar el valor que tendrá cada elemento de la colección creada. Habitualmente, después de **select** se indica la misma variable usada después de **from**, pero si queremos devolver valores que no sean ese valor simple, podemos indicar algo después de **select**. Ese “algo” será lo que se devuelva; por ejemplo, en el listado 6.11, en lugar de devolver un valor entero, en cada iteración del bucle que cumpla con la condición indicada después de la cláusula **where**, devolvemos una cadena formada por el texto entrecomillado, además del valor del número analizado (y que cumple esa condición).

Y si queremos utilizar un tipo anónimo como elemento a añadir a la colección generada por la consulta, podemos hacerlo tal como vemos en el listado 6.12.

```

var res = from n in nums
          where n > 4
          select "Número " + n;

```

Listado 6.11. Esta consulta devuelve los elementos como una cadena compuesta por el texto y el valor

```

var res2 = from n in nums
           where n > 4
           select new
           {
               Valor = "Número " + n,
               EsPar = n % 2 == 0
           };

foreach(var v in res2)
    Console.WriteLine("{0}, es par = {1}",
                      v.Valor, v.EsPar);

```

Listado 6.12. Una consulta que contiene elementos de tipo anónimo

En este código definimos un tipo anónimo con dos propiedades, la primera de tipo **string** y la segunda de tipo **bool** que nos indica con un valor **true** si el número en cuestión es par. En el bucle **foreach** comprobamos que el tipo de datos se infiere a partir de ese tipo anónimo, ya que la variable usada en ese bucle implementa las dos propiedades definidas en la inicialización que hacemos después de **select**. En la figura 6.3 podemos ver el resultado de ejecutar ese código.

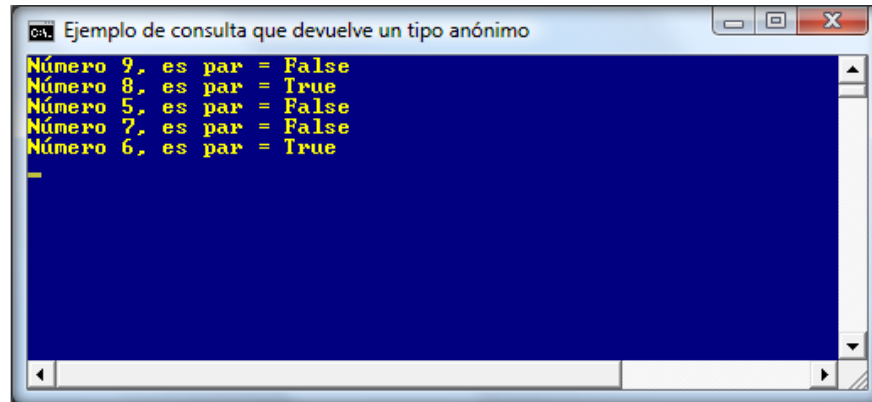


Figura 6.3. Resultado de ejecutar el código del listado 6.12

Ordenar los elementos de la consulta

Otra de las instrucciones que también usaremos con frecuencia es la que nos permite clasificar el resultado de la consulta: **orderby**. Esta ordenación la podemos hacer de forma ascendente o descendente, para ello usaremos las palabras clave **ascending** o **descending** respectivamente, (si no se indica ninguna de estas dos instrucciones, el valor predeterminado será ascendente). En el listado 6.13 vemos un ejemplo de uso de estas instrucciones de ordenación.

```
var res3 = from n in nums
           where n > 4
           orderby n descending
           select new
           {
               valor = "Número " + n,
               EsPar = n % 2 == 0
           };
```

Listado 6.13. Consulta con ordenación de los elementos

No es oro todo lo que reluce

Las nuevas instrucciones de consulta de C# en realidad son una forma “amigable” de utilizar todo lo que hemos visto en los capítulos anteriores, y en realidad, son instrucciones que el compilador finalmente convierte en llamadas a métodos extensores aderezados con funciones anónimas. Por ejemplo, la consulta del listado 6.14 también la podríamos escribir como la mostrada en el listado 6.15.

```
var res4 = from n in nums
           where n > 4
           orderby n descending
           select n;
```

Listado 6.14. Una consulta LINQ con las instrucciones de C#

```
var res5 = nums.where(n => n > 4).  
                orderByDescending(n => n).  
                select(n => n);
```

Listado 6.15. La consulta del listado 6.14 utilizando métodos extensores y expresiones lambda

Como podemos comprobar, es mucho más intuitivo el código del listado 6.14 que el correspondiente al listado 6.15, pero nos sirve para saber qué es lo que en realidad ocurre entre bastidores.

En el siguiente capítulo veremos cada una de las instrucciones de C# que se han incorporado al lenguaje para dar soporte a LINQ.

Versiones

Esta característica solo la podemos usar con .NET Framework 3.5 y debemos tener una referencia a System.Linq.dll (todas las plantillas de proyectos de Visual C# 2008 incluyen esa referencia).

(Esta página se ha dejado en blanco de forma intencionada)

Capítulo 7

C# y LINQ (II)

Instrucciones de C# para las consultas de LINQ

Como ya he comentado en varias ocasiones, C# incluye en el lenguaje nuevas instrucciones para permitirnos trabajar con todo lo relacionado con LINQ. Esas instrucciones son las mismas independientemente del proveedor de LINQ que estemos usando, ya que al compilador le da igual en qué datos las apliquemos, porque siempre lo haremos sobre objetos que permiten la utilización de esas instrucciones de consultas. Los detalles de si esos datos están directamente en la memoria (*LINQ to Objects*) o se han obtenido a partir de un documento XML (*LINQ to XML*) o proceden de un **Data-Set** (*LINQ to DataSet*) o es porque tenemos un proveedor de SQL Server que nos permite la utilización de esta forma de acceso a los diferentes elementos de una base de datos relacional (*LINQ to SQL*), en realidad solo nos tiene que preocupar a la hora de obtener el origen de esa información con la que vamos a trabajar.

Empecemos viendo esas instrucciones (o cláusulas) con ejemplos relacionados con la manipulación de los datos en memoria (*LINQ to Objects*), después veremos algunos ejemplos con las otras tecnologías.

Para poder utilizar las expresiones de consulta LINQ, debemos crear los proyectos usando la versión 3.5 de .NET Framework. En las plantillas de proyectos de Visual Studio 2008 (o de las versiones Express) siempre se agregan las referencias necesarias, así como todas las importaciones que necesitamos, pero la más importante de ellas es la importación al espacio de nombres **System.Linq**; sin esa importación no podremos usar las extensiones de LINQ.

En los listados de ejemplo de cada una de estas cláusulas usaremos varias colecciones, una de ellas es la colección *artículos*, que vimos en el listado 6.1 del capítulo anterior, otra será el *array nums* del listado 6.7 (también en el capítulo anterior) y también el *array noms*, definido en el listado 7.1 de este capítulo.

```
var noms = new[] { "Pepe", "Juan", "Eva", "Pedro", "Luisa" };
```

Listado 7.1. Array para usar en los ejemplos de esta sección

select

Esta cláusula sirve para indicar el dato que se devolverá en cada elemento de la consulta resultante. Tal como vimos en el capítulo anterior, el valor indicado en la cláusula **select** puede ser un valor simple o uno representado por un tipo anónimo.

Además de crear tipos anónimos en la cláusula **select** (ver listado 6.12 del capítulo anterior) o devolver un dato que contenga algunas de las propiedades del objeto evaluado en la colección de datos (ver listado 6.4 del capítulo anterior), también podemos crear un tipo que utilice el nombre de la propiedad que queramos, una especie de alias, como ocurre cuando usamos **AS** en una sentencia **SELECT** de T-SQL en la que podemos indicar el nombre de las columnas devueltas en lugar de usar el nombre de la propiedad. Mejor lo vemos con un ejemplo. En el listado 7.2 utilizamos dos alias para las propiedades **Código** y **PrecioVenta** de cada uno de los artículos cuyo precio de venta sea inferior a 1.0. Como vemos en el bucle **foreach**, la colección resultante de esa expresión tendrá un tipo anónimo con las propiedades que hemos indicado en la cláusula **select**.

```
var res = from a in artículos
          where a.PrecioVenta < 1.0M
          select new
          {
              Cod = a.Código,
              PVP = a.PrecioVenta
          };

foreach(var a in res)
    Console.WriteLine("{0} {1}", a.Cod, a.PVP);
```

Listado 7.2. Podemos usar alias en los nombres de las propiedades devueltas en la cláusula select

Esta forma de indicar un alias es especialmente útil cuando utilizamos funciones de agregado, con idea de darle nombre a cada uno de los resultados producidos por esas funciones agregadas, ya que, como tendremos ocasión de ver en un momento, podemos utilizar los agregados como parte de una consulta, de forma que el valor de esa función forme parte del tipo de datos resultante como elemento de la colección devuelta.

from y join

Todas las expresiones de consulta de C# deben empezar con la cláusula **from** y produce una consulta de ejecución aplazada.

Después de **from** indicamos la variable que usaremos para acceder a cada uno de los elementos que vamos a consultar (variable de rango). Esta variable representa un elemento de la colección que indicamos después de la instrucción **in**. Como ya hemos tenido ocasión de comprobar en los ejemplos anteriores, el formato de esta cláusula es: **from variable in colección**.

Si necesitamos comprobar más de una colección, podemos usar varios **from** seguidos, esto sería similar a usar la sentencia **JOIN** en una consulta de SQL, que como veremos, en el lenguaje de consulta de LINQ también se puede usar de forma parecida por medio de la cláusula **join**, aunque con ciertas restricciones.

En el listado 7.3 vemos un ejemplo en el que recorremos dos *arrays*: el primero con los valores enteros que ya vimos en el listado 6.7 del capítulo anterior y el segundo con los nombres que definimos en el listado 7.1. La condición que ponemos (cláusula **where**) es que solo se incluyan los nombres que tengan la cantidad de caracteres de cualquiera de los valores numéricos. Cada elemento de

la colección producida por esta consulta tendrá el nombre (el valor de *s*) y el número de caracteres (el valor de *n*) y, como no indicamos ningún alias, se usan esas mismas variables como propiedades del elemento resultante (por eso en el bucle **foreach** usamos esos nombres de propiedades para mostrar los datos). En la figura 7.1 vemos el resultado de ejecutar ese código.

```
var res1 = from n in nums
           from s in noms
           where s.Length == n
           select new { n, s };

foreach(var v in res1)
    Console.WriteLine("{0}, {1}", v.s, v.n);
```

Listado 7.3. Dos sentencias from concatenadas

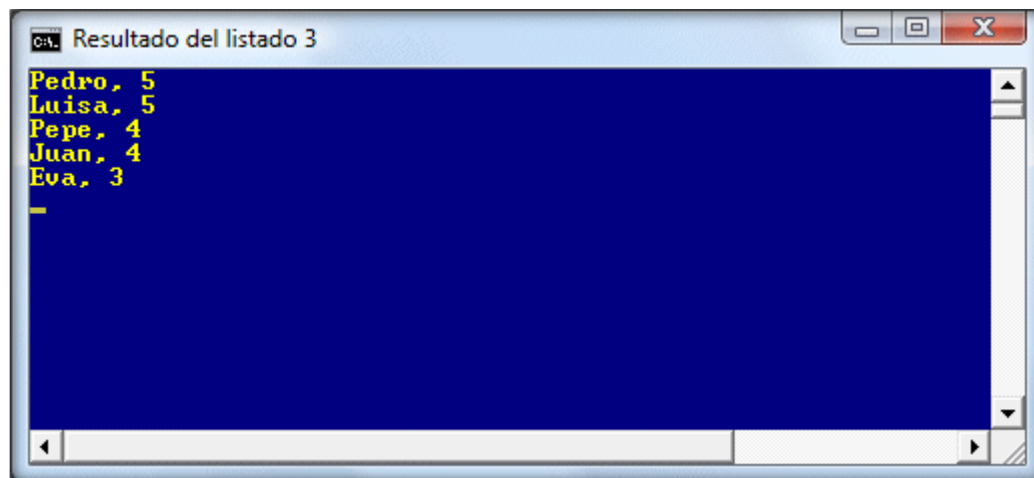


Figura 7.1. Resultado del listado 7.3

Cuando utilizamos varias cláusulas **from**, podemos realizar comprobaciones en cada una de ellas, esto es útil si queremos hacer algún filtro en cada una de las colecciones usadas en cada cláusula. Por ejemplo, en el listado 7.4 filtramos los números para que solo se tengan en cuenta los menores de 5.

```
var res2 = from n in nums where n < 5
           from s in noms
           where s.Length == n
           select new { Nombre = s, Len = n };

foreach(var v in res2)
    Console.WriteLine("{0}, {1}", v.Nombre, v.Len);
```

Listado 7.4. Cada sentencia from puede tener su propio filtro

Estos dos ejemplos los podemos cambiar para usar la cláusula **join**, de esa forma (sobre todo para los que estén acostumbrados a usar sentencias de T-SQL) resultará más fácil su lectura. En el listado

7.5, tenemos el código equivalente al del listado 7.3, en este caso, el resultado mostrado sería el mismo que el de la figura 7.1.

```
var res3 = from n in nums
           join s in noms
           on n equals s.Length
           select new { n, s };
```

Listado 7.5. Ejemplo usando join en lugar de dos from concatenados

Cuando comenté que utilizar **join** tenía limitaciones con respecto a dos **from** seguidos, es porque después de la instrucción **on** debemos indicar la comprobación que queremos hacer, y si esa comprobación es para comparar los elementos individuales de cada grupo de datos, solo podemos usar la instrucción **equals** para comparar la igualdad de las dos expresiones indicadas. Sin embargo, con los dos **from**, al utilizar **where**, podemos comparar lo que queramos, no solo la igualdad. Además, cuando usamos **equals**, los valores usados en la comparación de igualdad, deben contener las dos variables usadas en las colecciones que unimos, una de esas variables debemos usarla a la izquierda de esa instrucción y la otra a la derecha (aunque la documentación no lo aclara tácitamente, la variable de rango indicada después de **join** debemos usarla a la derecha).

El uso de **equals** es para hacer evidente que **join** sirve para hacer una combinación solo basada en la igualdad de dos claves, si necesitamos hacer otros tipos de comparaciones, la solución es usar varios **from** con sus correspondientes **where**.

Funciones de agregado

Las funciones de agregado devuelven un valor que es el resultado de realizar un cálculo en los valores indicados, normalmente en una colección o en un *array*, aunque, como veremos en próximos capítulos, los podremos aplicar a cualquier resultado de una consulta de LINQ.

A diferencia de Visual Basic, en C# no existe una cláusula específica para usar las funciones de agregado, por tanto, éstas debemos usarlas como las define el propio .NET Framework, es decir, como métodos extensores.

Las funciones de agregado definidas en Visual Basic, pero que desde C# podemos usar como métodos extensores de la interfaz **IEnumerable<T>**, son: **All**, **Any**, **Average**, **Count**, **LongCount**, **Max**, **Min** y **Sum**.

Nota

Al final de este capítulo veremos cómo definir un par de funciones de agregado para que nos devuelva la suma de todos los números pares (e impares) de una colección o array numérico.

Si el lector ha escrito alguna consulta de SQL, seguramente habrá utilizado algunas de esas funciones de agregado, particularmente la función **Count**, y si no ha trabajado nunca con SQL, seguramente habrá visto algún que otro código parecido al del listado 7.6, en el que nos devuelve el total de elementos que hay en la tabla **Cientes** cuyo campo **ID** sea mayor de 3.

```
SELECT COUNT(*) FROM Cientes WHERE ID > 3
```

Listado 7.6. Uso de la función COUNT en una consulta de T-SQL

Estas funciones de agregado las podemos aplicar al resultado de una consulta, en ese caso, dicha consulta se ejecutará inmediatamente para poder obtener el resultado del método extensor que representa a la función de agregado. En el listado 6.10 del capítulo anterior vimos cómo usar el método **Sum**, aunque haciendo que esa consulta se ejecutara inmediatamente y usando solo el resultado (es decir, no se podía reutilizar la consulta), eso mismo lo podríamos hacer a una variable que contenga una consulta de LINQ (con idea de poder reutilizar esa misma consulta para aplicarle otros métodos/funciones de agregado). Para aclararlo, veamos mejor un ejemplo. En el listado 7.7 realizamos una consulta similar a la que estamos viendo en los últimos ejemplos, y después usamos varias funciones de agregado en esa variable que contiene la consulta de LINQ.

```
var res = from n in nums
          where n > 4
          select n;

// Suma todos los valores que hay
// en el resultado de la consulta
var sum1 = res.Sum();
Console.WriteLine("Sum = {0}", sum1);

// Cuántos elementos hay
var num = res.Count();
Console.WriteLine("Count = {0}", num);

// La media de todos los valores
var num1 = res.Average();
Console.WriteLine("Average = {0}", num1);

// El mayor de los valores
var num2 = res.Max();
Console.WriteLine("Max = {0}", num2);

// El menor de los valores
var num3 = res.Min();
Console.WriteLine("Min = {0}", num3);
```

Listado 7.7. Varios ejemplos de funciones de agregado

A continuación vamos a ver algunos ejemplos en los que nos pueden ser de utilidad estas funciones de agregado, aunque recordemos que en C# esas funciones no están definidas directamente en el lenguaje, y por tanto debemos usarlas como métodos extensores aplicados a una consulta de LINQ.

Las funciones Average, Count, LongCount, Max, Min y Sum

Cuando usamos estas funciones, los cálculos se hacen con los elementos que devuelva la consulta LINQ (aunque también lo podemos aplicar a cualquier objeto que implemente **IEnumerable<T>**). Por ejemplo, **Sum** devuelve la suma de todos los valores, **Average** devuelve el valor medio (la media aritmética), **Count** nos dice cuantos elementos hay, **LongCount** también devuelve el total de elementos pero como valor **long (Int64)** en lugar de **int (Int32)**, **Max** y **Min** nos indican cuál es el valor máximo y mínimo respectivamente.

Todas estas funciones se pueden usar con o sin un argumento, si no indicamos el argumento, se evalúa cada uno de los elementos de la colección a la que se aplica el método. Si lo indicamos, siempre será como una expresión *lambda*, y el valor devuelto por esa expresión será el valor a tener en cuenta en la función de agregado. Veamos un ejemplo simple para entender mejor cómo trabajan estas funciones. En el listado 7.8 tenemos cuatro formas de usar la función **Sum**: las dos primeras son equivalentes, es decir, si no se indica un argumento, se evalúa cada uno de los valores contenidos en la colección, y la expresión *lambda* del segundo ejemplo simplemente devuelve cada uno de los elementos de la colección, por tanto hace exactamente lo mismo que la primera. En el tercer ejemplo lo que se suma es el valor indicado como argumento, pero lo que devuelve la expresión *lambda* no utiliza la variable de rango, por tanto, ese valor se sumará tantas veces como elementos tenga la colección, en este ejemplo, el resultado de la consulta (variable *res*) tiene 5 elementos. En el cuarto ejemplo, se suma el valor de cada elemento, pero añadiéndole 2. El resultado de ejecutar este código es de 35 para los dos primeros, 15 para el tercero (3 multiplicado tantas veces como elementos tiene la colección) y en el último, devolverá 45 ($35 + (5 * 2)$).

Como ya vimos antes, estas funciones de agregado las podemos usar tanto con el resultado de una consulta o aplicándola directamente a la consulta, por ejemplo, usando el código del listado 7.9 obtenemos el mismo valor que en las dos primeras líneas del listado anterior.

```
var res = from n in nums
          where n > 4
          select n;

// Las dos primeras son equivalentes
var sum1 = res.Sum();

var sum2 = res.Sum(n => n);

// Se sumará 3 tantas veces como elementos haya en la colección
var sum3 = res.Sum(n => 3);

// Se le suma 2 al valor analizado y se devuelve el total
var sum4 = res.Sum(n => n + 2);
```

Listado 7.8. El valor devuelto por la expresión lambda del parámetro indicado es el que se sumará para cada uno de los elementos que tenga la colección

```
var sum5 = (from n in nums where n > 4 select n).Sum();
```

Listado 7.9. Podemos usar una función de agregado directamente en una consulta

Si nos decidimos por aplicar las funciones de agregado a una consulta directa, también podemos pasarle una expresión *lambda* como argumento. El código del listado 7.10 es equivalente a la última línea del listado 7.8.

```
var sum6 = (from n in nums where n > 4 select n).Sum(n => n + 2);
```

Listado 7.10. Podemos usar argumentos al aplicar una función de agregado a una expresión de consulta, pero indicando la expresión lambda a usar para realizar la operación

El resto de las funciones de agregado son parecidas a ésta que acabamos de ver, solo que realizando operaciones diferentes. En el listado 7.11 vemos cómo usar esas funciones; en los comentarios se indican los valores que devuelve cada una de ellas.

Las funciones **Count** y **LongCount** son un caso aparte, ya que podemos indicar como argumento una condición a evaluar. Por ejemplo, si queremos que solo se cuenten los elementos que sean mayor de tres, lo haríamos como en el listado 7.12 (recordemos que la diferencia entre esas dos funciones es el tipo de datos que devuelven).

```
// Calculamos la media, el resultado es 5 (45 / 9)
var res1 = (from n in nums select n).Average();

// El total de elementos (9)
var res2 = (from n in nums select n).Count();

// El valor mayor (9)
var res3 = (from n in nums select n).Max();

// El valor menor (1)
var res4 = (from n in nums select n).Min();
```

Listado 7.11. Ejemplo de las funciones de agregado

```
// Total de elementos mayores de 3 (6)
var res5 = (from n in nums select n).Count(n => n > 3);
```

Listado 7.12. El total de elementos en la colección que sean mayores de 3

Tal como vimos en el listado 7.8, si no indicamos un argumento en las funciones, es como si le pasáramos la variable usada para recorrer los elementos (salvo en las funciones **Count**, que se utiliza para hacer una comprobación extra). En estos ejemplos ese valor es un “simple” número con el que operar, pero si el elemento que contiene la colección es una clase o una estructura, tendremos que indicar con qué propiedad debe realizar la operación. Por ejemplo, si usamos la colección **artículos**, podríamos hacer los cálculos en la propiedad **PrecioVenta**. El código del listado 7.13 muestra esas mismas funciones utilizando elementos del tipo **Artículo**. Las funciones **Count** simplemente cuentan los elementos, pero también podemos indicar una condición de los elementos que se deben contabilizar, en este ejemplo, se evalúan todos los artículos cuyo precio sea menor de 0.99.

```
var res0 = (from a in artículos select a).Count();
var res1 = (from a in artículos select a).Count(n => n.PrecioVenta < 0.99M);
var res2 = (from a in artículos select a).Average(n => n.PrecioVenta);
var res3 = (from a in artículos select a).Max(n => n.PrecioVenta);
var res4 = (from a in artículos select a).Min(n => n.PrecioVenta);
var res5 = (from a in artículos select a).Sum(n => n.PrecioVenta);
```

Listado 7.13. Funciones de agregado que utilizan una propiedad de una clase

En la expresión *lambda* usada como argumento de la función de agregado, el tipo de datos sobre el que tenemos que operar siempre será sobre uno de los elementos usados como resultado de la consulta, es decir, el tipo de datos indicado después de la cláusula **select**, tal como vemos en el listado 7.14.

```
var res6 = (from a in artículos
            select new { PVP = a.PrecioVenta }
            ).Max(n => n.PVP);
```

Listado 7.14. El elemento a usar en la expresión lambda de las funciones de agregado siempre será del tipo contenido en la colección

Las funciones All y Any

Estas dos funciones devuelven un valor de tipo **bool** en lugar de un valor numérico.

La función **All** comprueba que *todos* los elementos de la colección cumplan la condición que indiquemos como argumento. Si alguno de los elementos no cumple esa condición devolverá un valor **false**, por tanto solo devolverá **true** si **todos** los elementos cumplen esa condición (también devuelve verdadero si la colección está vacía).

Por otra parte, **Any** devuelve **true** si *alguno* de los elementos cumple la condición indicada. Si no indicamos ninguna condición, podemos usar esta función para saber si la colección tiene algún elemento, ya que en ese caso, devolverá **true**. ¿Cuándo devuelve un valor falso? Cuando ninguno de los elementos cumple la condición o la colección está vacía.

En el listado 7.15 vemos un ejemplo de estas dos funciones en las que usamos como condición que cada uno de los elementos sea mayor de 3.

```
// true si todos son mayor de 3
var res1 = (from n in nums select n).All(n => n > 3);
// true si alguno es mayor de 3
var res2 = (from n in nums select n).Any(n => n > 3);
```

Listado 7.15. Las funciones All y Any siempre devuelven un valor de tipo bool

Si el contenido de la colección es una clase, la evaluación la haremos sobre alguna de las propiedades (incluso podemos comprobar más de una condición, utilizando cualquier operador de comparación). En los listados 7.16 y 7.17 vemos cómo evaluar el precio de venta y hacer más de una com-

probación. En los dos últimos casos, se deben cumplir las dos condiciones, ya que usamos el operador **&&** (AND), también podríamos usar **&**, pero **&&** tiene mejor rendimiento, ya que solo evalúa el segundo operando si el primero cumple la condición indicada, al igual que ocurre con **||** (OR) frente a **|**.

```
// Devuelve false, ya que no todos los precios son mayores de 0.75
var res3 = (from a in artículos
            select a
            ).All(n => n.PrecioVenta > 0.75M);

// Devuelve true, porque algunos precios son menores de 0.95
var res4 = (from a in artículos
            select a
            ).Any(n => n.PrecioVenta < 0.95M);
```

Listado 7.16. Si los elementos son clases, tendremos que evaluar alguna propiedad

```
// true si todos tienen un precio mayor de 0.15 y el IVA es 16
var res5 = (from a in artículos
            select a
            ).All(n => n.PrecioVenta > 0.15M && n.IVA == 16M);

// true si alguno tiene un precio menor de 0.95 y el IVA es 16
var res6 = (from a in artículos
            select a
            ).Any(n => n.PrecioVenta < 0.95M && n.IVA == 16M);
```

Listado 7.17. Podemos usar más de una condición

Filtrar las consultas con where

La cláusula **where** la usaremos para filtrar las consultas, es decir, para indicar qué condición queremos poner para extraer de la colección de datos los que realmente nos interesen, solo se incluirán en el resultado de la consulta los elementos que devuelvan **true** a la expresión indicada como condición.

Esta instrucción la usaremos en las consultas de LINQ, y para hacer el filtro tendremos que indicar una condición, que puede ser simple o múltiple (varias condiciones usando los operadores condicionales: **&**, **&&**, **|**, **||**, **!**, etc.)

En el listado 7.18 vemos un ejemplo de cómo filtrar una consulta utilizando funciones de agregado. En estos ejemplos primero se hace el filtro con la cláusula **where** (los artículos que tengan un valor 16 en la propiedad **IVA**) y después se aplica la función de agregado con la condición a tener en cuenta, por tanto esas comprobaciones solo se harán sobre el contenido de la consulta, pero una vez filtrado por la condición indicada en **where**. Para verlo más claro, en el listado 7.19 se utilizan las mismas funciones de agregado, pero aplicadas a una variable con una consulta de LINQ.

```
// Para que no todos tengan 16 en el IVA
artículos[0].IVA = 7M;

// Si todos los artículos con el IVA 16
// tienen el precio de venta superior a 0.15
var res1 = (from a in artículos
            where a.IVA == 16M
            select a).All(n => n.PrecioVenta > 0.15M);

// Si hay artículos cuyo IVA sea 16
// y el precio de venta sea menor de 0.95
var res2 = (from a in artículos
            where a.IVA == 16M
            select a).Any(n => n.PrecioVenta < 0.95M);

// La suma total del precio venta de los artículos
// cuyo IVA sea 16
var res3 = (from a in artículos
            where a.IVA == 16M
            select a).Sum(n => n.PrecioVenta);
```

Listado 7.18. Podemos filtrar los datos y utilizar funciones de agregado

```
// Si las funciones de agregado las utilizaremos
// sobre el mismo conjunto de datos,
// podemos crear primero la consulta y
// aplicar posteriormente las funciones
var res = from a in artículos
          where a.IVA == 16M
          select a;

var res4 = res.All(n => n.PrecioVenta > 0.15M);
var res5 = res.Any(n => n.PrecioVenta < 0.95M);
var res6 = res.Sum(n => n.PrecioVenta);
```

Listado 7.19. Esas funciones las podemos aplicar directamente a una consulta existente

En la cláusula **where** también podemos hacer un filtro con más de una condición, por supuesto con valores que estén relacionados con los elementos analizados (condiciones múltiples). En el listado 7.20 añadimos un nuevo artículo a la colección *artículos*, no es que yo compre la leche a ese precio, pero es para que veamos una pequeña “trampa” o error que muchos solemos cometer cuando utilizamos los operadores lógicos AND y OR (&& y ||). Las condiciones de filtrado para esta consulta son: que el precio de venta sea superior a 0.45 y además, que el código empiece por cero o por tres, cualquiera de esos dos nos vale, pero de forma que la condición sea excluyente, es decir, si el precio **no** es superior a 0.45, da igual qué código tenga el artículo; por eso he puesto entre paréntesis las dos condiciones del carácter inicial de la propiedad *Código*, ya que si las condiciones las encerramos entre paréntesis, serán agrupadas como un solo resultado, antes de evaluar otras condiciones que puedan existir por fuera.

```
artículos.Add(  
    new Artículo("30")  
    {  
        Descripción = "Leche entera brick",  
        IVA = 7M,  
        PrecioVenta = 0.1M  
    });  
  
var res1 = from a in artículos  
           where a.PrecioVenta > 0.45M  
              && (a.Código.StartsWith("0")  
                 || a.Código.StartsWith("3"))  
           select new {  
               ID = a.Descripción,  
               IVA = a.IVA,  
               PVP = a.PrecioVenta  
           };  
  
foreach(var a in res1)  
    Console.WriteLine("{0}, {1}, {2}",  
        a.ID, a.IVA, a.PVP);
```

Listado 7.20. Una consulta con varias condiciones

Con esas condiciones, no se debe mostrar el artículo que hemos añadido al principio del listado 7.20, ya que, aunque el código empieza por 3, el precio es inferior al indicado después de **where**. Sin embargo, si quitamos los paréntesis que encierran las dos condiciones que hay después del operador **&&** (ver el listado 7.21), la condición cambia; en ese caso, lo que le indicamos al motor de ejecución es que queremos todos los artículos que tengan ese valor en **PrecioVenta** y que el código empiece por cero, o bien que el código empiece por 3. Pero la condición del precio de venta solo es aplicable a los artículos que tengan un cero al inicio del código, debido a que el orden de evaluación de las condiciones es de izquierda a derecha, por tanto, también se mostrará el artículo que añadimos al principio del listado 7.20 aunque el contenido de la propiedad **PrecioVenta** sea inferior al indicado después de **where**.

```
var res2 = from a in artículos  
           where a.PrecioVenta > 0.45M  
              && a.Código.StartsWith("0")  
              || a.Código.StartsWith("3")  
           select new {  
               ID = a.Descripción,  
               IVA = a.IVA,  
               PVP = a.PrecioVenta  
           };
```

Listado 7.21. La misma consulta que en el listado 7.20, pero sin los paréntesis

Así es **&&** (incluso **&**) que se une a la condición que sigue, precisamente porque significa “y”, por tanto, solo da por buena la condición “*si esto y esto otro*” se cumple. Sí, ya sé que esto es cosa de principiantes, pero no nos confiemos demasiado, que cuando empezamos a complicar las condiciones con operadores lógicos, al final acabamos totalmente desconcertados. Y para ver qué tal estamos en esto de la lógica, vamos a añadir un nuevo artículo tal como se muestra en el código del listado 7.22, con idea de hacer un pequeño ejercicio (o triple ejercicio, ya que se compone de tres propuestas):

1. Qué condición (o condiciones) tendríamos que usar para incluir todos los artículos cuyos precios sean mayores de 0.75 o menores de 0.95, es decir, que estén comprendidos entre 0.75 y 0.95 (ambos inclusive), y además queremos todos los artículos que la propiedad **IVA** tenga un valor 7, independientemente del precio que tengan.
2. Como en este ejercicio tendremos en cuenta dos cosas diferentes (los valores de la propiedad **PrecioVenta** y de **IVA**), hacer dos consultas: una en la que primero comprobemos los precios y después el **IVA**, y otra al revés: primero evaluamos el valor de la propiedad **IVA** y después los precios.
3. ¿Cuál de las dos es más eficiente? Explicar por qué.

```
artículos.Add(  
    new Artículo("31")  
    {  
        Descripción = "Leche semi desnatada brick",  
        IVA = 7M,  
        PrecioVenta = 0.85M  
    });
```

Listado 7.22. Añadimos un nuevo artículo para el ejercicio propuesto

Las respuestas al final del capítulo. Como pista, decir que los dos artículos añadidos en los listados 7.20 y 7.22 estarán en la colección devuelta por la consulta (ver la figura 7.2 con los artículos que se mostrarán).

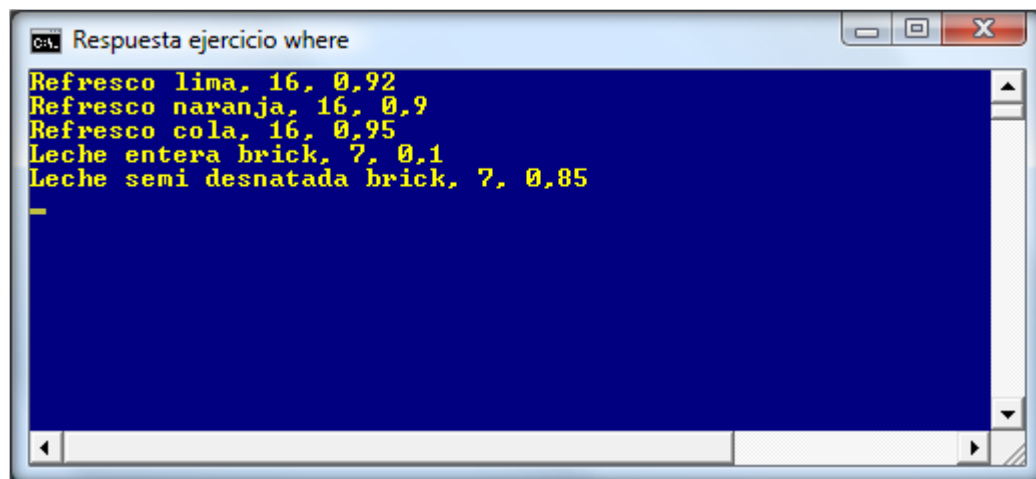


Figura 7.2. Los valores que se mostrarán con el código del ejercicio

Distinct

En C#, a diferencia de Visual Basic, no existe una instrucción (o cláusula) para extraer de una colección solo los elementos que son distintos, pero existe como un método extensor de la interfaz **IEnumerable<T>**, por tanto, podemos aplicarlo a cualquier consulta de LINQ.

Este método comprobará los elementos de la colección en la que se utiliza y devolverá solo aquellos elementos que son diferentes, usando para ello la comparación de igualdad predeterminada del tipo de datos al que se aplica.

En el listado 7.23 tenemos un *array* de números enteros en el que algunos de los valores están repetidos, por tanto la colección resultante (asignada a la variable *dis1*) solo contendrá aquellos valores que sean distintos.

```
var nums2 = new[] { 1, 3, 4, 2, 5, 3, 7, 6, 5 };
var dis1 = nums2.Distinct();

Console.WriteLine("Los valores usando Distinct");
foreach(var n in dis1)
    Console.WriteLine(n);
```

Listado 7.23. Los valores que son únicos en el array

Sí, esto es claro, incluso si ese método lo aplicamos a una colección de facturas en la que queremos obtener todos los códigos de los clientes que hayan realizado alguna compra, tal como vemos en el listado 7.24. Pero si en la cláusula **select** indicamos algo más que el código del cliente (ver listado 7.25), ¿qué será lo que se evalúe? (recordemos que la cláusula **select** creará un tipo anónimo). La respuesta es la esperada: todos los elementos que tengan datos diferentes en las propiedades de ese tipo anónimo, es decir, lo que suponíamos, ya que en los tipos anónimos las comparaciones de los contenidos se realizan de la misma forma que en los tipos de datos normales (solo era una especie de desconcierto para ver si el lector se confundía). Lo que se hace en el código del listado 7.25 es simplemente sacar todas las facturas que haya, ya que se comprueba que tanto el número como el cliente deben ser distintos y, salvo error u omisión, nunca deben existir dos facturas que tengan el mismo número, por tanto, al menos en este caso concreto, esa cláusula **Distinct** no nos sirve de mucho.

```
var res1 = (from f in facturas select f.Cliente).Distinct();
foreach(var c in res1)
    Console.WriteLine("{0} {1}", c.Codigo, c.Nombre);
```

Listado 7.24. Los diferentes clientes incluidos en la colección facturas

```
var res2 = (from f in facturas
            select new { f.Numero, f.Cliente }).Distinct();

foreach(var c in res2)
    Console.WriteLine("{0} {1}", c.Numero, c.Cliente.Codigo);
```

Listado 7.25. Esta consulta debería generar todos los datos (salvo que haya números de facturas repetidos)

Pero supongamos que tenemos una serie de facturas y queremos saber qué clientes han realizado compras en una fecha determinada, o para ser más exactos, queremos saber todos los clientes que han realizado compras en todas las fechas que contiene esa colección, pero sin duplicidades, es decir, si un mismo cliente ha hecho varias compras en un mismo día, solo aparecerá una vez. En este caso, y sabiendo que **Distinct** opera sobre el tipo de datos que contiene la colección resultante de la consulta, podríamos decidir crear un tipo anónimo que resulte de la fecha y el código del cliente de cada factura. En el listado 7.26 vemos un ejemplo de esa consulta de LINQ, en la que usamos el método **ToString** para tener en cuenta solo los datos de la fecha y no de la hora, ya que se supone que cada factura se hará en una hora diferente (minutos y segundos) de cualquier otra que pueda tener el mismo cliente en ese mismo día. Si no comprobáramos solo la fecha, en el resultado se incluirían todas las facturas, ya que es poco probable que el mismo cliente tuviese todas sus facturas en la misma hora, minuto y segundo, por tanto, la única forma de saber qué clientes distintos han facturado en una fecha es comprobando solo el día, el mes y el año.

```
var res3 =(from f in facturas
            select new {
                f.Numero,
                f.Fecha.ToString("dd/MM/yyyy")
            }).Distinct();
```

Listado 7.26. Consulta errónea para obtener las facturas distintas de cada cliente por fecha

El problema del código del listado 7.26 es que no compilará. Tal como vemos en la figura 7.3 lo que obtenemos es un error indicándonos que no podemos usar el resultado de un método como propiedad del tipo anónimo. La solución es sencilla, simplemente tendremos que crear nombres de propiedades asignadas por nosotros. En el listado 7.27 vemos cómo solucionarlo.

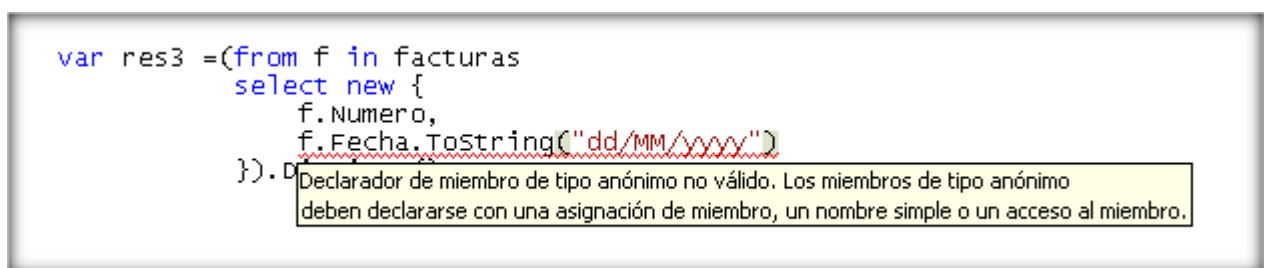


Figura 7.3. Los nombres de los campos no se pueden inferir desde un método

```
var res3 =(from f in facturas
           select new {
               Numero = f.Numero,
               Fecha = f.Fecha.ToString("dd/MM/yyyy")
           }).Distinct();
```

Listado 7.27.Las facturas distintas de cada cliente por fecha

En este ejemplo, el nombre de la propiedad solo sería necesario usarlo para contener la fecha, ya que es el campo o propiedad cuyo nombre no se podría generar automáticamente, por tanto, el código del listado 7.28 producirá un tipo anónimo con las mismas propiedades que el código del listado 7.27.

```
var res4 =(from f in facturas
           select new {
               f.Numero,
               Fecha = f.Fecha.ToString("dd/MM/yyyy")
           }).Distinct();
```

Listado 7.28.Código equivalente al del listado 7.27

Indicar la ordenación de los elementos de una consulta

Si queremos que los elementos resultantes de una consulta LINQ estén ordenados por algún campo en particular, utilizaremos la cláusula **orderby**, indicando a continuación el campo o campos por los que queremos clasificar, además de poder indicar si esa ordenación la queremos en modo ascendente o descendente.

Por ejemplo, si queremos mostrar los artículos que tengan un precio inferior a 0.95, podemos hacer que se clasifiquen por la descripción usando el código del listado 7.29.

```
var res1 = from a in artículos
           where a.PrecioVenta < 0.95M
           orderby a.Descripción
           select a;
```

Listado 7.29. Consulta LINQ que devuelve los elementos ordenados por la propiedad Descripción

Si no indicamos cómo queremos que se ordenen los elementos, el orden será ascendente y equivaldría a agregar la cláusula **ascending**. Si queremos que esa ordenación sea descendente, tendremos que usar **descending**, tal como vemos en el listado 7.30.

```
var res2 = from a in artículos
           orderby a.Descripción descending
           select a;
```

Listado 7.30. Esta consulta devuelve los elementos ordenados de mayor a menor

Si queremos que la clasificación sea por varios campos, debemos indicarlos después de **orderby**, separando cada uno de ellos por una coma. Además, en cada campo podemos indicar cómo queremos que se clasifiquen, por ejemplo, en el código del listado 7.31 se clasifican los clientes de forma ascendente por el país y de forma descendente por la empresa.

```
var res3 = from c in clientes
           orderby c.País ascending,
                  c.Empresa descending
           select c;
```

Listado 7.31. Podemos ordenar por varios campos

Cuando indicamos varios campos después de **orderby**, el orden de clasificación siempre será de izquierda a derecha. En el código del listado 7.31, primero se clasifican por la propiedad **País** y después por la propiedad **Empresa**, por tanto, se mostrarán todos los datos de un mismo país juntos, pero clasificados por el nombre de la empresa.

Si no se indica el modo de ordenación, siempre se usará **ascending**, incluso cuando haya varios campos y algunos de ellos se ordenen de forma descendente. Es decir, si no se indica cómo se ordenarán, siempre será ascendente, por tanto, no se tendrá en cuenta el último criterio utilizado. Para clarificarlo más, cada campo indicado debe tener su propio criterio de ordenación y si no se indica, será ascendente. En el listado 7.32 tenemos un ejemplo con varios campos y varios criterios.

```
var res4 = from a in artículos
           orderby a.Código descending,
                  a.Descripción,
                  a.PrecioVenta descending
           select a;
```

Listado 7.32. Varios criterios de ordenación

En C#, la cláusula **orderby** siempre hay que usarla antes de **select** (en Visual Basic podemos usarla antes o después).

Invertir los datos con Reverse

Si queremos invertir el orden de los elementos de una colección, podemos usar el método extensor **Reverse**, éste método lo aplicaremos a cualquier colección que implemente **IEnumerable<T>**, por tanto lo podemos aplicar a cualquier *array* o colección, como puede ser el resultado de una consulta. En el código del listado 7.33 invertimos la ordenación realizada en la consulta del listado 7.32.

```
var res5 = res4.Reverse();
```

Listado 7.33. El método extensor Reverse invierte el orden del contenido de una colección

Agrupaciones y combinaciones: group by y join into

Estas dos cláusulas se pueden usar para agrupar los datos de una consulta. Por un lado, **group by** nos permite crear grupos basados en la colección que indicamos en la consulta, mientras que **join into** nos permitirá combinar dos colecciones basadas en las claves que coincidan de éstas.

Para ser precisos, **join into** no agrupa datos, sino que los combina, al igual que hace **join**. De hecho, si buscamos equivalencias entre estas cláusulas de C# y las instrucciones de una consulta de T-SQL, la cláusula **join** equivaldría a **INNER JOIN**, mientras que **join into** se puede decir que equivale a **LEFT JOIN**.

Estas dos cláusulas que utiliza C# para las consultas de LINQ equivalen a los métodos extensores **GroupBy** y **GroupJoin**.

Veamos por separado cómo podemos utilizarlas en nuestro código.

group by (GroupBy)

Por ejemplo, tenemos una serie de facturas y queremos agrupar las de cada cliente, de forma que el resultado de la consulta tenga cada uno de esos clientes y un grupo con las facturas que le corresponden. Esto es lo que hace el código del listado 7.34.

```
var res1 = from f in facturas
           group f by f.Cliente;
```

Listado 7.34. Las facturas agrupadas por cada uno de los clientes

Cuando usamos la cláusula **group by** el resultado de la consulta es una colección del tipo **IGrouping<TKey, TElement>** donde **TKey** es el tipo de datos por el que se agrupa (el indicado después de **by**) y **TElement** es una colección con cada uno de los elementos que estamos agrupando (el indicado después de **group**). En nuestro ejemplo la clave de cada elemento de la consulta resultante es un objeto del tipo **Cliente** (el tipo que devuelva la propiedad **f.Cliente**) y está representado por la propiedad **Key**. Por otro lado, los elementos de cada colección son las facturas que corresponden a ese cliente. Por tanto, para mostrar los datos debemos usar dos bucles anidados, tal como vemos en el listado 7.35.

```
foreach(var f in res1)
{
    Console.WriteLine("{0} {1}",
        f.Key.Empresa, f.Key.País);
    foreach(var f1 in f)
    {
        Console.WriteLine(" {0} {1}",
            f1.Numero, f1.Fecha);
    }
}
```

Listado 7.35. Mostrar los datos de la consulta del listado 7.34

Como vemos en el código del listado 7.34, no estamos usando una cláusula **select**, esto es así porque las consultas de LINQ en C# deben acabar con **select** o **group by**.

Si quisiéramos hacer algún tipo de filtro, por ejemplo, ordenarlos, etc., podemos hacerlo después de **group by**, en ese caso, agruparíamos los datos usando **into** seguido de la variable que contendrá la agrupación, pero en este caso sí que tendríamos que agregar una cláusula **select** (o bien usar otra instrucción **group by**). Un ejemplo de esto lo vemos en el listado 7.36, en el que ordenamos los elementos por el país de cada cliente. Para mostrar estos datos, podemos usar un código similar al mostrado en el listado 7.35.

```
var res2 = from f in facturas
            group f by f.Cliente into g
            orderby g.Key.País
            select g;
```

Listado 7.36. Podemos filtrar la agrupación realizada

Si necesitamos hacer alguna comprobación extra, por ejemplo, solo tener en cuenta los datos de las facturas que contengan más de 3 artículos, podemos utilizar los datos de la agrupación para realizar otra consulta y filtrar los datos que nos interesen. Eso lo vemos en el listado 7.37 en el que solo tenemos en cuenta las facturas con más de tres artículos, ordenamos los datos por el contenido de la propiedad **País** del cliente y además generamos un tipo anónimo con nombres de propiedades que nos den una pista más clara sobre el contenido, aunque en este caso, no sería necesario crear la propiedad **Cli**, ya que la variable con los datos (**g**) sigue siendo del tipo **IGrouping<K, E>**, por tanto define una propiedad llamada **Key** que hace referencia al cliente al que pertenecen esas facturas, pero este “follón” es más bien para que veamos que podemos complicarnos la vida de muchas formas, y como siempre, de nosotros depende que finalmente consigamos lo que queremos, algo que solo podremos hacer si tenemos las cosas claras y sabemos qué podemos hacer.

Para mostrar los datos de la consulta del listado 7.37, podemos hacerlo tal como vemos en el listado 7.38.

```
var res3 = from f in facturas
            group f by f.Cliente into g

            from fv in g
            where fv.Articulos.Count > 3
            orderby g.Key.País descending
            select new
            {
                Cli = g.Key,
                Facts = g
            };
```

Listado 7.37. También podemos hacer una sub consulta en el resultado de la agrupación

```
foreach(var f in res3)
{
    // Estos dos datos mostrarán lo mismo
    Console.WriteLine(f.Facts.Key);
    Console.WriteLine(f.Cli);

    foreach(var f1 in f.Facts)
    {
        Console.WriteLine(" {0} {1}",
            f1.Numero, f1.Fecha);
    }
}
```

Listado 7.38. Mostrar los datos de la consulta del listado 7.37

join into (GroupJoin)

La cláusula **join into** nos permite agrupar dos colecciones en una sola, pero siempre usando una comparación de igualdad entre dos de los elementos de esas colecciones (ver lo comentado anteriormente sobre **join**). Por ejemplo, si queremos todas las facturas de todos los clientes, agrupadas por cliente, usando **join into** podríamos hacerlo tal como vemos en el listado 7.39.

```
var res1 = from c in clientes
            join f in facturas
            on c.Codigo equals f.Cliente.Codigo
            into g
            select new
            {
                Cli = c,
                Facts = g,
                TotalArts = g.Sum(n => n.Articulos.Count)
            };
```

Listado 7.39. Una consulta usando join into

Debido a que usamos **join**, la comparación debe ser de igualdad y siempre deben intervenir una propiedad de la variable usada en la colección a la izquierda de **join** (en este ejemplo la colección *clientes*) y otra en la colección indicada a la derecha (en este ejemplo, la colección *facturas*). El grupo generado será el de las facturas del cliente analizado, y en la cláusula **select** creamos un tipo de datos anónimo en el que incluimos el cliente, las facturas de ese cliente y por medio de la función de agregado **Sum**, obtenemos la suma total de todos los artículos de todas las facturas de ese cliente. Para mostrar los datos obtenidos en esta consulta, podemos usar el código del listado 7.40.

```
foreach(var f in res1)
{
    Console.WriteLine("{0}, {1}", f.Cli, f.TotalArts);

    foreach(var fv in f.Facts)
        Console.WriteLine(" {0}", fv);
}
```

Listado 7.40. Mostrar los datos de la consulta obtenida en el listado 7.39

Como ya comenté al principio de esta sección, las instrucciones usadas por C# para realizar estas acciones se equivalen con métodos extensores, y para que veamos qué código nos ahorramos de escribir, además de que comprobaremos que usar las instrucciones es más fácil de leer que usar los métodos equivalentes.

En el listado 7.41 tenemos el equivalente de la consulta **join into** del listado 7.39 y en el listado 7.42 vemos el código equivalente del listado 7.36, pero usando el método **GroupBy** en lugar de la cláusula **group by**.

Pero si queremos saber lo que nos ahorramos de verdad al usar las instrucciones propias del lenguaje, podemos comparar los listados 7.37 (usando **group by, where, orderby**, etc.) y el correspondiente a los métodos extensores del listado 7.43 (recordemos que el compilador convertirá las cláusulas en llamadas a los métodos extensores equivalentes, por tanto, la mejor forma de saber qué es lo que el compilador genera, es inspeccionando el ejecutable resultante; que es casi lo que yo he hecho en este caso concreto, ya que no estaba dispuesto a perder la cabeza para conseguir el código equivalente).

```
var res2 = clientes.GroupJoin(
    facturas,
    c => c.Codigo,
    f => f.Cliente.Codigo,
    (c, f) => new
    {
        Cli = c,
        Facts = f,
        TotalArts = f.Sum(n => n.Articulos.Count)
    });
```

Listado 7.41. Una consulta usando el método extensor GroupJoin (equivale al listado 7.39)

```
var res4 = facturas.GroupBy(f => f.Cliente).OrderBy(g => g.Key.País);
```

Listado 7.42. Una consulta usando el método extensor GroupBy (equivale al listado 7.36)

```
var res5 = facturas.
    GroupBy(f => f.Cliente).
    SelectMany(g => g, (g, fv) => new { g, fv }).
    where(f => f.fv.Articulos.Count > 3).
    OrderByDescending(c => c.g.Key.País).
    Select(c => new
    {
        Cli = c.g.Key,
        Facts = c.g
    });
```

Listado 7.43. Una consulta usando el método extensor GroupBy (equivale al listado 7.37)

Divide y vencerás: Skip y Take

En C# no existen cláusulas específicas para estas operaciones de partición, por tanto debemos usar los métodos extensores definidos por .NET Framework.

Si de una consulta de LINQ queremos tomar solo una parte de la misma, utilizaremos las funciones **Skip** y **Take**. Como argumento de esas funciones indicaremos un número de elementos; la primera se “saltará” esos elementos, mientras que la segunda “tomará” esa cantidad de elementos.

Por ejemplo, en el listado 7.44 vemos cómo usar **Skip** para saltar los 3 primeros elementos de la colección *artículos*, es decir, incluirá en el resultado desde el cuarto artículo hasta el final.

```
var res1 = (from a in artículos select a).Skip(3);
```

Listado 7.44. Skip se saltará los elementos que le indiquemos

En el listado 7.45 vemos cómo usar **Take** para tomar solo los tres primeros elementos de la colección.

```
var res2 = (from a in artículos select a).Take(3);
```

Listado 7.45. Take tomará el número de elementos que indiquemos a continuación

También podemos utilizar estas dos funciones para que se salte un número de elementos y a partir de esa posición tome los que indiquemos. Por ejemplo, en el listado 7.46 tomamos 2 elementos a partir del tercero (saltamos dos y tomamos dos).

```
var res3 = (from a in artículos select a).Skip(2).Take(2);
```

Listado 7.46. Podemos usar Skip y Take en la misma consulta

Como es de suponer, estas funciones las podemos usar tanto en consultas directas (como en estos ejemplos) o bien en consultas que previamente se han asignado a una variable. En el código del listado 7.47 vemos cómo realizar las tres operaciones mostradas anteriormente, pero usando una variable a la que previamente le hemos asignado la consulta en la que filtramos solo los artículos que cumplan cierta condición y estén ordenados por la descripción.

```
var res = from a in artículos
          where a.IVA == 16M
          orderby a.Descripción
          select a;

var res01 = res.Skip(3);
var res02 = res.Take(3);
var res03 = res.Skip(2).Take(2);
```

Listado 7.47. Las funciones de partición las podemos aplicar a variables con consultas LINQ

Aunque en los primeros ejemplos de esta sección, al no realizar ningún tipo de filtro, podíamos haber usado directamente la colección artículos (ver listado 7.48), pero lo importante aquí no es saber a qué colección se puede aplicar, si no cómo aplicar estas funciones, que como vemos, en este caso concreto es bien simple.

```
var res4 = artículos.Skip(2).Take(3);
```

Listado 7.48. Las funciones de partición las podemos aplicar a cualquier colección de datos

Estas dos funciones las podemos usar para realizar paginaciones de datos, por ejemplo, en el listado 7.49 vemos cómo utilizar estas instrucciones para ir mostrando de 5 en 5 las facturas que tenemos. En cada repetición del bucle usamos un valor para indicar cuantos elementos debemos saltar y lo indicaremos después de **Skip**. Inicialmente saltamos cero elementos, pero después vamos saltando los que vayamos acumulando, es decir, los que ya hemos mostrado. Por otro lado, **Take** siempre tomará el número que queremos mostrar en la página. Como es de suponer, si quedan menos artículos que los indicados en **Take**, solo se devolverán los que resten a partir de la posición indicada por **Skip**.

```
var actual = 0;
var total = 5;
var pg = 0;

while(true)
{
    var res1 = (from f in facturas
                select f).Skip(actual).Take(total);

    // Si no hay datos, salir
    if(res1.Count() == 0)
        break;

    Console.WriteLine("Página {0}", ++pg);

    foreach(var f in res1)
    {
        Console.WriteLine("{0}, {1} {2:dd/MM/yyyy}",
            f.Numero, f.Cliente.Empresa, f.Fecha);
    }
    Console.WriteLine();
    actual += total;
};
```

Listado 7.49. Podemos usar Skip y Take para realizar paginaciones

SkipWhile y TakeWhile

Además de estas dos funciones, existen otras dos en las que podemos indicar una condición para saltar o tomar los elementos mientras se cumpla dicha condición (en Visual Basic sería equivalente a añadir a continuación de las cláusulas la instrucción **While**, pero como ya he comentado, en C# no

existen como instrucciones propias del lenguaje, por tanto, debemos usar métodos extensores para realizar estas operaciones).

Cuando utilizamos **SkipWhile** o **TakeWhile** se realizarán las mismas operaciones saltar o tomar, pero en lugar de indicar un número de elementos, éstos se evaluarán según la condición que pongamos en la expresión *lambda* que indicaremos como argumento de las funciones. Pero debemos tener en cuenta que en cuanto la condición deje de cumplirse, se incluirán los elementos que queden. Para entenderlo mejor, debemos tener en cuenta que se hace algo como: saltar (o tomar) mientras se cumpla la condición, y en cuanto deje de cumplirse, todo lo que haya también se incluye.

Por ejemplo, si utilizamos el código del listado 7.50, saltaremos los artículos cuyo precio de venta sea superior a 0.9; en cuanto se encuentre uno que no cumple esa condición, se incluirá ese artículo y **todos** los que sigan (independientemente de que se cumpla o no la condición).

```
var res1 = (from a in artículos
            select a).SkipWhile(a => a.PrecioVenta > 0.9M);
```

Listado 7.50. SkipWhile se saltará los elementos que cumplan la condición hasta que se encuentre uno que no la cumpla

Si realmente quisiéramos todos los elementos que **no** tengan un precio mayor de 0.9, podríamos usar la cláusula **where**, tal como vemos en el listado 7.51.

```
var res2 = from a in artículos
            where a.PrecioVenta <= 0.9M
            select a;
```

Listado 7.51. Si realmente queremos todos los elementos que cumplan una condición, es preferible usar where

TakeWhile funciona de forma parecida, solo que en lugar de saltarse los elementos que cumplan esa condición, los tomará “mientras” se cumpla; en este caso, hay que comentar lo mismo que con **SkipWhile**, ya que cuando se encuentre con un elemento que no cumpla esa condición, se dejará de comprobar y, por tanto, solo se incluirán en la consulta los primeros elementos que cumplan esa condición. Aunque en el caso de **TakeWhile** podemos tener la certeza de que solo se tomarán los primeros elementos que cumplan la condición indicada en la expresión *lambda* del argumento pasado a la función.

Por ejemplo, con el código del listado 7.52, la consulta tendrá cero elementos, ya que la condición es “toma los elementos mientras sea mayor de tres” y como resulta que el primer elemento es menor de ese valor, pues, deja de analizar los restantes.

```
var nums = new[] { 1, 9, 8, 2, 5, 7, 4, 3, 6 };
var res3 = nums.TakeWhile(n => n > 3);
```

Listado 7.52. TakeWhile solo tomará los que cumplan la condición hasta que se encuentre con un elemento que no la cumple

Por supuesto, podemos combinar **Skip** y **Take** con o sin **While**. Por ejemplo, en el código del listado 7.53, nos saltamos unos cuantos elementos y después ponemos la condición de los que queremos tomar, en este caso en particular, obtendríamos los valores 5, 7 y 4.

```
var res4 = nums.Skip(4).TakeWhile(n => n > 3);
```

Listado 7.53. Podemos mezclar Skip y Take con o sin While

Vuelve un clásico de los tiempos de BASIC: let

Creo que desde los tiempos del *Spectrum* no se utiliza la instrucción **let** para asignar valores a las variables, de hecho, si la usamos en cualquier versión de Visual Basic para .NET, automáticamente quitará esa instrucción y solo dejará la asignación, (en Visual Basic 6.0 y anteriores la sigue dejando ya que tiene su propia “razón de ser”). En C# esa instrucción (o cláusula) es nueva, y solo se puede usar en las consultas de LINQ, pero la funcionalidad es la misma que en BASIC: declarar variables, aunque en C# la usaremos para asignar un valor a la variable y el tipo de datos que tendrá esa variable se infiere del tipo de datos que estemos asignando, aunque las variables declaradas con la cláusula **let** tienen un ámbito exclusivo en la consulta de LINQ (como las variables de rango usadas después de **from**).

Las variables declaradas con **let** las podemos usar, por ejemplo, en la condición **where** para que resulte más legible nuestro código. Aunque también lo podemos usar para crear una variable con los datos indicados en la expresión que asignamos. Lo que sí debemos saber es que una vez asignado un valor por medio de **let**, ya no se puede volver a asignar otro a esa misma variable. Es como si la variable fuese de solo lectura y solo permite una única asignación.

En el listado 7.54 tenemos un ejemplo algo rebuscado de cómo usar la cláusula **let** para hacer asignaciones intermedias en una misma consulta.

```
var res1 = from f in facturas
            let arts = f.Articulos
            let res = from a in arts
                      where a.Código.StartsWith("0")
                      select a
            select new { f, arts, res };
```

Listado 7.54. let permite crear variables intermedias en una consulta

En el código del listado 7.54, la variable **res1** contendrá elementos con tres propiedades: una será cada una de las facturas que cumplan las condiciones puestas (el elemento **f**); otra es el contenido de todos los artículos de cada factura (elemento **arts** de la consulta); y, por último, también tendrá una colección que será la asignada a la variable **res**, que como vemos es el resultado de otra consulta que utiliza los valores almacenados en la variable **arts**.

En el listado 7.55 vemos cómo mostrar algunos de los datos contenidos en la colección resultante de la consulta.

```
foreach(var r in res1)
{
    Console.WriteLine(r.f.Numero);
    foreach(var a in r.res)
    {
        Console.WriteLine(" {0}", a);
    }
}
```

Listado 7.55. Mostrando los datos de la consulta del listado 7.54

También podemos usar **let** para crear un valor intermedio que después podemos usar en la cláusula **where**, como en el código del listado 7.56 en el que se asigna una variable llamada *par* que contendrá un valor verdadero si el número examinado es par.

```
var res1 = from n in nums
            let par = n % 2 == 0
            where par
            select "El número " + n + " es par";
```

Listado 7.56. Podemos usar let en construcciones simples

En el ejemplo del listado 7.56 simplemente usamos el valor como condicionante en la cláusula **where**, pero si quisiéramos que se mostraran todos los números y nos indicara si es par o impar, podríamos hacerlo tal como se muestra en el listado 7.57.

```
var res2 = from n in nums
            let par = n % 2 == 0
            let parText = par ? "par" : "impar"
            select "El número " + n + " es " + parText;
```

Listado 7.57. let lo podemos usar para crear variables a partir de otras ya creadas anteriormente

Y para terminar, veamos en el listado 7.58 cómo usar la cláusula **let** para asignar el valor de una función de agregado de una sub consulta y valores calculados a partir de otros datos intermedios.

```
var res1 = from f in facturas
            let cant = f.Articulos.Count
            let totArt = (from a in f.Articulos
                          select a).Sum(n => n.PrecioVenta)
            let media = totArt / cant
            select new
            {
                Fact = f,
                Cant = cant,
                SumaPrecios = totArt,
                Media = media
            };
```

Listado 7.58. A las variables creadas con let, podemos asignarle cualquier valor válido, ya sea simple o calculado desde otra sub consulta

Y con esto terminamos nuestra revisión de las instrucciones que se han incorporado al lenguaje para utilizarlo en las consultas de LINQ y algunas de las funciones definidas como métodos extensores que nos pueden ser de utilidad.

Versiones

Esta característica solo la podemos usar con .NET Framework 3.5 y debemos tener una referencia a System.Linq.dll (todas las plantillas de proyectos de Visual C# 2008 incluyen esa referencia).

Respuesta al ejercicio de la sección dedicada a where

La pregunta estaba hecha con algo de trampa, ya que si en realidad la propuesta fuera: “*todos los artículos cuyos precios sean mayores de 0.75 o menores de 0.95*”, esto sería una tontería, ya que siempre se cumplirá que los precios o sean mayor que el primer valor o menor que el segundo, es decir, se incluirían todos. Pero la aclaración que seguía: “*que estén comprendidos entre 0.75 y 0.95 (ambos inclusive)*”, dejaba claro lo que se pretendía obtener: solamente los artículos cuyos precios de venta estén comprendidos entre los dos indicados, además de los que tengan esos valores. Por otra parte, queremos que también se incluyan todos los que tengan un valor 7 en la propiedad **IVA**, independientemente del precio que tuvieran.

El código lo podemos escribir haciendo la comprobación de los precios al principio, esos dos precios los comprobaremos con **AND** (& o && que es más eficiente) y la comparación será: precio >= 0.75 y precio <= 0.95. A esta doble comparación le seguirá una instrucción **OR** (| o ||) comprobando el valor de **IVA**. El listado 7.59 muestra ese código.

La propuesta de invertir las comparaciones, también tenía un poco de trampa, pero en realidad no tenemos que hacer nada en especial, salvo hacer primero la comparación del valor de **IVA** y usar el operador **||** (**OR**) para evaluar los precios de venta; el uso de paréntesis es opcional, ya que solamente cuando la primera condición no se cumpla es cuando se evaluará la siguiente, y debido a cómo funciona el operador condicional **AND** (&&), solo se seguirá analizando si esa primera se cumple y se dará por buena si la siguiente también se cumple. El código será el mostrado en el listado 7.60.

```
var res3 = from a in artículos
           where a.PrecioVenta >= 0.75M
               && a.PrecioVenta <= 0.95M
               || a.IVA == 7M
           select new
           {
               ID = a.Descripción,
               IVA = a.IVA,
               PVP = a.PrecioVenta
           };
```

Listado 7.59. Respuesta al ejercicio propuesto en where

```

var res4 = from a in artículos
           where a.IVA == 7M
              || a.PrecioVenta >= 0.75M
              && a.PrecioVenta <= 0.95M
           select new
           {
               ID = a.Descripción,
               IVA = a.IVA,
               PVP = a.PrecioVenta
           };

```

Listado 7.60. Segunda respuesta al ejercicio

En cuanto a cuál es más eficiente de las dos formas de hacerlo, en este caso en particular (por los precios y el valor de la propiedad **IVA** de los artículos), si contamos el número de comparaciones, el primer código hará una comparación menos que el segundo, y aunque eso no suponga mucho ahorro de tiempo, sería la que se llevaría el premio. Sin embargo, si hubiera mayoría de artículos que tengan la propiedad **IVA** con valor 7, la segunda sería más efectiva, ya que solo se necesita una comprobación para saber si debe tenerse en cuenta ese artículo (la del **IVA**). Pero también tenemos que tener en cuenta, que si tenemos más artículos que cumplan la condición de los precios, y el valor de **IVA** no es 7, la primera será más eficiente, ya que en el segundo código, habrá que hacer tres comparaciones para evaluar esos artículos (la del **IVA** y las dos de los precios).

En cualquier caso, siempre que utilicemos una comparación que dependa de que dos valores se cumplan (**AND**) deberíamos encerrarlos entre paréntesis para facilitar la lectura, tal como vemos en los extractos de código del listado 7.61.

```

where (a.PrecioVenta >= 0.75M
      && a.PrecioVenta <= 0.95M)
      || a.IVA == 7M

where a.IVA == 7M
      || (a.PrecioVenta >= 0.75M
      && a.PrecioVenta <= 0.95M)

```

Listado 7.61. Cuando usemos comparaciones con AND, mejor ponerlas entre paréntesis

Funciones de agregado personalizadas

Las funciones de agregado (o métodos de agregación) las crearemos como métodos extensores de la interfaz *generic* **IEnumerable<T>**, y según los tipos de datos que queramos manipular (o para los que permitamos el uso de esa función), crearemos tantas sobrecargas como necesitemos. Para simplificar, vamos a crear una función de agregado que devolverá la suma de todos los números pares de una colección (o *array*) de tipo entero.

En el listado 7.62 vemos la definición de esa función, que como podemos comprobar no es más que un método extensor de la interfaz **IEnumerable<int>**, por tanto, solo la podremos usar con valores de ese tipo (el compilador se encarga de que no la podamos usar con colecciones de un tipo diferente, ya que solo nos mostrará esa función si la colección es del tipo para el que lo hemos definido).

Como vemos en el listado 7.62, este tipo de funciones actúan sobre la colección indicada en el primer parámetro; en este caso concreto, recorreremos todos los valores de esa colección y vamos sumando solo los valores que son pares. El código usado en la función, es un bucle normal, pero esto sólo lo he hecho así por simplificar y para que se vea más claro cuál es el propósito de la función, pero en ese código también podríamos usar consultas de LINQ para obtener los valores que buscamos. Por ejemplo, en el listado 7.63 tenemos la definición de otra función de agregado que devuelve la suma de los valores impares de la colección, pero en lugar de emplear un bucle tradicional, he optado por emplear la función **Sum** al resultado de una consulta que devuelve los números impares de la colección indicada (el filtro de los valores a tener en cuenta lo hacemos en la condición de la cláusula **where**).

Para usar el método extensor del listado 7.62 (o el del listado 7.63), lo haremos como con cualquier otra función de agregado, es decir, como cualquier otro método extensor. En el listado 7.64 vemos varias formas de utilizar el método **SumaPares** (por supuesto, esta función solo la podremos usar con colecciones -o resultados de consultas- de tipo entero).

```
public static class ExtensionesAgregados
{
    public static int SumaPares(this IEnumerable<int> source)
    {
        int t = 0;
        foreach(var n in source)
        {
            if(n % 2 == 0)
                t += n;
        }
        return t;
    }
}
```

Listado 7.62. Método extensor para usar como función de agregado

```
public static int SumaImpares(this IEnumerable<int> source)
{
    var res2 = from n in source
               where n % 2 != 0
               select n;
    return res2.Sum();
}
```

Listado 7.63. Función de agregado que internamente utiliza la función agregada Sum


```

var nums = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var resPar = nums.SumPares();
var resPar1 = (from n in nums
               where n > 3
               select n).SumPares();

```

Listado 7.64. Varios ejemplos que utilizan la función de agregado del listado 7.62

Si esas funciones de agregado las compilamos y las usamos desde Visual Basic 2008 (VB 9.0), las podríamos usar con la cláusula **Aggregate**, es decir, el compilador de Visual Basic las consideraría como funciones de agregado.

En el listado 7.65 vemos cómo usar las dos funciones de agregado que hemos definido en los listados 7.62 y 7.63 desde Visual Basic (en el código que acompaña al libro se incluyen los proyectos completos).

```

Dim resPar = Aggregate n In nums Into SumaPares()
Dim resPar1 = nums.SumPares()
Dim resPar2 = Aggregate n In nums Where n > 3 Into SumaPares()

Dim resImpar = Aggregate n In nums Into SumaImpares()
Dim resImpar1 = nums.SumImpares()
Dim resImpar2 = Aggregate n In nums Where n > 3 Into SumaImpares()

```

Listado 7.65. Desde Visual Basic podemos usar nuestras funciones con la cláusula Aggregate

(Esta página se ha dejado en blanco de forma intencionada)

Capítulo 8

C# y LINQ (III)

Otros métodos extensores para consultas LINQ

En el capítulo anterior ya vimos las cláusulas o instrucciones incorporadas en C# para realizar consultas de LINQ, también vimos que esas instrucciones finalmente se convierten en métodos extensores, es decir, el compilador sustituye las instrucciones incorporadas en C# en llamadas a métodos extensores, y comenté algunos de esos métodos extensores que en Visual Basic se han incorporado como instrucciones propias del lenguaje. En este capítulo vamos a revisar otros métodos extensores que podremos usar en nuestras consultas de LINQ.

Nota

*Le recomiendo al lector que revise la documentación de Visual Studio 2008 para ver todos los métodos extensores que se han agregado a las interfaces **IEnumerable<T>** e **IQueryable<T>**, particularmente en el tópico **Información general sobre operadores de consulta estándar**, particularmente la sección **Secciones relacionadas** en la que se encontraremos enlaces a todos estos métodos englobados por el tipo de funcionalidad que proporcionan.*

Voy a usar la misma lista de operaciones que podemos hacer con esos métodos extensores usada en la documentación de Visual Studio 2008 para indicar qué funciones se incluyen y cómo usarlas. Aunque no veremos todas las sobrecargas de esas funciones, ya que intentaré centrarme en las que yo considero más importantes, también porque al explicar algunas de ellas, las otras sean fáciles de utilizar.

Muchas de las funciones que se relacionan en este capítulo ya las hemos visto en el capítulo anterior, por tanto, simplemente dejaré la referencia al nombre. Confío que después de leerlo, al lector le quede claro cómo usar todos estos métodos extensores para poder sacarle el máximo provecho a LINQ desde C#.

Ordenar datos

En este apartado se incluyen todos los métodos que dan funcionalidad a la cláusula **orderby** además del método **Reverse**, las funciones englobadas en este grupo son:

- OrderBy
- OrderByDescending
- ThenBy
- ThenByDescending
- Reverse

Todas estas ya las vimos en el capítulo anterior, solo aclarar que la misma instrucción **orderby** (con o sin las cláusulas **ascending** y **descending**) se utiliza para las cuatro primeras.

Las variantes **ThenBy** se utilizan cuando se indican varios elementos de ordenación en la misma cláusula **orderby**.

Operaciones de conjuntos (Set)

En este apartado se incluyen todas las operaciones con conjuntos, de las cuales ya vimos cómo usar **Distinct**, pero que también nos pueden ser interesantes las otras tres, de las que veremos algunos ejemplos para entender mejor su funcionamiento. Estas funciones son:

- Distinct
- Except
- Intersect
- Union

Debido a que estos métodos (salvo **Distinct**) trabajan sobre dos conjuntos de datos, en estos ejemplos usaremos dos colecciones (en realidad dos *arrays* numéricos) en los que algunos elementos están incluidos en las dos. Esos *arrays* son los mostrados en el listado 8.1.

```
var nums1 = new[] { 1, 2, 3, 3, 4, 5, 6 };  
var nums2 = new[] { 1, 3, 5, 6, 6, 7, 8 };
```

Listado 8.1. Los dos arrays usados para las funciones de operaciones de conjuntos

Except devuelve los elementos de la colección a la que se aplica el método que no están en la colección indicada como argumento. En el listado 8.2 vemos un ejemplo de cómo usar este método, la colección resultante solo contendrá los valores 2 y 4.

```
var res2 = nums1.Except(nums2);
```

Listado 8.2. Ejemplo de Except

Como ejercicio invito al lector a que cambie los *arrays* usados con este método para ver qué valores devolvería en cada caso. Por ejemplo, en el listado 8.3 el método **Except** lo aplicamos al *array* **nums2** y pasamos como argumento el *array* **nums1** (para las siguientes funciones no mostraré el

código, pero también sería buen ejercicio hacer ese cambio mentalmente y comprobar si hemos acertado, en los proyectos de ejemplo que acompañan al libro se incluye el código completo).

```
var res02 = nums2.Except(nums1);
```

Listado 8.3. Segundo ejemplo de Except

El método **Intersect** devuelve los elementos que están en ambas colecciones (sin repeticiones). Si usamos el ejemplo del listado 8.4 la colección resultante contendrá los valores: 1, 3, 5 y 6.

```
var res3 = nums1.Intersect(nums2);
```

Listado 8.4. Ejemplo de Intersect

El método **Union** combina las dos colecciones, quitando los elementos duplicados. Después de usar el código del listado 8.5 obtendremos los valores: 1, 2, 3, 4, 5, 6, 7 y 8, es decir, todos salvo los repetidos.

```
var res4 = nums1.Union(nums2);
```

Listado 8.5. Ejemplo de Union

Filtrar datos

El filtrado de datos lo usaremos para incluir en la consulta solo aquellos datos que cumplan cierta condición, la cláusula **where** es la que utilizaremos para este tipo de operaciones de filtrado, pero además del método **Where** (que es el que el compilador utiliza cuando se encuentra con la cláusula **where**), en este mismo grupo también tenemos el método **OfType**, que podemos usar para seleccionar los datos que sean del tipo de datos indicado (o de uno derivado).

En el listado 8.6 vemos un ejemplo que podemos aplicar este método para crear una colección con todos los controles de un formulario que sean del tipo **TextBox**.

```
var res1 = this.Controls.OfType<TextBox>();
```

Listado 8.6. El método OfType lo usaremos para filtrar los datos del tipo indicado

Si tenemos una colección de datos como la del listado 8.7 en el que se incluyen elementos del tipo **Cliente** y del tipo **ClienteVIP** (que se deriva de **Cliente**), podemos utilizar el código del listado 8.8 para realizar filtros tanto para obtener todos los elementos que sean del tipo **Cliente** (en los que se incluirán también los de tipo **ClienteVIP**) o solo los que sean del tipo **ClienteVIP**.

```
var clientes = new[] {  
    new Cliente { Nombre = "Pepe"},  
    new Cliente { Nombre = "Luis"},  
    new ClienteVIP { Nombre = "Juan"},  
    new Cliente { Nombre = "Elena"},  
    new ClienteVIP { Nombre = "Luisa"}  
};
```

Listado 8.7. Array con valores de los tipos Cliente y ClienteVIP

```
var res1 = clientes.ofType<Cliente>();  
var res2 = clientes.ofType<ClienteVIP>();
```

Listado 8.8. Solo los elementos que sean del tipo indicado (o se derive de ese tipo)

Operaciones cuantificadoras

En este grupo se incluyen las funciones **All** y **Any** que ya vimos en el capítulo anterior, además del método **Contains**, el cuál podremos usar para filtrar los datos que contengan el elemento que indiquemos como argumento. Este método es parecido al que algunas colecciones incluyen, la ventaja de este método extensor es que lo podemos aplicar a todos los tipos de datos que implementen las interfaces **IEnumerable<T>** o **IQueryable<T>**.

La forma de usar este método es bien simple, indicaremos el elemento que queremos comprobar si se encuentra en la colección y devolverá un valor verdadero o falso según esté o no. Si usamos el código del listado 8.9 la variable `res1` contendrá un valor `true`, ya que el valor 3 está en el *array* al que aplicamos el método.

```
var nums = new[] { 1, 2, 3, 4, 5 };  
var res1 = nums.Contains(3);
```

Listado 8.9. Ejemplo del método Contains

Operaciones de proyección

Este grupo incluye los métodos **Select** y **SelectMany** que equivalen a usar la cláusula **select**. La equivalencia de **SelectMany** es cuando usamos varias cláusulas **from**.

En los listados 8.10 y 8.11 vemos códigos equivalentes usando la cláusula **select** o los dos métodos extensores. En el primer caso, una forma simple de usar **Select**. En el segundo, al incluir un filtro con **Where** vemos que el uso del método **SelectMany** se complica un poco.

```
var res1 = from n in nums1
           select "Número " + n;

var res01 = nums1.Select(n => "Número " + n);
```

Listado 8.10. Equivalencia entre la cláusula select y el método Select

```
var res2 = from n1 in nums1
           from n2 in nums2
           where n1 > n2
           select n1 + " es mayor que " + n2;

var res02 = nums1.SelectMany(n1 => nums2, (n1, n2) => new { n1, n2 }).
    where(n => n.n1 > n.n2).
    Select(n => n.n1 + " es mayor que " + n.n2);
```

Listado 8.11. Equivalencia entre la cláusula select y el método SelectMany

Particiones de datos

Para las operaciones de particiones de datos no existen cláusulas propias de C#, por tanto, usaremos los métodos **Skip**, **SkipWhile**, **Take** y **TakeWhile** que ya vimos en el capítulo anterior.

Operaciones de combinación

En este grupo se incluyen los métodos **Join** y **GroupJoin** que como vimos en el capítulo anterior tienen su equivalencia con las cláusulas **join** y **join into** respectivamente.

Agrupar datos

Los métodos incluidos en este grupo de operaciones son:

- GroupBy
- ToLookup

El primero equivale a la cláusula **group by** que ya vimos en el capítulo anterior, y el segundo no tiene equivalencia con ninguna instrucción de C# y lo podemos usar tal como se muestra en el listado 8.12, en el que agrupamos los elementos de la colección *clientes* según el valor de la propiedad *País*, ese valor será la clave de cada elementos de la colección resultante que a su vez contendrá una colección con todos los valores que hayamos indicado en la segunda expresión *lambda* usada como argumento del método. Por tanto, en el resultado de ese código tendremos los clientes agrupados según el valor de la propiedad *País*.

```
var clientes = new[] {  
    new Cliente { Nombre = "Pepe", País = "ES" },  
    new Cliente { Nombre = "Luis", País = "UK" },  
    new ClienteVIP { Nombre = "Juan", País = "ES" },  
    new Cliente { Nombre = "Elena", País = "ES" },  
    new ClienteVIP { Nombre = "Luisa", País = "UK" }  
};  
  
var res1 = clientes.ToLookup(c => c.País, c => c.Nombre + " " + c.País);  
  
foreach(var r in res1)  
{  
    Console.WriteLine(r.Key);  
    foreach(var c in r)  
        Console.WriteLine(" {0}", c);  
}
```

Listado 8.12. Ejemplo del método ToLookup

Operaciones de generación

Las funciones incluidas en este grupo son:

- DefaultIfEmpty
- Empty
- Range
- Repeat

DefaultIfEmpty la podemos usar para devolver un valor predeterminado en caso de que la colección a la que se aplica pueda estar vacía, si no proporcionamos ese valor predeterminado, se devolverá el valor predeterminado del tipo contenido en la colección, si son tipos por referencia, el valor predeterminado es un valor nulo (**null**).

En el listado 8.13 tenemos una colección de clientes y definimos un valor predeterminado. Después creamos un par de consultas, en la primera (listado 8.14) no produce ningún valor, por tanto, se usa el valor predeterminado que hemos indicado, en la segunda, (listado 8.15) al coincidir algunos elementos, es decir, la colección resultante no está vacía, se devolverán esos elementos.

```
var clientes = new[] {  
    new Cliente { Nombre = "Pepe", País = "ES" },  
    new Cliente { Nombre = "Luis", País = "UK" },  
    new ClienteVIP { Nombre = "Juan", País = "ES" },  
    new Cliente { Nombre = "Elena", País = "ES" },  
    new ClienteVIP { Nombre = "Luisa", País = "UK" }  
};  
  
var cliDefault = new Cliente { Nombre = "Guille", País = "ES" };
```

Listado 8.13. Datos para los ejemplos de DefaultIfEmpty


```
var res = from c in clientes
          where c.País == "US"
          select c;

var res2 = res.DefaultIfEmpty(cliDefault);
```

Listado 8.14. Si la consulta está vacía, se devuelve el valor indicado en el argumento

```
var res1 = from c in clientes
           where c.País == "ES"
           select c;

var res3 = res1.DefaultIfEmpty(cliDefault);
```

Listado 8.15. En este ejemplo la consulta no está vacía, por tanto, se usan los datos devueltos por la consulta

Empty simplemente genera una colección vacía del tipo indicado. Este método es estático y está definido en la clase **Enumerable**, la forma de usarlo sería tal como vemos en el listado 8.16.

```
var res4 = Enumerable.Empty<Cliente>();
```

Listado 8.16. Forma de usar el método Empty de la clase Enumerable

Range genera una colección con los valores numéricos indicados en los argumentos. Éste también es un método estático de la clase **Enumerable** y lo podemos usar para generar una colección (ver listado 8.17) o como fuente de una colección a la que podemos aplicar cualquiera de las operaciones de consulta de LINQ (ver listado 8.18).

```
var res5 = Enumerable.Range(1, 10);
```

Listado 8.17. Ejemplo de uso del método Range

```
// solo los valores pares
var res6 = from n in Enumerable.Range(1, 10)
           where n % 2 == 0
           select n;
```

Listado 8.18. Ejemplo de uso del método Range en una consulta de LINQ

Repeat es un método estático de la clase **Enumerable** que genera una colección que repite el dato indicado como primer argumento el número de veces del valor indicado en el segundo argumento.

Si usamos el código del listado 8.19 obtendremos una colección del tipo **IEnumerable<string>** con 10 cadenas como la indicada en el primer argumento.

```
var res7 = Enumerable.Repeat("Repitiendo que es gerundio", 10);
```

Listado 8.19. Repeat crea una colección con el dato indicado en el primer argumento tantas veces como se indique en el segundo

Operaciones de igualdad

En este grupo tenemos el método **SequenceEqual** el cual comprueba si dos colecciones son iguales, es decir, si tienen el mismo número de elementos y éstos son iguales. El valor devuelto es del tipo **bool**. Si no indicamos cómo comparar los elementos, se usará el comparador predeterminado de igualdad.

En el listado 8.20 vemos un ejemplo simple, mientras que en el listado 8.22 hemos definido una clase para realizar una comparación personalizada (que implementa **IEqualityComparer<T>**) en la que no se tiene en cuenta la diferencia entre mayúsculas y minúsculas en las cadenas (listado 8.21). En el primer caso, devolverá **false** y en el segundo devolverá **true**.

```
var str1 = new[] { "Hola", "Mundo" };
var str2 = new[] { "hola", "mundo" };

var res1 = str1.SequenceEqual(str2);
```

Listado 8.20. Ejemplo de uso del método SequenceEqual

```
class MiStringComparer : IEqualityComparer<string>
{
    public bool Equals(string x, string y)
    {
        return x.ToUpper().Equals(y.ToUpper());
    }

    public int GetHashCode(string obj)
    {
        return obj.ToUpper().GetHashCode();
    }
}
```

Listado 8.21. Una clase que implementa IEqualityComparer<T> para comparar dos cadenas

```
var strComp = new MiStringComparer();
```

```
var res2 = str1.SequenceEqual(str2, strComp);
```

Listado 8.22. Ejemplo de uso del método SequenceEqual usando una clase para realizar las comparaciones

Operaciones de elementos

Las funciones que encontramos en este grupo son:

- ElementAt
- ElementAtOrDefault
- First
- FirstOrDefault
- Last
- LastOrDefault
- Single
- SingleOrDefault

Como vemos, existen dos versiones de cada método, en el sentido de que la misma acción la podemos realizar con o sin un valor predeterminado (recordemos que el valor predeterminado de los tipos por referencia es un valor nulo). Los métodos que finalizan su nombre con **OrDefault** devolverán un valor nulo si el elemento al que se quiere acceder no está disponible, mientras que las otras funciones, si ese elemento no existe (o ninguno cumple la condición indicada) se producirá una excepción, es decir, en estos casos, **Default** no significa que nosotros podamos dar un valor predeterminado para los casos que no exista el elemento al que queremos acceder.

Todos estos métodos (salvo **ElementAt** y **ElementAtOrDefault**) definen una sobrecarga en la que podemos indicar una expresión *lambda* que se usará para filtrar los datos, en el código que mostraré a continuación uso tres expresiones *lambda* para filtrar esos datos, por comodidad he almacenado esas funciones en variables, pero las podríamos usar directamente. En el listado 8.23 están definidas esas tres expresiones *lambda* que nos sirven para filtrar los datos de la colección *clientes* (ver listado 8.24) por el contenido de la propiedad *País*.

```
Func<Cliente, bool> predicateUK = c => c.País == "UK";  
Func<Cliente, bool> predicateAR = c => c.País == "AR";  
Func<Cliente, bool> predicateUS = c => c.País == "US";
```

Listado 8.23. Expresiones lambda usadas para filtrar los datos

```
var clientes = new[] {  
    new Cliente { Nombre = "Pepe", País = "ES" },  
    new Cliente { Nombre = "Luis", País = "UK" },  
    new ClienteVIP { Nombre = "Juan", País = "ES" },  
    new Cliente { Nombre = "Elena", País = "ES" },  
    new ClienteVIP { Nombre = "Luisa", País = "UK" },  
    new Cliente { Nombre = "Emilio", País = "US" }  
};
```

```
var cliDefault = new Cliente { Nombre = "Guille", País = "ES" };  
var cliVacia = Enumerable.Empty<Cliente>();
```

Listado 8.24. Colección y variables usadas en los ejemplos

Por suerte, ya sabemos cómo podemos crear nuestros propios métodos extensores, de forma que nosotros mismos podemos crear nuestras propias sobrecargas de estos métodos para que se incluya el valor que indiquemos si lo que se devolverá es un valor nulo. En el listado 8.25 vemos cómo crear nuestras propias versiones de las sobrecargas de estos métodos que tienen en cuenta si el valor devuelto es nulo, en cuyo caso, se usará el valor indicado como predeterminado. Hay que tener en cuenta que el método **SingleOrDefault** definido en .NET producirá una excepción si la colección a la que se aplica contiene más de un elemento, por tanto, la funcionalidad de la sobrecarga definida en este código no tiene la misma funcionalidad, ya que en caso de que la colección contenga más de un elemento devolverá el valor predeterminado que indiquemos.

```
static class Extensiones  
{  
    public static TSource ElementAtOrDefault<TSource>(this IEnumerable<TSource> source, int index, TSource defaultValue)  
    {  
        var res = source.ElementAtOrDefault(index);  
        if(res == null)  
            return defaultValue;  
        return res;  
    }  
  
    public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source, TSource defaultValue)  
    {  
        var res = source.FirstOrDefault();  
        if(res == null)  
            return defaultValue;  
        return res;  
    }  
  
    public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate, TSource defaultValue)  
    {  
        var res = source.FirstOrDefault(predicate);  
        if(res == null)  
            return defaultValue;  
        return res;  
    }  
}
```

```
public static TSource LastOrDefault<TSource>(
    this IEnumerable<TSource> source,
    TSource defaultValue)
{
    var res = source.LastOrDefault();
    if(res == null)
        return defaultValue;

    return res;
}

public static TSource LastOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate,
    TSource defaultValue)
{
    var res = source.LastOrDefault(predicate);
    if(res == null)
        return defaultValue;

    return res;
}

public static TSource SingleOrDefault<TSource>(
    this IEnumerable<TSource> source,
    TSource defaultValue)
{
    try
    {
        var res = source.SingleOrDefault();
        if(res == null)
            return defaultValue;

        return res;
    }
    catch
    {
        return defaultValue;
    }
}

public static TSource SingleOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate,
    TSource defaultValue)
{
    try
    {
        var res = source.SingleOrDefault(predicate);
        if(res == null)
            return defaultValue;

        return res;
    }
    catch
    {

```

```
        return defaultvalue;
    }
}
```

Listado 8.25. Sobrecargas que devuelven un valor en lugar de producir una excepción

Una vez que tenemos las sobrecargas definidas, veamos cuál es el comportamiento de cada uno de estos métodos, tanto los definidos por el propio .NET Framework como los que están definidos en el listado 8.25.

ElementAt sirve para acceder al elemento que esté en el índice indicado en el argumento, mientras que **ElementAtOrDefault** devolverá un valor nulo (el predeterminado del tipo) en el caso de que el índice indicado esté fuera del rango de elementos contenidos en la colección. Usando la sobrecarga definida en el listado 8.25 se devolverá el elemento indicado como predeterminado en lugar de un valor nulo (esto mismo es aplicable al resto de sobrecargas que hemos definido para los otros métodos). En el listado 8.26 vemos las tres formas de usar estos métodos. En el segundo ejemplo, al no existir un elemento en la posición 10, se devuelve un valor nulo, mientras que en el último, se devuelve el elemento indicado como segundo argumento.

```
cliente resElementAt = clientes.ElementAt(3);
resElementAt = clientes.ElementAtOrDefault(10);
// Usando la sobrecarga definida en el listado 8.25
resElementAt = clientes.ElementAtOrDefault(10, cliDefault);
```

Listado 8.26. Ejemplos de uso de ElementAt y ElementAtOrDefault

El método **First** devuelve el primer elemento de la colección, si se indica una expresión *lambda*, devolverá el primer elemento que coincida con la expresión indicada, si la colección está vacía o no hay datos que coincida con la expresión *lambda* indicada, fallará produciendo una excepción del tipo **InvalidOperationException**. Por otra parte, **FirstOrDefault** devolverá un valor nulo si no hay coincidencias o la colección está vacía. Usando las sobrecargas definidas en el listado 8.25 si no hay datos coincidentes, se devolverá el elemento indicado como predeterminado. En el listado 8.27 vemos cómo usar estas funciones.

```
// El primero de la colección
var resFirst = clientes.First();

// fallará si la colección está vacía
resFirst = cliVacía.First();

// El primero que coincida con la expresión lambda
resFirst = clientes.First(predicateUK);

resFirst = clientes.FirstOrDefault(predicateUK);

// Si la colección está vacía,
// devolverá un valor nulo
resFirst = cliVacía.FirstOrDefault();
```

```
// Este fallará si no hay datos de AR
resFirst = clientes.First(predicateAR);

// Usando nuestra sobrecarga siempre funcionará
resFirst = clientes.FirstOrDefault(predicateAR, cliDefault);
```

Listado 8.27. Ejemplos de uso de First y FirstOrDefault

El método **Last** devuelve el último elemento de la colección, si se indica una expresión *lambda*, devolverá el último elemento que coincida con esa expresión, si la colección está vacía o no hay datos que coincida con la expresión *lambda* indicada, se producirá una excepción. Como es de suponer, **LastOrDefault** devolverá un valor nulo si no hay coincidencias o la colección está vacía, y si usamos las sobrecargas definidas en el listado 8.25, nos aseguramos de que si no hay datos coincidentes, devuelva el valor indicado como predeterminado. En el listado 8.28 vemos varios ejemplos de cómo usar estas funciones.

Recordemos que si usamos una expresión *lambda* como argumento, se filtrarán los datos y se seleccionará el último elemento que coincida con ese filtro, si usamos **Last(expresión)** y no hay datos coincidentes, se producirá un error, pero al usar **LastOrDefault(expresión)**, si no hay datos, se devuelve un valor nulo.

```
// El último elemento de la colección
var resLast = clientes.Last();
Console.WriteLine(resLast);

// El último que coincida con la expresión lambda
resLast = clientes.Last(predicateUK);
Console.WriteLine(resLast);

resLast = clientes.LastOrDefault(predicateUK);
Console.WriteLine(resLast);

resLast = clientes.LastOrDefault(predicateUK, cliDefault);

// Este fallará si no hay datos de AR
resLast = clientes.Last(predicateAR);

// Usando nuestra sobrecarga siempre funcionará
resLast = clientes.LastOrDefault(predicateAR, cliDefault);
```

Listado 8.28. Ejemplos de uso de Last y LastOrDefault

El método **Single** es algo especial, ya que solo devolverá un valor, solo y exclusivamente si hay solamente un valor, o en caso de usar la sobrecarga de la expresión *lambda*, si solamente un valor coincide con la expresión indicada. **SingleOrDefault** funcionará solo si no hay elementos que coincida o solamente hay un elemento que cumpla la condición indicada. Es decir, si la colección tiene más de un elemento coincidente, estos métodos fallarán. Aunque esta es la funcionalidad de estos métodos, desde mi punto de vista, no es útil, ya que obligará a hacer comprobaciones extras cada vez que se usen. Afortunadamente, con las sobrecargas del listado 8.25 se puede solventar en parte esas comprobaciones aportando un valor por defecto, que bien puede ser un valor nulo, cuando no se cumpla la condición de que haya solamente un elemento en la colección que cumpla la condición.

Esto trae a colación lo que siempre digo: al final siempre somos nosotros los que debemos decidir cómo queremos que **nuestro** código actúe.

En el listado 8.29 vemos varios ejemplos de uso de **Single** y **SingleOrDefault**, tanto los definidos en .NET como nuestras sobrecargas definidas en el listado 8.25.


```

cliente resSingle;

// Estos dos darán error
// porque la colección tiene más de un elemento
resSingle = clientes.Single();

resSingle = clientes.SingleOrDefault();

// Con nuestra sobrecarga siempre funciona
resSingle = clientes.SingleOrDefault(cliDefault);

// Si hay uno o no hay, siempre funciona
resSingle = clientes.SingleOrDefault(predicateUS);

resSingle = clientes.SingleOrDefault(predicateAR);

// Si hay más de uno, fallará
resSingle = clientes.SingleOrDefault(predicateUK);

// Con nuestra sobrecarga siempre funciona
resSingle = clientes.SingleOrDefault(predicateUK, cliDefault);

```

Listado 8.29. Ejemplos de uso de Single y SingleOrDefault

Convertir tipos de datos

Los métodos que se engloban en este apartado sirven para cambiar el tipo de datos devuelto por las consultas de LINQ. Veamos una relación de los métodos extensores que podemos usar para convertir los datos y cómo usarlos, algunos de ellos ya los hemos usados en capítulos anteriores o como es el caso de **ToLookup** que ya vimos en la sección **Agrupar datos** de este mismo capítulo.

- AsEnumerable
- AsQueryable
- Cast
- OfType
- ToArray
- ToDictionary
- ToList
- ToLookup

El método **AsEnumerable** convierte una colección que implementa **IEnumerable<T>** en un tipo **IEnumerable<T>**, aunque esto pueda parecer que es una pérdida de tiempo, tiene su razón de ser, de forma que si la colección que queremos convertir define algunos métodos que podrían entrar en conflicto con los definidos en la interfaz **IEnumerable<T>**, se utilicen los definidos para esa interfaz, recordemos que los métodos definidos en los tipos tienen preferencia sobre los definidos como métodos extensores.

Por otro lado, el método **AsQueryable** convierte una colección que implementa **IEnumerable<T>** en una del tipo **IQueryable<T>**, si la colección implementa esta última interfaz, simplemente la devuelve, sin embargo, si la colección solo implementa **IEnumerable<T>**, la convierte a **IQueryable<T>** pero los métodos extensores que ejecuta son los de la clase **Enumerable** en vez de los de la clase **Queryable**.

En el listado 8.30 vemos un par de ejemplos de cómo usar estos dos métodos. En este código he declarado las variables **col1** y **col2** de forma explícita con los tipos de datos que recibirá, pero solo para dejar claro el tipo de datos que tiene cada variable, si hubiera usado **var** el resultado hubiera sido el mismo.

```
var nums = new[] { 1, 2, 3, 4, 5, 6 };
IEnumerable<int> col1 = nums.AsEnumerable();
IQueryable<int> col2 = nums.AsQueryable();

var res1 = from n in col1
           where n > 1
           select n;

var res2 = from n in col2
           where n > 1
           select n;
```

Listado 8.30. Ejemplos de uso de AsEnumerable y AsQueryable

El método **Cast** lo utilizaremos para indicar el tipo de datos que queremos usar para la variable de rango de una consulta, esto equivale a indicar el tipo de datos en la variable usada con la cláusula **from**. En el listado 8.31 vemos un ejemplo usando el método y otro indicando el tipo de datos en la cláusula **from**.

```
var nums = new[] { 1, 2, 3, 4, 5, 6 };

var res1 = from int n in nums
           select n;

var res2 = nums.Cast<int>().Select(n => n);
```

Listado 8.31. Ejemplos de conversión de tipos implícita

Como ya vimos en el apartado **Filtrar datos**, el método **OfType** lo usaremos para obtener de una colección solo aquellos datos que coincidan con el tipo indicado. En los listados 8.6 y 8.8 vimos dos ejemplos de cómo usar este método.

El método **ToArray** convierte la colección del tipo **IEnumerable<T>** a la que se aplica en un *array*. Por otro lado, **ToList** convierte esa colección en un objeto del tipo **List<T>**.

En el listado 8.32 tenemos una clase **Colega** de la que crearemos un *array* (listado 8.33) para convertirlo en otros tipos de datos. En el listado 8.34 por medio del método **ToList** convertimos ese

array en un objeto **List<Colega>**, le añadimos un nuevo elemento y lo volvemos a convertir en un *array* aplicando el método **ToArray** a la colección **List<Colega>** generada anteriormente. En este ejemplo, he creado un nuevo *array* como destino, pero también podríamos haber asignado el *array* generado por el método **ToArray** a la variable *colegas*.

```
class Colega
{
    public string Nombre { get; set; }
    public string Correo { get; set; }
}
```

Listado 8.32. Una clase simple para los ejemplos

```
var colegas = new[] {
    new Colega { Nombre = "Pepe", Correo = "pepe@nombres.com" },
    new Colega { Nombre = "Luis", Correo = "luis@nombres.com" },
    new Colega { Nombre = "Eva", Correo = "eva@nombres.com" }
};
```

Listado 8.33. Array con objetos del tipo Colega para convertir a otros tipos

```
// Convertimos el array en un objeto List<T>
var colList = colegas.ToList();

// Le añadimos un nuevo colega
colList.Add(
    new Colega {
        Nombre = "Luisa",
        Correo = "luisa@nombres.com"
    });

// Convertimos nuevamente la colección en un array
var colArray = colList.ToArray();
```

Listado 8.34. Ejemplos de ToList y ToArray

El método **ToDictionary** por su parte, convierte una colección del tipo **IEnumerable<T>** en un objeto del tipo **Dictionary<TKey, TElement>**. Existen básicamente dos sobrecargas (en realidad son cuatro sobrecargas, pero en dos de ellas se indica el comparador usado para comparar las claves), en ambas el tipo de la clave (**TKey**) se obtiene según la función anónima indicada como primer argumento. Si usamos la sobrecarga en la que solo indicamos una expresión *lambda*, el tipo de datos de cada valor (**TElement**) será el mismo que tenga la colección original (**T**), si usamos la sobrecarga que acepta dos expresiones *lambda*, la segunda será la que indique el tipo de datos de cada valor.

La colección mostrada en el listado 8.33 la usaremos para convertirla en un diccionario usando dos de las sobrecargas de este método. En el listado 8.35 vemos cómo seleccionar como clave la propiedad **Correo**, en este ejemplo, el valor de cada elemento de la colección generada será el mismo dato que está en el *array* original. Pero si queremos cada elemento de la colección generada sean de un tipo diferente al que contiene el *array*, podemos usar otra de las sobrecargas de este método, en el listado 8.36 vemos cómo indicar qué valor tendrá cada elemento de la colección **Dictionary** generada, en este caso será lo que la segunda expresión *lambda* devuelva.

```
var colDic = colegas.ToDictionary(c => c.Correo);  
foreach(var c in colDic)  
    Console.WriteLine("{0}, {1}", c.Value.Nombre, c.Key);
```

Listado 8.35. Convertimos el array en un objeto Dictionary<string, Colega>

```
var colDic2 = colegas.ToDictionary(c => c.Correo, c => c.Nombre);  
foreach(var c in colDic2)  
    Console.WriteLine("{0}, {1}", c.Value, c.Key);
```

Listado 8.36. Convertimos el array en un objeto Dictionary<string, string>

Todos los métodos que empiezan con **To** crean un nuevo objeto con los elementos que deben contener (es decir, crean una copia), mientras que los dos métodos que empiezan con **As** simplemente cambian el tipo de datos pero como una referencia a la colección generada (no crean una copia).

Operaciones de concatenación

El único método incluido en este grupo de operadores, es **Concat**, el cual podemos usar para unir (o concatenar) dos colecciones en una nueva. La nueva colección será generada cuando se enumere, es decir, se utiliza lo que se conoce como ejecución diferida (o aplazada), si necesitamos convertir esos datos en una colección o *array* “real”, podemos aplicarle cualquiera de los métodos que vimos en el apartado anterior.

En el listado 8.37 vemos un ejemplo de cómo concatenar dos colecciones, debido a que la ejecución es aplazada, si cambiamos cualquiera de los elementos de las colecciones que intervienen en esa concatenación, ese cambio se reflejara cuando volvamos a enumerar esa colección, tal como vemos en el código del listado 8.38.

```
var nums1 = new[] { 1, 3, 5 };  
var nums2 = new[] { 1, 2, 3 };  
var nums3 = nums1.Concat(nums2);
```

Listado 8.37. Concat creará una consulta con la unión de las dos colecciones indicadas

```

nums1[0] = 9;

foreach(var n in nums3)
    Console.Write("{0} ", n);
Console.WriteLine();

```

Listado 8.38. Cualquier cambio en los elementos de las dos colecciones se reflejará en la el resultado

Recordemos que si añadimos o eliminamos valores de cualquiera de las dos colecciones usadas con el método **Concat**, esos cambios no afectarán al resultado, al menos en los de la colección modificada. Por ejemplo, si cambiamos el tamaño del *array* **nums1** y también modificamos los datos que antes se tuvieron en cuenta, esto no cambiará el contenido que se obtuvo antes del cambio. Pero si cambiamos cualquiera de los valores de la otra colección, esos cambios sí que se reflejarán. El código del listado 8.39 muestra un ejemplo que clarifica lo comentado.

```

// Si cambiamos el tamaño de los orígenes
// la colección resultante no cambiará
Array.Resize(ref nums1, nums1.Length + 1);
nums1[nums1.Length - 1] = 88;

nums1[1] = 4;

foreach(var n in nums3)
    Console.Write("{0} ", n);
Console.WriteLine();

// Aunque si la otra colección
// sigue teniendo los mismos elementos
// si se incluirán los cambios

nums2[2] = 77;

foreach(var n in nums3)
    Console.Write("{0} ", n);
Console.WriteLine();

```

Listado 8.39. Si cambiamos el número de elementos de una de las colecciones, se mantendrán los valores que originalmente tenía

Operaciones de agregación

Los métodos para realizar las operaciones de agregación, son los que se enumeran a continuación, todos ellos (salvo **Aggregate**) ya los vimos en la sección **Funciones de agregado** del capítulo anterior.

- Aggregate
- Average
- Count
- LongCount
- Max

- Min
- Sum

El método **Aggregate** permite realizar una operación de agregado (o acumulación) en una colección. No debemos confundir este método con la cláusula del mismo nombre que define Visual Basic, ya que no hay una equivalencia exacta, en Visual Basic esa cláusula se utiliza para realizar operaciones de agregación, pero usando las funciones de agregado, tanto las definidas en .NET Framework (las mostradas en la lista anterior) o las que nosotros podamos definir en nuestro código (en el capítulo anterior vimos cómo definir las y cómo usarlas tanto desde Visual Basic como desde C#).

El método **Aggregate** define tres sobrecargas, según se indique o no un valor inicial (semilla) y si se quiere realizar una transformación (similar al uso de la cláusula **select**) en el valor devuelto. Veamos ejemplos de cómo utilizar las tres sobrecargas.

En el listado 8.40 vemos dos formas de sumar el contenido de una colección numérica, en el primer caso, usamos el método **Aggregate** al que le pasamos una expresión *lambda* con la operación de acumulación que queremos hacer, la primera variable de esa función anónima será la que mantenga el valor acumulado, y la segunda la que representa a cada elemento. El valor inicial que tendrá la variable usada como acumulador será el primer elemento del *array*. En la segunda asignación usamos la función **Sum** que hace eso mismo. En ambos casos, el valor devuelto es 15, que es la suma de todos los valores del *array* **nums**.

```
var nums = new[] { 1, 2, 3, 4, 5 };  
  
var suma1 = nums.Aggregate((t, n) => t + n);  
Console.WriteLine(suma1);  
  
var suma2 = nums.Sum();  
Console.WriteLine(suma2);
```

Listado 8.40. Dos formas de sumar todos los valores de una colección numérica

Las otras dos sobrecargas del método **Aggregate** permiten indicar el valor inicial del acumulador (semilla), (de esta forma, se usará ese valor en lugar del primer elemento de la colección) y si se debe usar una expresión *lambda* para indicar cuál será el elemento resultante.

En el listado 8.41 vemos un ejemplo de la sobrecarga en la que solo se indica la semilla y la operación de acumulación que queremos realizar. En este caso, el valor inicial será 10 y se hará la misma operación que en el ejemplo anterior, es decir, se sumarán todos los valores del *array* al que se le aplica este método. Por tanto, el resultado será 25 (10 + 15).

```
var suma3 = nums.Aggregate(10, (t, n) => t + n);  
Console.WriteLine(suma3);
```

Listado 8.41. Aggregate podemos usarlo indicando el valor inicial del acumulador

En el listado 8.42 usamos la sobrecarga en la que además del valor inicial de la semilla y la expresión *lambda* que usaremos para la operación de acumulación, también indicamos cómo queremos que se devuelva ese valor. En este ejemplo, creamos una cadena con el valor a devolver. En esta segunda expresión *lambda*, el valor que se recibe es el total acumulado.

```
var suma4 = nums.Aggregate(  
    10,  
    (t, n) => t + n,  
    t => "El total es " + t);  
  
Console.WriteLine(suma4);
```

Listado 8.42. Con Aggregate también podemos indicar cómo devolver el valor acumulado

En el listado 8.43 vemos una versión de los métodos *SumaPares* y *SumaImpares* (de los listados 7.62 y 7.63 del capítulo anterior) en las que usamos el método **Aggregate** para sumar los números pares o impares de una colección de tipo entero.

```
public static class ExtensionesAgregados  
{  
    public static int SumaPares(this IEnumerable<int> source)  
    {  
        return source.Aggregate(  
            0, (t, n) => t + (n % 2 == 0 ? n : 0)  
        );  
    }  
  
    public static int SumaImpares(this IEnumerable<int> source)  
    {  
        return source.Aggregate(  
            0, (t, n) => t + (n % 2 == 0 ? 0 : n)  
        );  
    }  
}
```

Listado 8.43. Versiones de las funciones de agregado definidas en el capítulo anterior en las que usamos el método Aggregate para realizar la suma de los valores

Estos ejemplos que he usado para mostrar cómo se utiliza el método **Aggregate** solo han sido para que veamos cómo podemos usarlo (*elemental querido Guille*) ya que no tienen mucha utilidad, puesto que para acumular valores de esta forma tenemos el método **Sum**, que como ya vimos en el capítulo anterior, le podemos pasar la expresión *lambda* que se aplicará para sumar cada elemento, por tanto, para sumar los valores pares e impares del array *nums* podríamos usar el código del listado 8.44 en el que las expresiones *lambda* usadas en el método **Sum** tendrán en cuenta si el número es par o impar (según sea el caso).

```
// Los impares  
var suma9 = nums.Sum(n => n % 2 == 0 ? 0 : n);  
Console.WriteLine("Suma de los impares: {0}", suma9);  
  
// Los pares  
var suma10 = nums.Sum(n => n % 2 == 0 ? n : 0);  
Console.WriteLine("Suma de los pares: {0}", suma10);
```

Listado 8.44. El método Sum hace perfectamente todo lo que acabamos de ver si usamos la expresión lambda adecuada

El resto de funciones de agregado ya vimos para qué servían y cómo usarlas en el capítulo anterior.

Versiones

Esta característica solo la podemos usar con .NET Framework 3.5 y debemos tener una referencia a System.Linq.dll (todas las plantillas de proyectos de Visual C# 2008 incluyen esa referencia).

(Esta página se ha dejado en blanco de forma intencionada)

Capítulo 9

C# y LINQ (IV)

Introducción

En este capítulo nos vamos a centrar en las características de lo que se conoce como *LINQ to XML*, pero desde el punto de vista del programador de C#. Como seguramente sabrá el lector, Visual Basic se compenetra totalmente con todo lo relacionado con esta tecnología, de forma que incluso permite el uso de literales XML directamente en el código, pero esto en C# no es así, ya que tendremos que usar las clases definidas en el espacio de nombres **System.Xml.Linq** en lugar de literales XML.

LINQ to XML

Esta tecnología proporciona un nuevo sistema de objetos con los que podemos trabajar con datos XML (documentos o elementos) en memoria, a los que podemos aplicar consultas de LINQ.

Para tener acceso a las clases definidas especialmente para trabajar con XML necesitamos una referencia al ensamblado **System.Xml.Linq.dll** el cual define el espacio de nombres **System.Xml.Linq** en el que se encuentran todas las clases que necesitamos para manipular la información XML. Este ensamblado se incluye en los proyectos de Visual C# 2008 que tienen como marco de trabajo la versión 3.5 de .NET Framework.

Debido a que yo soy una persona práctica, me centraré en explicar cómo usar esta tecnología, dejando las características técnicas y otras tecnologías relacionadas (o lo que es lo mismo cómo se trabaja con XML si no usamos las nuevas bibliotecas de .NET Framework 3.5) para que el lector pueda investigar por su cuenta. Aunque esa lectura la tiene a mano en la propia documentación de Visual Studio 2008 o bien en línea en [MSDN](#) en el tópico **Información general acerca de LINQ to XML**.

C# y LINQ to XML

Lo primero que necesitaremos para poder trabajar con *LINQ to XML* es tener en memoria los datos XML que queremos consultar (o manejar). Esto lo podemos hacer de varias formas. La más sencilla es cargar directamente el contenido de un archivo XML, pero incluso para esta tarea tan simple tenemos que saber qué tipo de datos debemos usar para leer la información de ese archivo.

Antes de ver cuáles son las opciones de las que disponemos para cargar el contenido de un archivo XML en la memoria, vamos a crear uno que nos servirá de ejemplo para las distintas pruebas que haremos, y qué mejor forma de crearlo que haciéndolo con código de C#, de esta forma veremos algunas de las clases que podemos usar desde C# para crear documentos XML. En el listado 9.1 vemos el código para crear un documento XML en memoria y guardar el contenido del mismo en un archivo llamado **colegas2.xml**. El contenido de ese archivo es el mostrado en el listado 9.2.

```
// Generado con el AddIn Paste XML as XElement
XElement xml = new XElement("Colegas",
    new XElement("Colega",
        new XElement("Nombre", "Pepe"),
        new XElement("Correo", "pepe@nombres.com"),
        new XElement("Telefono",
            new XAttribute("Tipo", "celular"),
            "666777888"
        ),
        new XElement("Telefono",
            new XAttribute("Tipo", "casa"),
            "952520011"
        )
    ),
    new XElement("Colega",
        new XElement("Nombre", "Luis"),
        new XElement("Correo", "luis@nombres.com"),
        new XElement("Telefono",
            new XAttribute("Tipo", "celular"),
            "677888999"
        )
    ),
    new XElement("Colega",
        new XElement("Nombre", "Eva"),
        new XElement("Correo", "eva@nombres.com"),
        new XElement("Telefono",
            new XAttribute("Tipo", "celular"),
            "688999000"
        ),
        new XElement("Telefono",
            new XAttribute("Tipo", "casa"),
            "977880011"
        )
    ),
    new XElement("Colega",
        new XElement("Nombre", "Luisa"),
        new XElement("Correo", "luisita@nombres.com")
    )
);

XDocument colegas = new XDocument(xml);

// Guardar el fichero .xml
colegas.Save("colegas2.xml");
```

Listado 9.1. Código para generar un documento XML

```
<?xml version="1.0" encoding="utf-8"?>
<Colegas>
  <Colega>
    <Nombre>Pepe</Nombre>
    <Correo>pepe@nombres.com</Correo>
    <Telefono Tipo="celular">666777888</Telefono>
    <Telefono Tipo="casa">952520011</Telefono>
  </Colega>
  <Colega>
    <Nombre>Luis</Nombre>
    <Correo>luis@nombres.com</Correo>
    <Telefono Tipo="celular">677888999</Telefono>
  </Colega>
  <Colega>
    <Nombre>Eva</Nombre>
    <Correo>eva@nombres.com</Correo>
    <Telefono Tipo="celular">688999000</Telefono>
    <Telefono Tipo="casa">977880011</Telefono>
  </Colega>
  <Colega>
    <Nombre>Luisa</Nombre>
    <Correo>luisita@nombres.com</Correo>
  </Colega>
</Colegas>
```

Listado 9.2. Contenido del archivo XML para las pruebas

Un AddIn para convertir código XML en código de C#

Antes de continuar quiero recomendarle al lector que instale un AddIn que se incluye con los ejemplos de Visual Studio 2008 para C# ([CSharpSamples.zip](#) lo puede descargar desde este enlace). Ese AddIn le permitirá convertir código XML (como el del listado 9.2) en el código de C# necesario para crear esa misma estructura de XML (como el del listado 9.1).

El proyecto para crear el AddIn está en la carpeta **LinqSamples\PasteXmlAsLinq** del directorio de ejemplos creado con el contenido del ZIP indicado arriba.

Una vez compilado (dejando los valores predeterminados, al menos no se debe deshabilitar el atributo de ensamblado **ComVisible**), crear una carpeta llamada **Addins** en el directorio **Visual Studio 2008** (que ya existirá) en los documentos del usuario, copiar en esa carpeta los archivos **PasteXmlAsLinq.AddIn** y **PasteXmlAsLinq.dll**. Tendremos que asegurarnos de cerrar Visual Studio, y al abrirlo nuevamente ya estará disponible.

A partir de ahora, en el menú **Editar** habrá una nueva opción (ver la figura 9.1) que nos permitirá pegar el código XML generando los objetos necesarios para crear el código necesario. Esa opción (**Paste XML as XElement**) solo estará disponible cuando haya código XML en el portapapeles. Aunque hay que tener en cuenta que el código original no detecta el lenguaje que estamos usando y siempre se pegará como código de C#. No es perfecto, pero nos facilitará la creación de código de C# a partir de código XML.

Yo he modificado el código de ese complemento para que solo se active esa opción si el lenguaje del documento activo es C#, además si la versión de Visual Studio está en castellano, esa opción se mostrará debajo de la opción **Pegar** (el código original la colocaba al principio del menú **Editar**), y si el idioma actual es español, la opción del menú se mostrará en castellano (**Pegar Xml como XElement**). La captura de la figura 9.1 utiliza el AddIn que yo he modificado. El código de mi versión está incluido en el ZIP con los proyectos de ejemplo del libro.

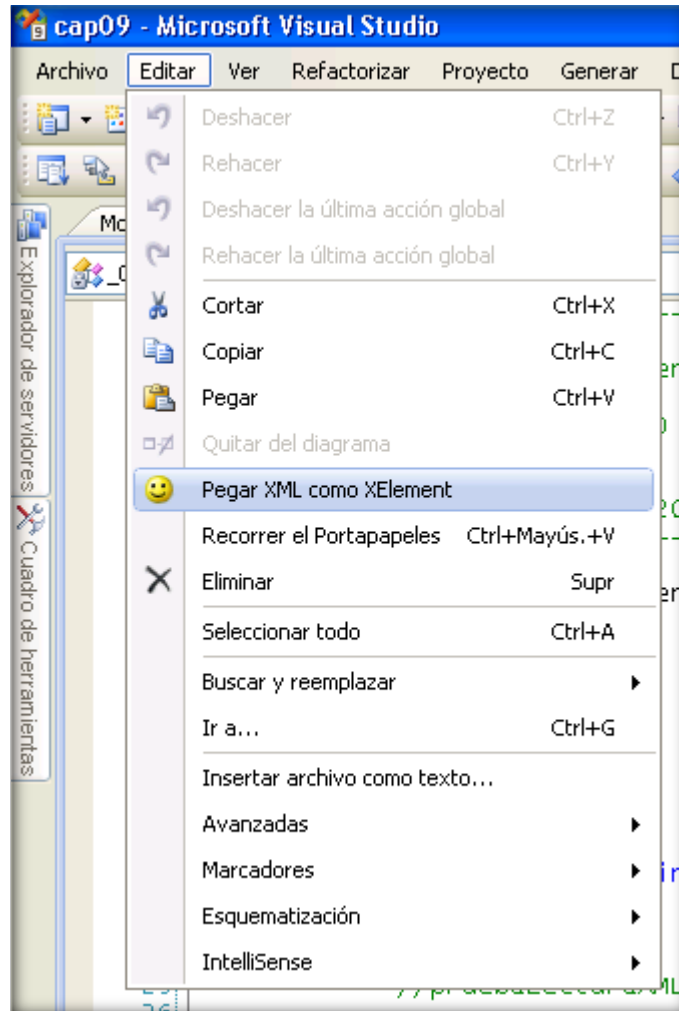


Figura 9.1. Pegar XML como XElement

Clases más usuales de LINQ to XML

Las clases que podemos usar para realizar consultas con *LINQ to XML* son las que están definidas en el espacio de nombres **System.Xml.Linq** y muchas de ellas tienen los mismos nombres que las contenidas en el espacio de nombres **System.Xml**, salvo que las clases de LINQ empiezan con **X**, mientras que las anteriores empiezan por **Xml**, por ejemplo, **XElement** y **XmlElement** ambas representan un elemento XML.

En el listado 9.1 vemos tres de las clases más habituales:

- **XDocument**, representa un documento XML, entre otras cosas, puede contener un elemento que será el nodo raíz.
- **XElement**, representa un elemento XML y puede contener otros elementos o atributos.
- **XAttribute**, representa un atributo de un elemento XML.

La recomendación es utilizar solo **XDocument** cuando sea estrictamente necesario, ya que en la mayoría de los casos, podemos utilizar el mismo código a partir de un objeto **XElement** que contenga el resto de nodos del documento XML. Por ejemplo, el AddIn comentado anteriormente, lo que hace es generar un elemento con todo el contenido. En estos casos, el objeto **XElement** representa al nodo raíz del documento XML.

En el listado 9.3 vemos cómo recorrer los elementos del documento creado en el listado 9.1, para seleccionar todos aquellos que tengan un elemento *Telefono*.

```
var res1 = from c in colegas.Element("Colegas").Elements("Colega")
           where c.Element("Telefono") != null
           select new
           {
               Nombre = c.Element("Nombre").Value,
               Telefono = c.Element("Telefono").Value
           };

foreach(var c in res1)
    Console.WriteLine("{0}, {1}", c.Nombre, c.Telefono);
```

Listado 9.3. Todos los colegas que tienen un elemento Telefono

En el listado 9.4 recorreremos todos los elementos del nodo principal (representado por la variable *xml*), de forma que obtenemos todos los teléfonos y mostramos el contenido de ese elemento, en este caso, la propiedad *Telefonos* del tipo anónimo generado en la consulta es una colección, ya que algunos colegas tienen dado de alta más de un número de teléfono.

```
var res2 = from c in xml.Elements("Colega")
           where c.Element("Telefono") != null
           select new
           {
               Nombre = c.Element("Nombre").Value,
               Telefonos = c.Elements("Telefono")
           };

foreach(var c in res2)
{
    Console.WriteLine("{0}", c.Nombre);
    foreach(var a in c.Telefonos)
        Console.WriteLine(" {0}", a.Value);
}
```

Listado 9.4. Los valores de los teléfonos de los colegas

Si solo queremos el teléfono de uno de los dos tipos (**casa** o **celular**), podríamos (posiblemente complicándonos la vida) hacer que se genere una colección con solo los teléfonos del tipo indicado. En el listado 9.5 vemos cómo podríamos extraer todos los que sean del tipo **casa**.

La complicación, entre otras cosas, es porque he usado algunas de las características que hemos visto en capítulos anteriores, y posiblemente esto se podría hacer de forma más simple.

```
var res3 = from c in colegas.Element("Colegas").Elements("Colega")
let uno = c.Elements("Telefono").Attributes("Tipo").
           FirstOrDefault(t => t.Value == "casa")
where uno != null
select new
{
    Nombre = c.Element("Nombre").Value,
    // uno.Value sería "casa"
    TelefonoCasa = uno.Parent.Value
};

foreach(var c in res3)
    Console.WriteLine("{0} \n {1}",
        c.Nombre, c.TelefonoCasa);
```

Listado 9.5. Solo los teléfonos con el atributo casa

Una forma fácil de crear documentos XML en C#

Antes de seguir viendo cosas relacionadas con *LINQ to XML* desde C#, creo que conviene saber que podemos crear árboles de elementos XML (o documentos XML) en memoria de forma casi parecida a como lo harían los desarrolladores de Visual Basic (como sabemos, ese lenguaje permite usar código de XML directamente sin necesidad de convertirlo usando los constructores de **XElement** o **XAttribute**, tal como vimos en el listado 9.1), en realidad no es más que un pequeño truco usando el método **Parse** de la clase **XElement** y la posibilidad de escribir múltiples líneas en C#. Por ejemplo, la variable *xml* del listado 9.1 (creada con el **AddIn**) la podríamos generar tal como vemos en el listado 9.6. No es perfecto, ya que si tenemos atributos, tendremos que cambiar las comillas dobles por comillas simples (o usar dos comillas dobles), pero... puede ser una alternativa al uso del **AddIn** en muchas situaciones.

```
XElement xml = XElement.Parse(@"
<Colegas>
  <Colega>
    <Nombre>Pepe</Nombre>
    <Correo>pepe@nombres.com</Correo>
    <Telefono Tipo='celular'>666777888</Telefono>
    <Telefono Tipo='\"casa\"'>952520011</Telefono>
  </Colega>
  <Colega>
    <Nombre>Luis</Nombre>
    <Correo>luis@nombres.com</Correo>
    <Telefono Tipo='celular'>677888999</Telefono>
  </Colega>
</Colegas>
```

```
<Colega>
  <Nombre>Eva</Nombre>
  <Correo>eva@nombres.com</Correo>
  <Telefono Tipo='celular'>688999000</Telefono>
  <Telefono Tipo='casa'>977880011</Telefono>
</Colega>
<Colega>
  <Nombre>Luisa</Nombre>
  <Correo>luisita@nombres.com</Correo>
</Colega>
</Colegas>");
```

Listado 9.6. El método Parse de XElement nos permite usar código XML directamente en nuestro código

Acceder al contenido XML en memoria

Como comenté al principio, y según vimos en los ejemplos de los listados 9.2 y 9.3, la forma de acceder a los elementos del código XML que tenemos en la memoria, varía según ese código XML esté en un objeto **XDocument** o en uno del tipo **XElement**.

Si el código XML que queremos consultar está en un objeto del tipo **XDocument** (variable *colegas* del listado 9.1), la forma de acceder a cada uno de los elementos **<Colega>** es recorriendo todos los elementos de ese tipo que descienden del nodo raíz, por eso en el listado 9.2 usamos:

```
from c in colegas.Element("Colegas").Elements("Colega")
```

Es decir, todos los elementos que tengan el nombre **Colega** que estén incluidos en el elemento **<Colegas>**.

Sin embargo, cuando accedemos a esos mismos elementos desde la variable *xml* definida como **XElement**, en realidad el nodo raíz ya lo tenemos accesible (la propia variable *xml*), por tanto, para recorrer todos los elementos **<Colega>** simplemente lo hacemos de esta forma:

```
from c in xml.Elements("Colega")
```

Si no queremos tener que pensar si estamos usando un tipo de objeto u otro, en lugar de recorrer los elementos con el método **Elements**, podemos hacerlo usando el método **Descendants**, de esta forma la manera de acceder a los elementos **<Colega>** es siempre la misma, independientemente de qué objeto contenga nuestro código XML.

En el listado 9.7 cargamos el contenido de un documento XML (guardado con el código del listado 9.1) en una variable del tipo **XDocument** y recorremos el contenido de la misma forma que si la hubiésemos cargado en una del tipo **XElement** (ver listado 9.8). En ambos casos, usamos el método **Load** para cargar el contenido del archivo **colegas2.xml**.


```
XDocument colegasDoc = XDocument.Load("colegas2.xml");  
var res = from c in colegasDoc.Descendants("colega")  
select new  
{  
    Nombre = c.Element("Nombre").Value,  
    Correo = c.Element("Correo").Value  
};  
foreach(var c in res)  
    Console.WriteLine("{0}, {1}", c.Nombre, c.Correo);
```

Listado 9.7. Usando Descendants, la forma de acceder a los nodos es la misma con XDocument o XElement

```
XElement colegasEle = XElement.Load("colegas2.xml");  
var res0 = from c in colegasEle.Descendants("colega")  
select new  
{  
    Nombre = c.Element("Nombre").Value,  
    Correo = c.Element("Correo").Value  
};  
foreach(var c in res0)  
    Console.WriteLine("{0}, {1}", c.Nombre, c.Correo);
```

Listado 9.8. Usando Descendants, la forma de acceder a los nodos es la misma con XDocument o XElement

Acceder a un elemento de la consulta

Como sabemos, cuando generamos una consulta (sea o no de *LINQ to XML*), la variable que recibe el contenido de esa consulta es una colección de tipo **IEnumerable<T>**, pero en realidad esa variable no contiene los datos, solo el código para realizar la consulta (lo que se conoce como ejecución aplazada o ejecución diferida), es decir, hasta que no se enumera esa variable no se ejecuta la consulta. Esto nos puede causar problemas a la hora de querer acceder a uno de los elementos del resultado de dicha consulta (particularmente si no nos damos cuenta de que el resultado está dentro de una colección del tipo **IEnumerable**), ya que podríamos intentar acceder al primer elemento de esa colección con un código parecido al del listado 9.9. Ese código es erróneo, e incluso el propio IDE de Visual C# 2008 nos avisará de que está mal (sin necesidad de compilar el código). Y es erróneo por dos razones, la primera es porque la interfaz **IEnumerable** no define un indexador y la segunda es porque en realidad la variable (*res* en nuestro ejemplo) no tiene elementos.

```
// Esto dará error, ya que
// IEnumerable no define un indexador
var s1 = res[0];
```

Listado 9.9. Intento erróneo de acceder a un elemento de una consulta usando un índice

Una forma de solucionarlo, pasa por ejecutar la consulta (para que haya elementos) y convertirla a un tipo que si defina un indexador, por ejemplo, usando el método **ToArray** o **ToList**, y una vez hecha esa conversión, acceder al elemento que necesitemos (si es que existe en el *array* o colección). En el listado 9.10 vemos cómo convertir el resultado de la consulta del listado 9.7 en un *array* y acceder a la propiedad **Nombre** del primer elemento.

```
var datos = res.ToArray();
var nombre = datos[0].Nombre;
Console.WriteLine("Nombre: {0}", nombre);
```

Listado 9.10. Al convertir en un array la consulta podemos usar un índice para acceder a los elementos

Otra forma sería usando cualquiera de los métodos que ya vimos en el capítulo anterior para operar con los elementos, por ejemplo, si queremos acceder al primer elemento, lo podemos hacer usando el método extensor **First** o bien **ElementAt** indicando que es el elemento en la posición cero. En el listado 9.11 vemos ejemplos de estos dos métodos.

```
var s2 = res.First().Nombre;
Console.WriteLine("Nombre del primero: {0}", s2);

var s3 = res.ElementAt(0).Nombre;
Console.WriteLine("Nombre del elemento 0: {0}", s3);
```

Listado 9.11. Dos formas de acceder al primer elemento de la consulta

Todo esto no está relacionado directamente con *LINQ to XML*, ya que es aplicable a cualquier tipo de consultas aplazadas de LINQ, pero es bueno recordarlo y así no caer en el error de querer acceder a esos elementos. En mi caso, salvo que no lo estime necesario, casi siempre convierto el resultado

de una consulta en una colección del tipo `List<T>` por medio del método extensor `ToList`, de forma que me resulte más fácil acceder a cada elemento individual.

Trabajar con espacios de nombres

Los elementos de un documento XML pueden estar contenidos en un espacio de nombres, en estos casos, a la hora de realizar consultas debemos tener en cuenta que la forma de definir esos elementos cambia con respecto a aquellos que no están incluidos en un espacio de nombres, y al igual que ocurre con las clases de .NET, al referenciar a esos elementos debemos indicar en qué espacio de nombres están.

Para las siguientes pruebas vamos a usar una modificación del archivo de colegas (en el proyecto de ejemplo que acompaña al libro, ese archivo se llama **colegas3.xml**) para que defina dos espacios de nombres. Los tres primeros colegas estarán incluidos en uno y el otro colega estará definido en el otro espacio de nombres. El listado 9.12 muestra el contenido de ese archivo y en el listado 9.13 vemos el código generado por el AddIn **Paste XML as XElement**.

```
<?xml version="1.0" encoding="utf-8"?>
<gi:Colegas xmlns:gi="http://www.elguille.info"
            xmlns:go="http://www.elguille.org">
  <gi:Colega>
    <Nombre>Pepe</Nombre>
    <Correo>pepe@nombres.com</Correo>
    <Telefono Tipo="celular">666777888</Telefono>
    <Telefono Tipo="casa">952520011</Telefono>
  </gi:Colega>
  <gi:Colega>
    <Nombre>Luis</Nombre>
    <Correo>luis@nombres.com</Correo>
    <Telefono Tipo="celular">677888999</Telefono>
  </gi:Colega>
  <go:Colega>
    <Nombre>Eva</Nombre>
    <Correo>eva@nombres.com</Correo>
    <Telefono Tipo="celular">688999000</Telefono>
    <Telefono Tipo="casa">977880011</Telefono>
  </go:Colega>
  <gi:Colega>
    <Nombre>Luisa</Nombre>
    <Correo>luisita@nombres.com</Correo>
  </gi:Colega>
</gi:Colegas>
```

Listado 9.12. Contenido del archivo XML con espacios de nombres

```
XNamespace gi = "http://www.elguille.info";
XNamespace go = "http://www.elguille.org";
XElement xml = new XElement(gi + "Colegas",
    new XAttribute(XNamespace.Xmlns + "gi", gi),
    new XAttribute(XNamespace.Xmlns + "go", go),
    new XElement(gi + "Colega",
        new XElement("Nombre", "Pepe"),
```

```
new XElement("Correo", "pepe@nombres.com"),
new XElement("Telefono",
    new XAttribute("Tipo", "celular"),
    "666777888"
),
new XElement("Telefono",
    new XAttribute("Tipo", "casa"),
    "952520011"
),
new XElement(gi + "Colega",
    new XElement("Nombre", "Luis"),
    new XElement("Correo", "luis@nombres.com"),
    new XElement("Telefono",
        new XAttribute("Tipo", "celular"),
        "677888999"
    )
),
new XElement(go + "Colega",
    new XElement("Nombre", "Eva"),
    new XElement("Correo", "eva@nombres.com"),
    new XElement("Telefono",
        new XAttribute("Tipo", "celular"),
        "688999000"
    )
),
new XElement("Telefono",
    new XAttribute("Tipo", "casa"),
    "977880011"
),
new XElement(gi + "Colega",
    new XElement("Nombre", "Luisa"),
    new XElement("Correo", "luisa@nombres.com")
);
```

Listado 9.13. Código generado por el AddIn Paste Xml as XElement

Si queremos acceder a todos los colegas que estén en el primer espacio de nombres, podemos hacerlo de dos formas, una es incluyendo en la cláusula **where** el espacio de nombres que nos interesa (ver el listado 9.14) y la otra es anexando ese espacio de nombres al nombre del elemento que queremos consultar (ver el listado 9.15), ese espacio de nombres lo hemos creado usando el tipo **XNamespace**.

```
var res = from c in xml.Elements()
where c.Name.Namespace == "http://www.elguille.info"
    && c.Element("Telefono") != null
select new
{
    Nombre = c.Element("Nombre").Value,
    Correo = c.Element("Correo").Value,
    Telefono = c.Element("Telefono").Value
};
```

Listado 9.14. Acceso a los elementos del espacio de nombres indicado (incluyendo el espacio de nombres en la cláusula where)

```
xNamespace ns = "http://www.elguille.info";
var res1 = from c in xml.Elements(ns + "Colega")
where c.Element("Telefono") != null
select new
{
    Nombre = c.Element("Nombre").Value,
    Correo = c.Element("Correo").Value,
    Telefono = c.Element("Telefono").Value
};
```

Listado 9.15. Acceso a los elementos del espacio de nombres indicado (incluyendo el espacio de nombres al seleccionar los elementos)

Ni que decir tiene que si el código XML que estamos consultando incluye los elementos en espacios de nombres, si no tenemos en cuenta ese detalle, no obtendremos ningún resultado, salvo que indiquemos el espacio de nombres al que queremos acceder o bien que recorramos los elementos de forma genérica, es decir, sin filtrar por el nombre del elemento. En el listado 9.16 vemos un código que fallará (no dará error, pero el resultado de la consulta estará vacío), y en el listado 9.17 al no indicar qué elemento queremos obtener de la colección **Elements**, no habrá problemas y obtendremos todos los elementos que tengan un sub elemento llamado **Telefono**.

```
var res2 = from c in xml.Elements("Colega")
where c.Element("Telefono") != null
select new
{
    Nombre = c.Element("Nombre").Value,
    Correo = c.Element("Correo").Value,
    Telefono = c.Element("Telefono").Value
};
```

Listado 9.16. Si los elementos están contenidos en un espacio de nombres, debemos indicarlo

```
var res3 = from c in xml.Elements()
where c.Element("Telefono") != null
select new
{
    Nombre = c.Element("Nombre").Value,
    Correo = c.Element("Correo").Value,
    Telefono = c.Element("Telefono").Value
};
```

Listado 9.17. Esto siempre funcionará, pero si hay distintos tipos de elementos, se recorrerán todos para comprobar si la cláusula where devuelve true

Añadir nuevos elementos al código XML

Otra cosa que seguramente haremos con los árboles XML (o el código XML que llevo diciendo yo todo este rato), será añadir nuevos valores (elementos). La forma de añadirlos también dependerá de que ese elemento pertenezca a algún espacio de nombres. En el listado 9.18 añadimos un par de datos a los colegas cargados desde el archivo **colegas2.xml** (los datos mostrados en el listado 9.2). El listado 9.19 muestra cómo añadir esos mismos datos a los colegas del archivo **colegas3.xml** (los

mostrados en el listado 9.12), pero teniendo en cuenta que usamos un espacio de nombres, en ese ejemplo los añadimos al espacio de nombres del prefijo **gi**.

```
xElement colegas2 = xElement.Load("colegas2.xml");

colegas2.Add(xElement.Parse(@"
    <Colega>
        <Nombre>Pedro</Nombre>
        <Correo>pedro@nombres.org</Correo>
        <Telefono Tipo=""celular"">654123456</Telefono>
    </Colega>")
);

// Un dato sin tipo de teléfono
colegas2.Add(xElement.Parse(@"
    <Colega>
        <Nombre>Ana</Nombre>
        <Correo>ana@nombres.ORG</Correo>
        <Telefono>677999888</Telefono>
    </Colega>")
);

// Añadir un colega sin correo
colegas2.Add(xElement.Parse(@"
    <Colega>
        <Nombre>David</Nombre>
        <Telefono Tipo=""celular"">676654321</Telefono>
    </Colega>")
);
```

Listado 9.18. Añadir dos nuevos elementos al contenido de colegas2.xml

```
xNamespace gi = colegas3.GetNamespaceOfPrefix("gi");

colegas3.Add(
    new XElement(gi + "Colega",
        new XElement("Nombre", "Pedro"),
        new XElement("Correo", "pedro@nombres.org"),
        new XElement("Telefono",
            new XAttribute("Tipo", "celular"),
            "654123456"
        )
    )
);

// Un dato sin tipo de teléfono
colegas3.Add(
    new XElement(gi + "Colega",
        new XElement("Nombre", "Ana"),
        new XElement("Correo", "ana@nombres.ORG"),
        new XElement("Telefono", "677999888")
    )
);

// Añadir un colega sin correo
colegas3.Add(
    new XElement(gi + "Colega",
        new XElement("Nombre", "David"),
        new XElement("Telefono",
```

```
        new XAttribute("Tipo", "celular"),  
        "676654321"  
    )  
);
```

Listado 9.19. Añadir dos nuevos elementos al contenido de colegas3.xml

Como vemos en el listado 9.19, si usamos espacios de nombres, debemos usar esta forma de añadir los datos, para usar el espacio de nombres, lo obtengo del elemento que ha cargado el archivo por medio del método **GetNamespaceOfPrefix** al que le indicamos el prefijo usado, en este ejemplo el prefijo **gi**.

Si intentamos usar el método **Parse** con el prefijo del espacio de nombres (ver listado 9.20) dará error indicando que el espacio de nombres **gi** no está declarado, tal como vemos en la figura 9.2.

```
// Esto así no funciona,  
// dará error de que "gi" no está definido  
colegas3.Add(XElement.Parse(@"  
    <gi:Colega>  
        <Nombre>Pedro</Nombre>  
        <Correo>pedro@nombres.org</Correo>  
        <Telefono Tipo=""celular"">654123456</Telefono>  
    </gi:Colega>");
```

Listado 9.20. Esto dará error indicando que el espacio de nombres gi no está definido

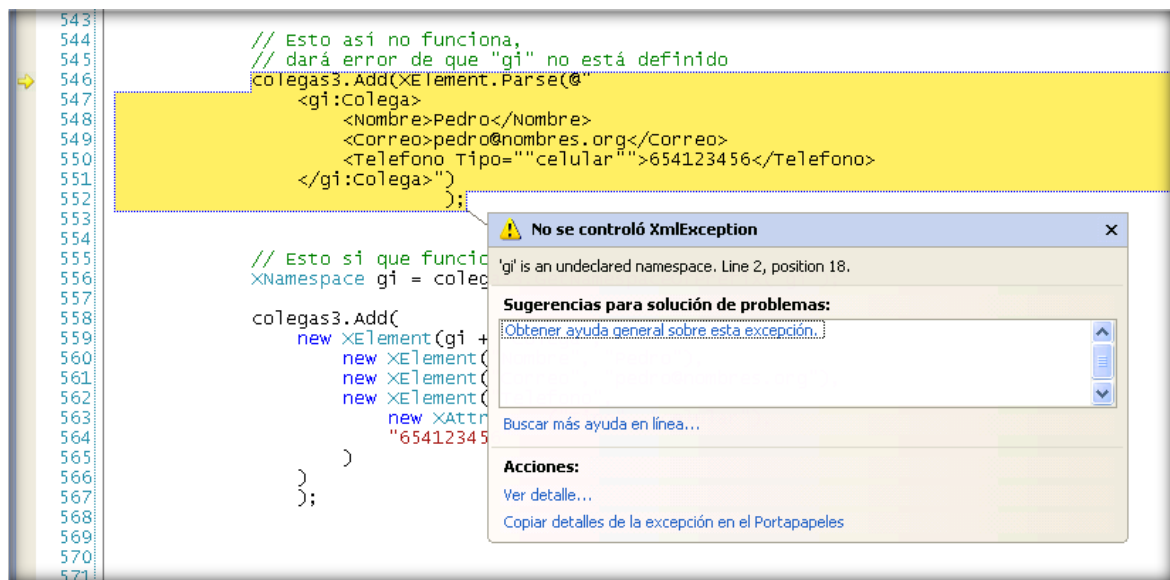


Figura 9.2. Error al usar Parse con prefijos de espacios de nombres

En estos casos, en el que los elementos no tienen todos los datos, si hacemos consultas y seleccionamos los datos, debemos tener en cuenta de que puede que haya elementos que no se encuentren. Por ejemplo, en los datos que hemos añadido, uno de los colegas no tiene correo y el otro en el teléfono no tiene ningún atributo, por tanto estos casos debemos tenerlo en cuenta al hacer la consulta. Lo más fácil es incluir una comprobación extra en la cláusula **where** o bien no utilizar aquéllos elementos que *posiblemente* no estén definidos.

En el listado 9.21 tenemos en cuenta solo los colegas que tengan datos de **Telefono** y **Correo**.

```
var res1 = from c in colegas2.Elements("Colega")
           where c.Element("Telefono") != null
              && c.Element("Correo") != null
           select new
           {
               Nombre = c.Element("Nombre").Value,
               Correo = c.Element("Correo").Value,
               Telefono = c.Element("Telefono").Value
           };
```

Listado 9.21. Solo los elementos que tengan datos de los elementos a usar

En el listado 9.22 solo tenemos en cuenta aquéllos que tengan datos de teléfono con el atributo **celular**, al no utilizar el dato del correo en el tipo anónimo del resultado de la consulta, no hace falta hacer una comprobación de si tiene datos del correo o no.

```
var res2 = from c in colegas2.Elements("Colega")
           let uno = c.Elements("Telefono").Attributes("Tipo").
                   FirstOrDefault(t => t.Value == "celular")
           where uno != null
           select new
           {
               Nombre = c.Element("Nombre").Value,
               TelefonoCelular = uno.Parent.Value
           };
```

Listado 9.22. Podemos filtrar en la consulta los datos que nos interesen

En el listado 9.23 vemos el código equivalente al del listado 9.22, pero teniendo en cuenta los elementos que están en el espacio de nombres indicado.

```
XNamespace ns = "http://www.elguille.info";

var res2 = from c in colegas3.Elements(ns + "Colega")
           let uno = c.Elements("Telefono").Attributes("Tipo").
                   FirstOrDefault(t => t.Value == "celular")
           where uno != null
           select new
           {
               Nombre = c.Element("Nombre").Value,
               TelefonoCelular = uno.Parent.Value
           };
```

Listado 9.23. Código equivalente al listado 9.22, pero teniendo en cuenta el espacio de nombres

Añadir nuevos elementos o atributos a un elemento

Otra de las situaciones en las que podemos querer modificar el código XML que tenemos en memoria, es para añadir nuevos elementos (o atributos) a algún elemento existente.

Por ejemplo, el último elemento que hemos añadido a la colección **colegas2** (o **colegas3**) no tiene datos del correo del colega, por tanto, podríamos hacer algo como lo mostrado en el listado 9.24 para añadir ese elemento (si se modifica la colección con espacios de nombres, ese espacio de nombres solo debemos tenerlo en cuenta a la hora de hacer la consulta, tal como se muestra en el listado 9.25).

```
var unColega = (from c in colegas2.Elements("Colega")
                where c.Element("Nombre").Value == "David"
                select c).FirstOrDefault();

if(unColega != null)
    unColega.Add(
        XElement.Parse(@"<Correo>david@nombres.com</Correo>"));
```

Listado 9.24. Añadir un elemento a un elemento existente

```
var unColega = (from c in colegas3.Elements(ns + "Colega")
                where c.Element("Nombre").Value == "David"
                select c).FirstOrDefault();

if(unColega != null)
    unColega.Add(
        XElement.Parse(@"<Correo>david@nombres.com</Correo>"));
```

Listado 9.25. Añadir un elemento a un elemento existente (con espacios de nombres)

También podemos añadir nuevos elementos usando el método **SetElementValue**. En realidad este método sirve para tres cosas diferentes: añadir, modificar o eliminar un elemento, todo dependerá de los valores indicados en los argumentos. En el primer argumento indicamos el nombre del elemento, si ese elemento no existe, lo añade, si ya existe, lo modifica, asignando el valor del segundo argumento, que si es un valor nulo, sirve para eliminarlo.

En el listado 9.26 vemos otra forma de añadir el elemento **Correo** al elemento devuelto en la consulta realizada en el listado 9.24 (e incluso en el listado 9.25).

```
if(unColega != null)
    unColega.SetElementValue("Correo", "david@nombres.com");
```

Listado 9.26. Otra forma de añadir un elemento

Si queremos añadir un atributo a un elemento, podemos hacerlo de dos formas, la más simple es usando el método **SetAttributeValue** que al igual que **SetElementValue** se puede usar tanto para añadir, cambiar o eliminar un atributo. En el listado 9.27 vemos un ejemplo de cómo añadir el atributo **celular** al elemento **Telefono** de un colega específico, se comprueba que el elemento **Telefono** no tenga atributos.

```
var unTelf = (from c in colegas2.Elements("Colega")
              where c.Element("Nombre").Value == "Ana"
                  && c.Element("Telefono").HasAttributes == false
              select c).FirstOrDefault();

if(unTelf != null)
    unTelf.Element("Telefono").
        SetAttributeValue("Tipo", "celular");
```

Listado 9.27. Añadir (o asignar) un atributo a un elemento

La otra forma sería eliminando primero ese elemento y volver a añadirlo con los datos adecuados, en el listado 9.28 vemos cómo modificar el mismo elemento que contiene la variable *unTelf* del listado 9.27.

```
if(unTelf != null)
{
    var elTlf = unTelf.Element("Telefono").Value;
    unTelf.Element("Telefono").Remove();
    unTelf.Add(
        XElement.Parse(@"<Telefono Tipo=""celular"">" +
            elTlf + "</Telefono>"));
}
```

Listado 9.28. Modificar un elemento (eliminándolo previamente)

Modificar valores de los elementos

En el apartado anterior hemos visto cómo usar los métodos **SetAttributeValue** y **SetElementValue** para realizar cualquiera de las tres tareas de añadir, modificar o eliminar atributos o elementos respectivamente. Por supuesto, si queremos cambiar valores, lo podemos hacer a la vieja usanza, es decir, cambiando el contenido de los valores de esos elementos.

Por ejemplo, si queremos cambiar todas las extensiones de las cuentas de correo que acaben en **.org** por **.com**, podríamos hacer algo como se muestra en el listado 9.29, en el que primero hacemos una consulta para que nos devuelva todos los elementos en los que el valor del elemento **Correo** finalice con **.org** sin importar si está en minúsculas o no, aunque a la hora de asignar lo deja en minúsculas.

```
var res4 = from c in colegas2.Elements("Colega")
            where c.Element("Correo") != null
                && c.Element("Correo").Value.EndsWith(
                    ".org",
                    StringComparison.InvariantCultureIgnoreCase)
            select c.Element("Correo");

foreach(var c in res4)
    c.Value = c.Value.ToLower().Replace(".org", ".com");
```

Listado 9.29. Modificar los valores de los elementos que coincidan con lo buscado

Validar los datos XML

Veamos cómo podemos validar el contenido de un documento XML (los datos que tenemos en memoria o los cargados desde un archivo existente).

En los listados de ejemplo que hemos estado usando hay datos de colegas que no tienen todos los elementos que deberían tener, a saber: **Nombre**, **Correo** y **Telefono**, este último con un atributo llamado **Tipo**. En el caso del elemento **Telefono**, este es opcional (al igual que el atributo **Tipo**), pero los otros dos son obligatorios, por tanto, si quisiéramos usar esos datos, deberíamos comprobar que en realidad son válidos. Esto lo podemos hacer usando la validación XSD.

En el listado 9.30 tenemos el esquema correspondiente al archivo **colegas2.xml**, (este esquema está guardado como **colegas2.xsd** y lo usaremos en el ejemplo que sigue).

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
            elementFormDefault="qualified"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Colegas">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Colega">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Nombre" type="xs:string" />
              <xs:element name="Correo" type="xs:string" />
              <xs:element minOccurs="0" maxOccurs="unbounded" name="Telefono">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:unsignedInt">
                      <xs:attribute name="Tipo" type="xs:string"
                                    use="optional" />
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listado 9.30. Esquema del archivo colegas2.xml

Para validar los datos XML necesitamos varias cosas: un objeto del tipo **XmlSchemaSet** (definido en el espacio de nombres **System.Xml.Schema**), el esquema a usar para la validación y llamar al método extensor **Validate**, que al llamarlo desde un objeto **XDocument** le pasaremos como argumentos el objeto esquema y un método de evento al que se llamará si la validación da error.

En el listado 9.31 vemos cómo validar el contenido de los archivos **colegas2.xml** y **colegas4.xml** (este último contiene los mismos elementos que **colegas2.xml** más los que hemos generado en los listados anteriores). El contenido de **colegas2.xml** es correcto, sin embargo, en el otro archivo, hay datos que son erróneos, por ejemplo, algunos elementos no definen el elemento **Correo**, por tanto, fallará la validación. En el ZIP con el código de ejemplo se incluyen estos cuatro archivos.

```
XDocument colegas2 = XDocument.Load("colegas2.xml");
// Cargar un archivo con datos no válidos
XDocument colegas4 = XDocument.Load("colegas4.xml");

// Crear el objeto con el esquema a validar
XmlSchemaSet esquema = new XmlSchemaSet();
esquema.Add("", "colegas2.xsd");

bool hayError = false;

colegas2.Validate(
    esquema,
    (sender, e) =>
    {
        Console.WriteLine(e.Message);
        hayError = true;
    });

Console.WriteLine("\nEl archivo colegas2.xml ");
if(hayError)
    Console.WriteLine("no se ha validado.");
else
    Console.WriteLine("es correcto.");

Console.WriteLine();

hayError = false;

colegas4.Validate(
    esquema,
    (sender, e) =>
    {
        Console.WriteLine(e.Message);
        hayError = true;
    });

Console.WriteLine("\nEl archivo colegas4.xml ");
if(hayError)
    Console.WriteLine("no se ha validado.");
else
    Console.WriteLine("es correcto.");
```

Listado 9.31. Ejemplos de cómo validar el contenido de archivos XML

LINQ to XML es LINQ

Puede parecer una cosa obvia, pero es bueno recordar que es así, por tanto, con los tipos definidos en el espacio de nombres **System.Xml.Linq** podemos usar todo lo que ya conocemos de LINQ. En los ejemplos anteriores ya hemos usado algunas de las cláusulas definidas en C# además de algunos métodos extensores, y para finalizar, en el listado 9.32 vemos cómo obtener los datos del archivo **colegas4.xml** y mostrar esa información ordenados por el elemento **Nombre**. Como sabemos, algunos datos, no tienen todos los elementos, por tanto, debemos hacer comprobaciones de si se han obtenido o no esos datos, que como vimos, pueden ser a la hora de realizar la consulta o bien a la hora de recorrer los datos devueltos en la consulta, en este ejemplo, esas comprobaciones se hacen a la hora de mostrarlos, pero teniendo en cuenta que puede que esos valores no estén, por tanto, en el tipo anónimo creado en la consulta no damos por sentado que esos datos existen, ya que en nuestro

archivo de datos, sabemos que algunos elementos no definen el elemento **Correo** o que el elemento **Telefono** no define el atributo **Tipo**.

```
xElement colegas4 = xElement.Load("colegas4.xml");  
var res1 = from c in colegas4.Elements()  
           orderby c.Element("Nombre").Value  
           select new  
           {  
               Nombre = c.Element("Nombre").Value,  
               Correo = c.Elements("Correo").FirstOrDefault(),  
               Telefonos = c.Elements("Telefono")  
           };  
foreach(var c in res1)  
{  
    Console.WriteLine("{0}", c.Nombre);  
    if(c.Correo != null)  
        Console.WriteLine(", {0}", c.Correo.Value);  
    Console.WriteLine();  
    foreach(var t in c.Telefonos)  
    {  
        Console.WriteLine(" {0}", t.Value);  
        if(t.HasAttributes)  
            Console.WriteLine(" ({0})", t.Attribute("Tipo").Value);  
        Console.WriteLine();  
    }  
}
```

Listado 9.32. Un ejemplo más de consulta con LINQ to XML

Versiones

Esta característica solo la podemos usar con .NET Framework 3.5 y debemos tener una referencia a System.Linq.dll (todas las plantillas de proyectos de Visual C# 2008 incluyen esa referencia).

Capítulo 10

C# y LINQ (V)

Introducción

En este último capítulo del libro sobre las novedades de C# 3.0 veremos los conceptos más elementales del acceso a datos con la tecnología LINQ. Comprobaremos que básicamente todo lo que ya vimos en los capítulos anteriores de esta parte de C# y LINQ es totalmente válido para realizar consultas de LINQ, pero en lugar de acceder a los datos almacenados en objetos de tipo **IEnumerable** o en documentos XML, esta vez trabajaremos con datos almacenados en bases de datos, y tal como vimos en el capítulo de introducción a esta tecnología, ese acceso a datos podemos hacerlo de dos formas: Usando *LINQ to DataSet* y *LINQ to SQL*, ambas englobadas en lo que se conoce como *LINQ to ADO.NET*, al menos si no tenemos instalado el Service Pack 1 de .NET Framework, ya que con la aparición del Service Pack 1 disponemos de una nueva forma de usar LINQ para el acceso a datos: *LINQ to Entities*.

C# y LINQ to ADO.NET

Para no ser demasiado teórico, he preferido dejar este último capítulo de acceso a datos con LINQ para ir directamente a casos prácticos, ejemplos que nos permitan utilizar las instrucciones de consulta LINQ para lo que “supuestamente” deberían estar más preparadas, al menos por la similitud de esas instrucciones con el lenguaje que casi todos hemos usado alguna vez para acceder a los datos de una base de datos, sea del tipo que sea: el lenguaje SQL (*Structured Query Language* o lenguaje de consultas estructurado).

Como veremos, la utilización de las instrucciones propias del lenguaje para acceder a datos nos facilitará esa tarea que para algunos es, digamos, tediosa y que, al menos en mi caso particular, tenemos que utilizar por “necesidades del guión”. También comprobaremos cómo gracias a los proveedores de LINQ para acceso a datos, toda esa gestión de los datos contenidos en una base de datos, en realidad, nos parecerá de lo más cómodo y fácil de utilizar. Todo lo que hemos visto en los capítulos anteriores nos será de utilidad para esta “otra” forma de crear consultas LINQ, ya que en el fondo, todo el trabajo lo seguiremos haciendo sobre colecciones del tipo **IEnumerable**, al menos cuando trabajemos con la primera de las dos tecnologías en que se divide *LINQ to ADO.NET* y que está más enfocada con el acceso a datos en modo desconectado (*LINQ to DataSet*). Otra forma de acceso a datos, *LINQ to SQL*, en el fondo supone un cambio algo más radical en la forma de acceder a la información contenida en las bases de datos relacionales de SQL Server; aunque gracias a los asistentes (o diseñadores), esa tarea será de lo más cómoda y fácil de “soportar”. En un momento lo veremos, pero antes, pasemos a ver cómo trabajar con *LINQ to DataSet*.

LINQ to DataSet

Empezaremos viendo lo que se conoce como *LINQ to DataSet* o lo que es lo mismo el acceso a datos usando el modo desconectado al que .NET Framework nos tiene acostumbrados desde su aparición hace ya seis años.

Como veremos, la forma de trabajar con los datos no se diferencia de lo que ya hemos aprendido de LINQ en los capítulos anteriores, y por supuesto, el que esta nueva tecnología esté presente no significa que lo que ya sepamos de acceso a datos deba cambiar, porque en este aspecto *LINQ to DataSet* no es intrusivo y no viene a reemplazar el acceso a datos que ya conocemos, solo que ahora nos resultará más intuitivo, y además usando instrucciones propias de C# con toda la ayuda que nos proporciona el entorno de desarrollo, principalmente a través de IntelliSense y la comprobación de errores en tiempo de compilación e incluso antes de llegar a compilar, ya que el entorno de Visual C# 2008 detecta algunos errores conforme vamos escribiendo nuestro código, no llega al nivel de pre compilación de Visual Basic, pero nos advierte de algunos errores, digamos lógicos.

Pero como los datos con los que vamos a trabajar están en una base de datos, lo primero que debemos hacer es acceder a esos datos, traerlos a la memoria y a partir de ahí, usaremos las instrucciones de consultas LINQ que se incluyen en el lenguaje.

La pregunta es: ¿cómo traemos esos datos a la memoria? La respuesta es obvia: si esta forma de realizar consultas se llama *LINQ to DataSet*, ¿qué debemos usar? ¡Efectivamente! ¡Un **DataSet**! Pero no un **DataSet** cualquiera, sino un **DataSet** *tipado* (o si lo prefiere, con establecimiento inflexible de tipos), es decir, de los que podemos crear con el asistente a datos. ¿Por qué? Por una razón muy sencilla, si queremos usar todo lo relacionado con IntelliSense para que podamos saber qué columnas son las que queremos mostrar, en qué columna queremos poner las condiciones para el filtro de los datos, etc., lo lógico es que utilicemos un **DataSet** que esté fuertemente *tipado* con idea de que nos permita saber qué campos (o columnas) tienen los datos.

Nota

Visual Studio 2008 Professional, que es la versión utilizada para realizar los proyectos de este libro, incluye el motor de acceso a datos de SQL Server Express, pero no incluye ningún entorno integrado para la creación y gestión de bases de datos de SQL Server. Por tanto, recomiendo al lector que si no dispone de SQL Server Management Studio, instale la versión Express de ese entorno integrado. A la hora de escribir este libro está disponible la versión 2005 con el Service Pack 2, y puede descargar directamente el instalador (SQLServer2005_SSMSEE.msi) desde esta dirección: <http://go.microsoft.com/fwlink/?LinkId=65110>.

Y si está utilizando la versión de 64 bits, puede descargarlo desde este enlace (en el que se incluye también la de 32 bits): <http://www.microsoft.com/downloads/details.aspx?familyid=6053C6F8-82C8-479C-B25B-9ACA13141C9E&displaylang=es>.

SQL Management Studio Express es válido para usarlo con cualquier versión de Visual Studio 2008, incluso con las versiones Express.

También utilizaremos la base de datos Northwind, la cual se puede descargar desde este enlace: <http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en>.

Por supuesto, podríamos utilizar un objeto **DataSet** “normal”, pero la desventaja es que las columnas y filas que contienen son de tipo **object**, y la verdad es que con un tipo de datos tan elemental poco vamos a aprovechar de la tecnología LINQ. De todas formas, después veremos un ejemplo de cómo realizar consultas de LINQ en tablas que no definen el tipo de datos de la misma forma que las creadas con el asistente.

Por tanto, necesitamos un **DataSet** en el que estén definidos los “tipos” de datos que contiene, con idea de que podamos acceder a ellos de forma fácil por medio de las instrucciones de consulta de LINQ.

Añadir un DataSet al proyecto

Empecemos añadiendo un **DataSet** a nuestro proyecto. Ese objeto tendrá inicialmente la tabla **Employees** de la base de datos **Northwind**, que es la que utilizaremos en nuestro primer ejemplo; después veremos cómo agregar más información a ese **DataSet**.

Creo que a estas alturas de la programación “asistida” ya sabremos crear un **DataSet** con los asistentes, pero por si algún lector ha llegado hasta aquí sin antes haber usado un asistente, le recomiendo que vea el siguiente artículo para crear una conexión a una base de datos y crear un **DataSet** a partir de una tabla de esa base de datos:

<http://blogs.solidq.com/ES/CuevaNet/Lists/Posts/Post.aspx?ID=11>.

Una vez que tenemos definido correctamente el **DataSet** que queremos usar, podemos pasar al código del archivo **Program.cs** de nuestro proyecto de consola y empezar a escribir el código para acceder al contenido del **DataSet** que acabamos de crear.

Crear un adaptador y llenar una tabla

Para poder acceder a los datos tendremos que “llenar” las tablas del **DataSet**, esto lo haremos de la forma tradicional, es decir, creamos un objeto del tipo **SqlDataAdapter** que será el que se conecte con el servidor de SQL Server y nos permita el acceso. Pero, en lugar de crear un objeto de ese tipo, podemos usar el que define el propio conjunto de datos que acabamos de añadir al proyecto; particularmente nos servirá el tipo **EmployeesTableAdapter** definido en el espacio de nombres **NorthwindDataSetTableAdapters**. También creamos un objeto del tipo de la tabla a la que queremos acceder (**NorthwindDataSet**) y a través de esa tabla (después de haberla rellenado con los datos) es como obtendremos los datos por medio de las instrucciones de consulta de LINQ. En el listado 10.1 vemos los pasos preliminares para dejar listos los objetos en memoria para acceder a los diferentes datos que tenemos en el **DataSet**.

```
using NorthwindDataSetTableAdapters;

class Program
{
    static void Main(string[] args)
    {
        EmployeesTableAdapter ta = new EmployeesTableAdapter();
        NorthwindDataSet datos = new NorthwindDataSet();
        ta.Fill(datos.Employees);
    }
}
```

Listado 10.1. Preparamos las variables que necesitaremos para acceder a los datos del DataSet

Consultar los datos de una tabla

La tabla de empleados de la base de datos **Northwind** estará accesible a través de la variable **datos.Employees**, y si queremos realizar una consulta en esa tabla, la podemos hacer tal como vemos en el listado 10.2, en el que indicamos que solo queremos los empleados que en el país (propiedad **Country**) contengan el valor **USA**.

```
var res1 = from e in datos.Employees
            where e.Country == "USA"
            select new
            {
                e.FirstName, e.LastName,
                e.BirthDate, e.Country
            };

```

Listado 10.2. Una consulta LINQ para acceder a los datos de la tabla Employees

El código del listado 10.2 sería el equivalente del listado 10.3, escrito con *Transact-SQL*.

```
SELECT FirstName, LastName, BirthDate, Country
FROM Employees
WHERE Country = 'USA'
```

Listado 10.3. Código de T-SQL equivalente a la consulta del listado 9.2

Recordemos que en las consultas de LINQ la cláusula **select** siempre se indica al final. Esto ya lo sabemos, y también sabemos que la creación de un tipo anónimo también es opcional, pero en este ejemplo en concreto, lo hemos usado, ya que nuestra intención es obtener ciertos campos (o columnas) de esa tabla, si quisiéramos obtener todas las columnas podríamos haberlo hecho indicando después de **select** la variable usada para obtener los datos (variable de rango). ¡Pero cuidado!, si en realidad no necesitamos acceder a todos los datos, ¿para qué queremos traer toda la información? Si le preguntáramos a un experto de SQL, seguramente nos diría lo mismo. Para que lo veamos más claro, indicar solo la variable usada después de **from**, sería el equivalente a crear una consulta de T-SQL en la que después de **SELECT** indicáramos el fatídico * (asterisco), que seguramente ningún experto en SQL nos recomendará. Como seguramente tampoco nos recomendará acceder directamente a una tabla, sino que deberíamos acceder a los datos devueltos por un procedimiento almacenado o una vista (en un momento veremos cómo hacer esto).

Consultas LINQ en DataSet no tipado

La ventaja de usar un objeto **DataSet** que contiene tipos de datos para acceder a la tabla (o tablas) de la base de datos, es que podemos usar IntelliSense para acceder a las columnas que queremos utilizar tanto en la condición como en la selección de los datos a incluir en el resultado de la consulta. Esto lo podemos comprobar en la figura 10.1. Sin esa “inflexión” de tipos no tendríamos toda la ayuda que IntelliSense nos proporciona cuando trabajamos con un entorno de desarrollo como Visual Studio 2008.

Pero si el **DataSet** no define los tipos de datos para acceder a las tablas, tendremos que utilizar algunos de los métodos de extensión de .NET Framework 3.5 para “intentar” el acceso a datos de una forma medianamente razonable, en el sentido de que no perdamos la razón por querer desentendernos de los asistentes de acceso a datos.

En el listado 10.4 vemos cómo crear un objeto **SqlDataAdapter** y un objeto **DataTable** para acceder a la tabla **Employees** de la base de datos **Northwind**. Para utilizar ese código debemos tener importaciones a los espacios de nombres **System.Data** y **System.Data.SqlClient**.

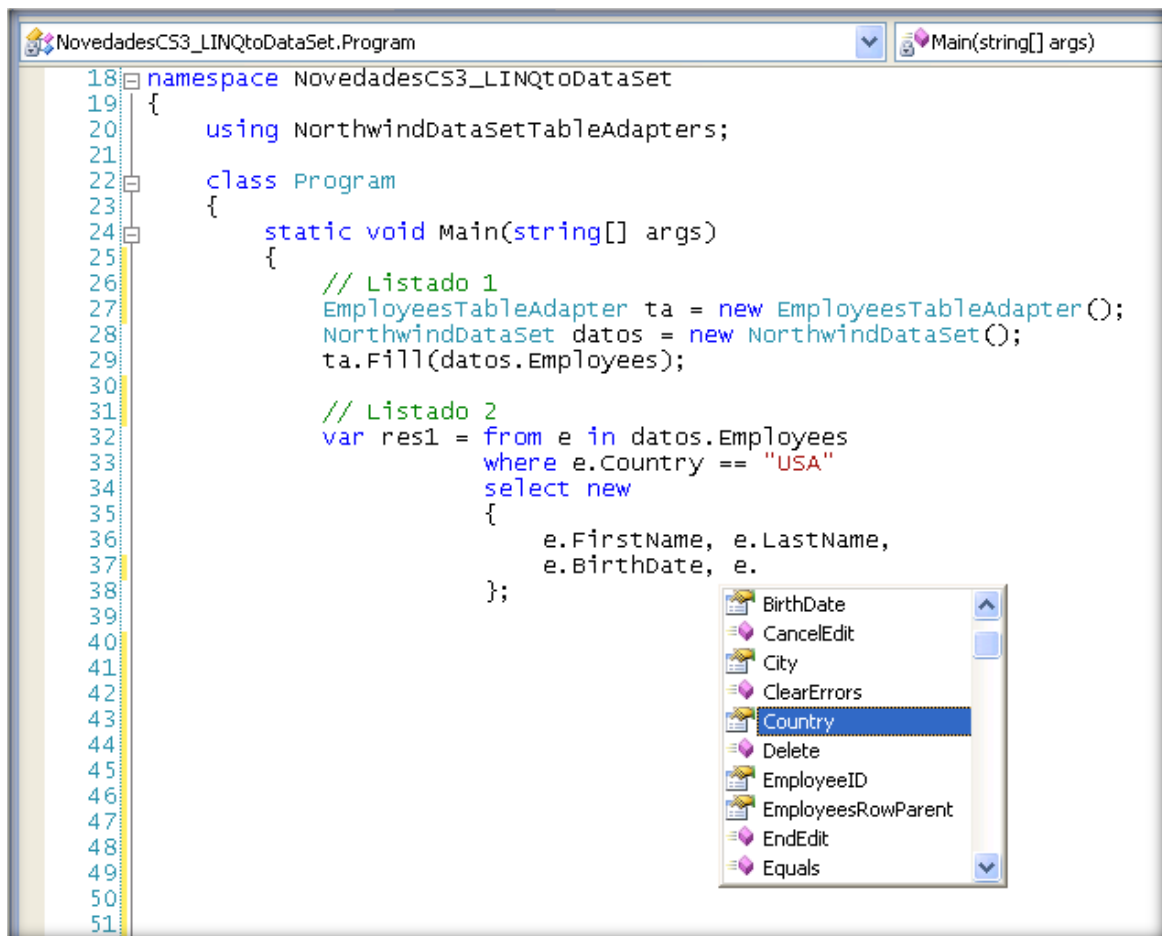


Figura 10.1. IntelliSense es de gran ayuda con los datasets tipados

```
SqlConnectionStringBuilder cs =  
    new SqlConnectionStringBuilder();  
cs.DataSource = @"(local)\SQLEXPRESS";  
cs.InitialCatalog = "Northwind";  
cs.IntegratedSecurity = true;  
  
SqlDataAdapter da =  
    new SqlDataAdapter("SELECT * FROM Employees", cs.ConnectionString);  
  
DataTable dt = new DataTable();  
da.Fill(dt);
```

Listado 10.4. Creación de objetos sin necesidad de usar asistentes de acceso a datos

Hasta aquí todo bien, vamos, que esto sería lo que habitualmente escribiríamos si no queremos utilizar los asistentes de acceso a datos, y por tanto, no tenemos ningún **DataSet** con información de los tipos que hacen referencia a las tablas que contiene. El hecho de utilizar un objeto **DataTable** es por conveniencia, ya que en este ejemplo solo accederemos a una tabla, por tanto, no necesitamos un objeto más complejo.

Si queremos escribir el código de una consulta LINQ es cuando las cosas empiezan a complicarse, ya que no podemos indicar qué columnas queremos utilizar, entre otras cosas, porque no hay ninguna clase que defina las columnas de la tabla como propiedades. Pero no está todo perdido, veamos el código del listado 10.5 para una posible solución.

```
var res1 = from e in dt.AsEnumerable()  
           where e.Field<string>("Country") == "USA"  
           select new  
           {  
               FirstName = e.Field<string>("FirstName"),  
               LastName = e.Field<string>("LastName"),  
               BirthDate = e.Field<DateTime>("BirthDate"),  
               Country = e.Field<string>("Country")  
           };
```

Listado 10.5. Consulta de LINQ para acceder a un objeto de tipo DataTable

Complicado el código del listado 10.5, ¿verdad?

En realidad, podría ser más complicado si no existiera el método extensor **AsEnumerable** para los objetos **DataTable**, pero aún así, es complicada la forma de indicar los nombres de las columnas, ya que debemos especificar manualmente el nombre de la “posible” columna a la que queremos acceder. Y digo posible entrecomillas, porque el que esa columna esté o no definida solo depende de que así lo hayamos indicado en la cadena de selección utilizada para llenar los datos. Además de que este tipo de código es muy propenso a que escribamos mal esos nombres de las columnas, y ese fallo solo lo detectaremos cuando ejecutemos el código, es decir, cuando ya no haya posibilidades de dar marcha atrás.

Como podemos comprobar en el listado 10.5, otro inconveniente de esta forma de acceder a los datos es que en la orden **select** debemos indicar expresamente los nombres de los campos que queremos devolver en la consulta, ya que el compilador no puede inferir los nombres de las propiedades

(del tipo anónimo que se usará como elemento de la colección devuelta), si los valores se obtienen a partir de métodos.

Otro inconveniente de este código es que debemos conocer de qué tipo es cada campo, ya que **Field** es un método extensor de tipo *generic* y siempre debemos indicar qué tipo de datos queremos que devuelva; aunque nada nos impide devolverlos todos como cadena o simplemente como **object**, que sería la alternativa fácil para escribir este tipo de código, tal como vemos en el listado 10.6.

```
var res2 = from e in dt.AsEnumerable()
           where e["Country"].ToString() == "USA"
           select new
           {
               FirstName = e["FirstName"],
               LastName = e["LastName"],
               BirthDate = e["BirthDate"],
               Country = e["Country"]
           };
```

Listado 10.6. Si no nos importa utilizar object, siempre podemos optar por la vía fácil

El código del listado 10.6 también lo podríamos escribir accediendo directamente a la colección **Rows** de la tabla (sin usar el método **AsEnumerable**), aunque en este caso, tal como vemos en el listado 10.7, debemos indicar de forma explícita de qué tipo es la variable de rango.

```
var res3 = from DataRow e in dt.Rows
           where e["Country"].ToString() == "USA"
           select new
           {
               FirstName = e["FirstName"],
               LastName = e["LastName"],
               BirthDate = e["BirthDate"],
               Country = e["Country"]
           };
```

Listado 10.7. Si C# no puede inferir el tipo de la variable de rango, debemos indicarlo explícitamente

En cualquier caso, por favor, no dejemos que la comprobación de si existen esas columnas se haga en tiempo de ejecución, ya que, cualquier error tipográfico, puede dar al traste con todo nuestro trabajo, y nuestros usuarios no se sentirán satisfechos. Así que, hagamos las cosas bien o al menos, intentemos hacerlas lo mejor posible.

Debo reconocer (los que me conocen bien, saben que no me gustan los asistentes de datos), que en esta ocasión está más que justificado el uso de asistentes para la creación de **DataSet** con establecimiento inflexible de tipos (o **DataSet** *tipados*).

Añadir más elementos al DataSet

Una vez que tenemos el **DataSet** con una tabla, podemos agregar otros objetos de la base de datos, por ejemplo, algún procedimiento almacenado o algunas de las vistas que hay definidas. De esa forma veremos cómo utilizar esos “tipos” para crear consultas de LINQ basados en ellos.

Teniendo abierta la ventana del **Explorador de servidores** y el **DataSet** en modo de diseño, podemos arrastrar de la rama de **Procedimientos almacenados** el que tiene el nombre **Employee Sales by Country** al diseñador y posteriormente de la rama **Vistas** el elemento con el nombre **Product Sales for 1997**. Después de agregar estos dos elementos, el **DataSet** tendrá el aspecto de la figura 10.2.

Para un acceso más optimizado (y tipificado), el diseñador del **DataSet** agrega elementos del tipo **SqlDataAdapter** (o al menos, que sirven para el mismo propósito de conectar con la base de datos y recuperar los datos que vamos a manipular de forma desconectada), pero de forma que están definidos para rellenar adecuadamente los tipos de datos creados en el **DataSet**. Esto ya lo vimos en el código del listado 10.1, en el que creamos un adaptador del tipo **EmployeesTableAdapter** para rellenar la tabla de empleados.

Por ejemplo, para utilizar la vista que acabamos de añadir al **DataSet** (las ventas del año 1997), la obtención de datos lo haremos tal como vemos en el listado 10.8.

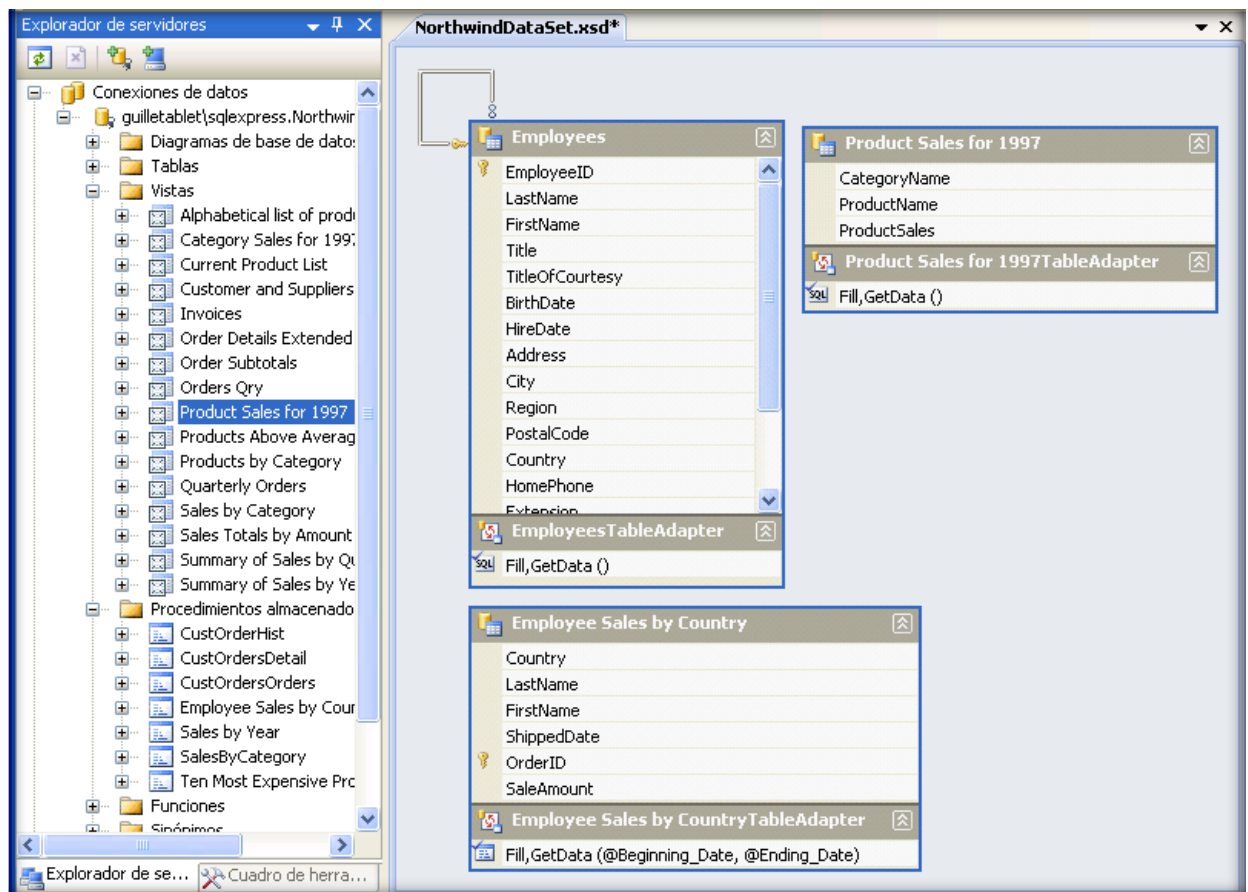


Figura 10.2. El DataSet con nuevos elementos de la base de datos Northwind

```
// Necesitamos una importación a NorthwindDataSetTableAdapters
var ta = new Product_Sales_for_1997TableAdapter();
var datos = new NorthwindDataSet();
ta.Fill(datos.Product_Sales_for_1997);
```

Listado 10.8. Creamos un adaptador adecuado al tipo de datos que queremos utilizar

El método **Fill** del adaptador está definido para aceptar una tabla de un tipo específico, esto evitará que por error utilicemos un objeto de otro tipo.

Y como podemos suponer, la forma de realizar una consulta LINQ para acceder a esos datos es como ya estamos acostumbrados, en el listado 10.9 vemos esa consulta, en la que el compilador infiere adecuadamente el tipo de datos de la variable usada para acceder a cada elemento. En este caso en concreto es del tipo específico creado por el propio diseñador y que nos permitirá acceder a cada una de las filas que la tabla define. Y esas filas tendrán definidas propiedades que representan a cada una de las columnas incluidas en esa vista (si expandimos la rama de la vista **Product Sales for 1997** en el **Explorador de servidores**, veremos que están definidos los tres campos usados en la cláusula **select** del código del listado 10.9).

```
var res1 = from e in datos.Product_Sales_for_1997
            select new
            {
                e.CategoryName,
                e.ProductName,
                e.ProductSales
            };
```

Listado 10.9. Consulta LINQ para acceder a la vista

La personalización del método **Fill** de cada adaptador creado por el asistente tendrá en cuenta los parámetros que se necesite en cada ocasión para acceder a los datos. Por ejemplo, el procedimiento almacenado que hemos añadido al **DataSet** espera dos valores con las fechas a tener en cuenta para mostrar los datos de venta de cada empleado. Por tanto, para llenar ese objeto con los datos adecuados tenemos que indicar el rango de fechas que queremos obtener (si miramos la definición del procedimiento almacenado **Employee Sales by Country**, veremos que define dos parámetros). En el listado 10.10 tenemos el código con la definición de los objetos adecuados para acceder a ese procedimiento almacenado.

```
var fecha1 = DateTime.Parse("15/01/1997");
DateTime? fecha2 = DateTime.Parse("31/01/1997");

var ta = new Employee_Sales_by_CountryTableAdapter();
var datos = new NorthwindDataSet();
ta.Fill(datos.Employee_Sales_by_Country, fecha1, fecha2);
```

Listado 10.10. El método Fill está definido para aceptar los argumentos que necesite el objeto

En el código del listado 10.10, recuperará los datos con las ventas realizadas entre las dos fechas indicadas en las dos variables usadas como argumentos del método **Fill**, pero eso no evita que noso-

tros podamos hacer un nuevo filtro al escribir la consulta de LINQ, tal como vemos en el código del listado 10.11, en el que solo queremos recuperar los datos que sean del día 16.

```
var res1 = from e in datos.Employee_Sales_by_Country
           where e.ShippedDate.Day == 16
           select new
           {
               e.FirstName, e.LastName,
               e.ShippedDate, e.Country
           };
```

Listado 10.11. Podemos hacer filtros extras con los datos obtenidos

Y con este ejemplo concluimos la primera parte de este capítulo dedicado al acceso a datos con LINQ.

LINQ to SQL

En esta parte final de este décimo capítulo veremos cómo utilizar *LINQ to SQL* para acceder a la base de datos **Northwind** y, por medio del diseñador de objetos relacionales (diseñador O/R), crear de forma fácil los objetos con los que queremos trabajar. Pero también veremos cómo acceder a esos datos de una forma algo más manual, es decir, sin usar el diseñador O/R.

Como hemos visto en la sección anterior, cuando trabajamos con *LINQ to DataSet*, en realidad lo que hacemos es lo que hacíamos antes de la llegada de LINQ, es decir, obtenemos los datos, los mantenemos en memoria y es ahí donde los manipulamos, ya que las consultas de LINQ que realizamos, en realidad trabaja con los datos en memoria. Sin embargo, *LINQ to SQL* trabaja de forma diferente. Cuando escribimos una consulta de LINQ, el motor en tiempo de ejecución (ayudado previamente por el compilador) convierte ese código de consulta LINQ en comandos de SQL Server (lo que el motor de SQL Server entiende y sabe manipular), obtiene o manipula los datos en la base de datos, y después recupera esa información y la trae a la memoria en forma de colecciones para que nosotros podamos trabajar utilizando las instrucciones de nuestro lenguaje favorito. En realidad es algo más complicado que todo esto, pero así podemos hacernos una idea del cambio de concepto que supone utilizar *LINQ to SQL* frente a otras formas de acceder a los datos almacenados en bases de datos relacionales.

Modelo de objetos de LINQ to SQL

Cuando trabajamos con *LINQ to SQL* en realidad utilizamos tipos de datos de C# que el compilador y el CLR convierten en objetos de bases de datos. De forma que una clase se relacione con una tabla, las propiedades o campos de esa clase se equiparan a los campos de esa tabla y los métodos serán equivalentes a las funciones y procedimientos almacenados de la base de datos. En la figura 10.3 vemos la equivalencia entre los objetos de .NET y los de la base de datos. Ahora veremos que la relación entre los objetos de programación y los de la base de datos deben tener un “condimento” que será el que en

realidad creará o preparará esa relación entre estos dos modelos de objetos. Ese condimento lo utilizaremos por medio de atributos aplicados a los elementos de programación.



Figura 10.3. Equivalencias entre el código de LINQ to SQL y los objetos de la base de datos

Para comprender mejor esta relación veamos un ejemplo en el que utilizaremos varios campos de la tabla **Employees** de la base de datos **Northwind**.

Lo primero que tenemos que hacer es añadir una referencia en nuestro proyecto al ensamblado **System.Data.Linq.dll** y las dos importaciones de los espacios de nombres mostradas en el listado 10.12.

```
// Necesita una referencia a System.Data.Linq.dll
using System.Data.Linq;
using System.Data.Linq.Mapping;
```

Listado 10.12. Importaciones necesarias para utilizar LINQ to SQL

La primera importación es para poder utilizar los nuevos tipos de datos que necesitaremos para que LINQ establezca una comunicación con la base de datos.

La segunda importación es para poder utilizar los atributos que moldearán nuestras clases y propiedades para relacionarlas con los elementos de la base de datos.

En el siguiente ejemplo vamos a crear una clase para acceder a la tabla **Employees** y de esa tabla accederemos a cuatro de los campos, por tanto, crearemos tantas propiedades (o campos públicos) como columnas de la tabla queramos referenciar en nuestro código. En el listado 10.13 tenemos la definición de esa clase que ligaremos con la tabla de la base de datos.

Como vemos en el código del listado 10.13, para indicar que esta clase se debe relacionar con una tabla, usamos el atributo **Table** y para que cada campo o propiedad de esa clase se relacione con un campo o columna de la tabla, tenemos que usar el atributo **Column**.

```
[Table(Name = "Employees")]
class Empleados
{
    [Column(Name = "FirstName")]
    public string Nombre { get; set; }

    [Column(Name = "LastName")]
    public string Apellidos { get; set; }

    [Column(Name = "BirthDate")]
    public DateTime? FechaNacimiento { get; set; }

    [Column(Name = "Country")]
    public string País { get; set; }
}
```

Listado 10.13. Una clase para acceder a la tabla Employees de Northwind

En ese código, en los nombres de la clase y las propiedades hemos usado nombres diferentes a como están en la base de datos, por tanto, en esos atributos tenemos que indicar los nombres reales que tienen en la base de datos. Si usamos los mismos nombres en la clase y las propiedades (o campos), no es necesario indicar expresamente a qué elementos de la base de datos se refieren, esto lo podemos ver en el código del listado 10.14.

```
[Table()]
class Employees
{
    [Column()]
    public string FirstName;

    [Column()]
    public string LastName;

    [Column()]
    public DateTime? BirthDate;

    [Column()]
    public string Country;
}
```

Listado 10.14. Si los elementos de programación utilizan los mismos nombres que en la base de datos, no es necesario indicar esos nombres en los atributos

Una vez que tenemos la clase “especial” creada, ahora debemos indicarle al compilador que relacione esa clase con el elemento de la base de datos. Esto lo haremos por medio de un objeto del tipo **DataContext**, al que tenemos que indicarle la cadena de conexión que debe utilizar para conectarse con la base de datos, esto lo haremos de la manera habitual, en el listado 10.15 vemos el código.

```
var sCnn = @"Data Source = (local)\SQLEXPRESS; " +  
    "Initial Catalog = Northwind; " +  
    "Integrated Security = True";  
  
var dc = new DataContext(sCnn);
```

Listado 10.15. Creamos un objeto DataContext y lo conectamos con la base de datos

El siguiente paso es obtener la tabla de la base de datos, esto lo hacemos por medio del método **GetTable** del objeto **DataContext**. Ese método, tal como vemos en el listado 10.16, necesita que le pasemos el tipo de datos que hemos definido para trabajar con los datos, en nuestro ejemplo, la clase que definimos en el listado 10.13.

```
var losEmpleados = dc.GetTable<Empleados>();
```

Listado 10.16. Obtenemos la tabla de la base de datos

A partir de este momento, cada vez que utilicemos la variable *losEmpleados*, el compilador sabrá que, en realidad, estamos utilizando la tabla de la base de datos, por tanto, podemos usar esa variable para realizar una consulta LINQ en la que trabajaremos con los datos de esa tabla. En el listado 10.17 vemos la consulta de LINQ para obtener todos los empleados que el país empiece por la letra “U”.

```
var res = from emp in losEmpleados  
    where emp.País.StartsWith("U")  
    orderby emp.Apellidos  
    select emp;
```

Listado 10.17. Una consulta LINQ utilizando la clase definida en el listado 10.13

Como vemos en el código del listado 10.17, debido a que nuestra clase tiene los nombres de las propiedades “castellanizadas”, utilizaremos esos nombres en lugar de los nombres originales definidos en la tabla.

En realidad, esto no es recomendable, ya que es mejor usar los mismos nombres que están definidos en la base de datos, sobre todo para no cometer errores posteriores. Pero nos sirve para que sepamos cómo utilizar esos atributos, que son los que en realidad le dan la información al compilador sobre cuáles son los campos a los que queremos acceder.

Y como podemos suponer, solo podremos acceder a los campos de la tabla que hemos definido en la clase, es decir, si quisiéramos acceder a la columna **EmployeeID**, tendríamos que haber añadido a la clase la propiedad (o campo) correspondiente. Pensemos en que los campos o propiedades de la clase son como los campos/columnas de la tabla que indicamos en una orden **SELECT** de T-SQL: solo se devolverán las columnas indicadas, desechando el resto.

Crear el código automáticamente

Pero no es necesario que tengamos que hacer las cosas manualmente, ya que disponemos de dos herramientas que pueden crear todo el código de C# por nosotros.

Una de esas herramientas es **SqlMetal.exe**, que tenemos que usarla desde la línea de comandos. Por ejemplo, si queremos crear el código de C# con todos los elementos de la base de datos **Northwind**, podemos escribir el código del listado 10.18 para crear un archivo de código llamado **Northwind.cs** en el que estarán todas las tablas, vistas, procedimientos almacenados, etc., que tenga esa base de datos.

```
SqlMetal /server:(local)\sqlexpress /database:northwind /code:Northwind.cs
```

Listado 10.18. Código para utilizar desde la línea de comandos para acceder a la base de datos Northwind

Si agregamos esa clase a nuestro proyecto, podremos acceder a los datos que contiene la base de datos de la misma forma que vimos en los listados 10.15 a 10.17, pero utilizando los nombres tal y como están definidos en la base de datos. Antes de agregar ese archivo o de escribir nuestro código, debemos añadir el ensamblado **System.Data.Linq.dll** a las referencias del proyecto, y para acceder a los datos, por ejemplo, usando el código del listado 10.19, tendremos que agregar una importación al espacio de nombres **System.Data.Linq**, que es donde se define la clase **DataContext** que necesitamos para acceder a la base de datos.

```
var sCnn = @"Data Source = (local)\SQLEXPRESS; " +
    "Initial Catalog = Northwind; " +
    "Integrated Security = True";

var dc = new DataContext(sCnn);

var employees = dc.GetTable<Employees>();

var res1 = from emp in employees
    where emp.Country.StartsWith("U")
    orderby emp.LastName
    select emp;

foreach(var r in res1)
    Console.WriteLine("{0}, {1} {2:dd/MM/yyyy} ({3})",
        r.LastName, r.FirstName, r.BirthDate, r.Country);
```

Listado 10.19. Código para acceder a los tipos creados con la utilidad SqlMetal

Crear las clases usando el diseñador relacional

Seguramente la forma más cómoda de crear estos tipos de datos que necesitamos en nuestros proyectos para acceder a los objetos de las bases de datos es mediante el diseñador de objetos relacional (O/R) incorporado en Visual Studio 2008 (tanto en la versión comercial como en la versión Express).

Para utilizar este diseñador necesitamos dos cosas: la primera es una conexión a una base de datos, pero que esté “visualmente” disponible, es decir, que la tengamos en el **Explorador de servidores**, ya que necesitaremos arrastrar elementos de la base de datos que queremos convertir en clases (sí, parecido a los **DataSet** *tipados*); el segundo requisito, que es el que marca la diferencia a lo que ya vimos en la primera parte de este capítulo, es añadir al proyecto un nuevo elemento del tipo **Clases de LINQ to SQL**, tal como vemos en la figura 10.4. Ese tipo de archivo es el que utilizará el diseñador de objetos relacionales (O/R) para crear los tipos de datos que nos permitan acceder a la base de datos.

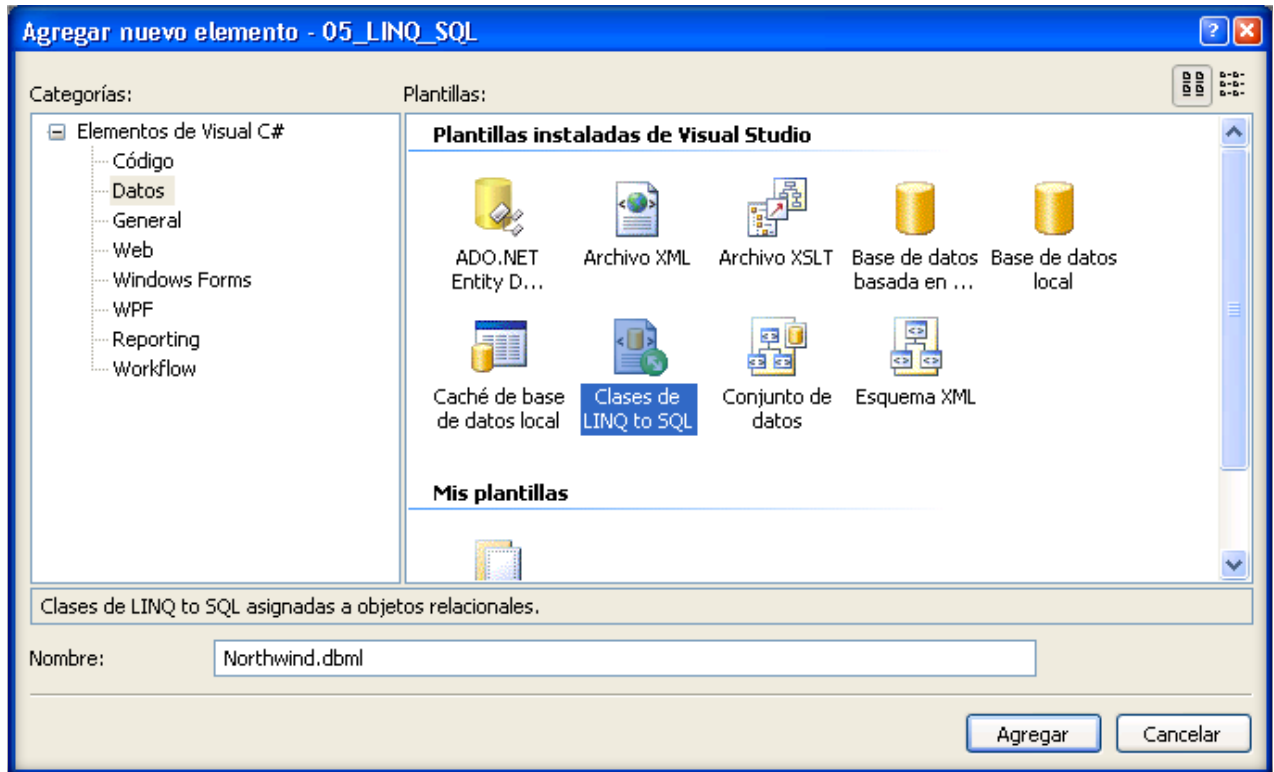


Figura 10.4. Agregar un nuevo elemento del tipo LINQ to SQL

El procedimiento es muy parecido al de crear un **DataSet** a partir de la plantilla **Conjunto de datos**, de hecho, también arrastraremos hasta el diseñador las tablas y demás elementos que queramos utilizar en nuestras clases para *LINQ to SQL*. La diferencia es el tratamiento que se le da a todos los objetos que creemos con este diseñador, incluso la forma de mostrarnos los objetos que vamos agregando al diseñador varía con respecto a lo que ya vimos anteriormente. En la figura 10.5 podemos ver una captura en la que tenemos los mismos elementos que ya vimos en la figura 10.2.

Pero no es solo el aspecto externo lo que cambia, ya que en realidad, aunque algo más complejo, lo que obtenemos gracias a este diseñador es utilizar los objetos de la base de datos de forma totalmente programática, tal como vimos en los ejemplos anteriores, solo que ahora tenemos todas las clases y métodos presentados de una forma más agradable y fácil de manejar.

La forma de utilizar estas clases y elementos es muy parecida a la que vimos en el listado 10.19, la diferencia es que ahora toda la información de conexión con la base de datos, etc., está integrada en las clases que hemos añadido. Esto lo podemos comprobar en el listado 10.20, en el que todos los objetos que utilizamos para acceder a la base de datos los obtenemos a partir de las clases creadas en el proyecto.

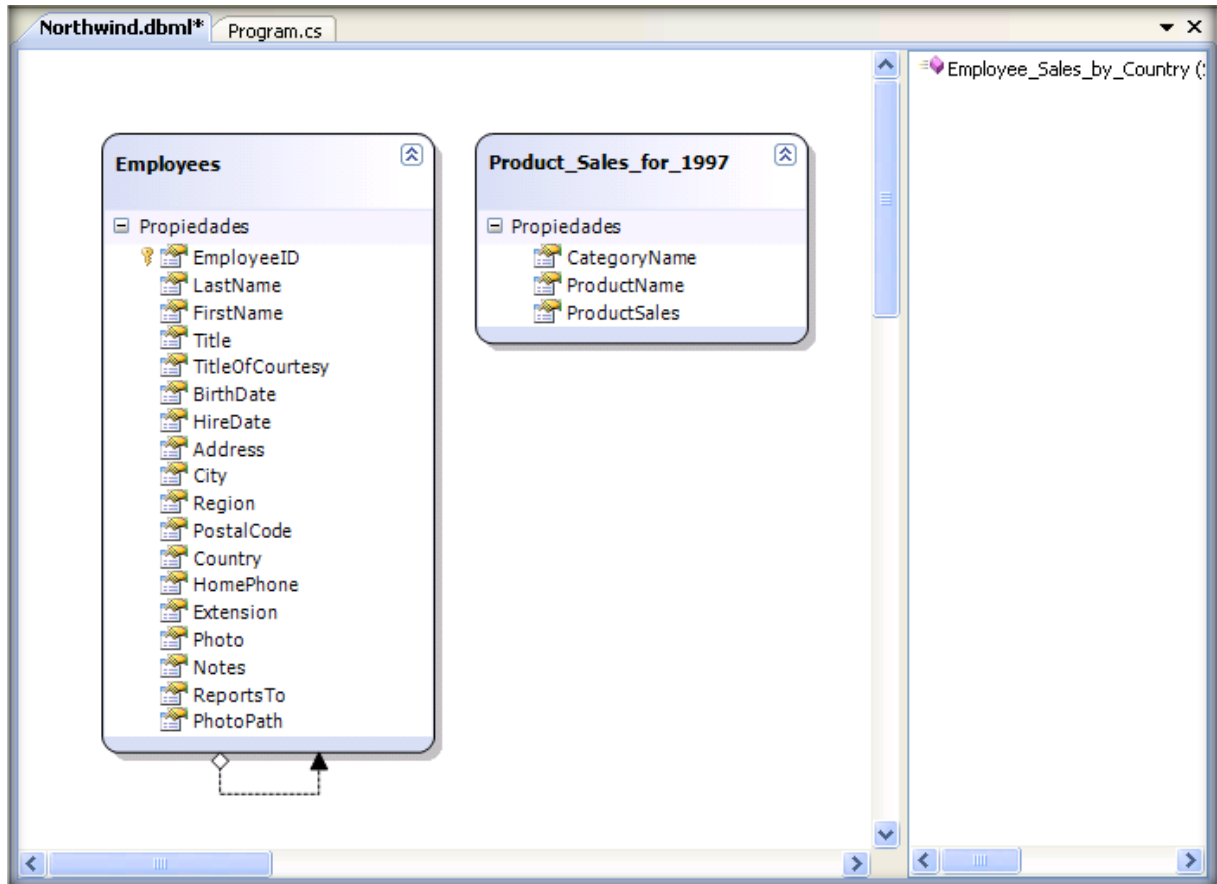


Figura 10.5. Varios elementos de la base de datos en el diseñador de objetos relacionales

```
var db = new NorthwindDataContext();
var losEmpleados = db.GetTable<Employees>();

var res = from emp in losEmpleados
          where emp.Country.StartsWith("U")
          orderby emp.LastName
          select emp;

foreach (var r in res)
    Console.WriteLine("{0} {1}, {2:dd/MM/yyyy} ({3})",
        r.FirstName, r.LastName, r.BirthDate, r.Country);
```

Listado 10.20. La forma de utilizar los objetos creados con O/R es parecida a los ejemplos anteriores

Y si queremos usar el procedimiento almacenado para averiguar las ventas entre dos fechas, lo podemos hacer tal como vemos en el listado 10.21, en el que utilizamos una consulta de LINQ para acceder solo a una parte de los datos devueltos (note el uso de **Value** para acceder a las propiedades de la fecha, ya que en realidad ese tipo de datos es un tipo anulable, concretamente **DateTime?**).

```
var db = new NorthwindDataContext();
var fecha1 = new DateTime(1997, 1, 1);
var fecha2 = new DateTime(1997, 1, 31);

var lasVentas = db.Employee_Sales_by_Country(fecha1, fecha2);

var res1 = from v in lasVentas
            where v.ShippedDate.Value.Day == 16
            select v;

foreach(var v in res1)
    Console.WriteLine("{0}, {1} {2:dd/MM/yyyy} {3:#,##0.00}",
        v.LastName, v.FirstName,
        v.ShippedDate, v.SaleAmount);
```

Listado 10.21. Forma de utilizar el procedimiento almacenado para ver las ventas entre dos fechas

Realmente es todo un mundo lo que nos ofrece el acceso a datos con *LINQ to SQL*, pero entrar en más detalles se escapa de las pretensiones originales de este libro, que es y ha sido explicar las novedades de C# 3.0.

Si el lector quiere más información sobre todo lo que encierra el acceso a datos con *LINQ to SQL* (y como se suele decir en estos casos, y con toda la razón, necesitaríamos otro libro para explicar todo lo que esta tecnología ofrece), le aconsejo que consiga **ese** libro sobre este apasionante tema que mi amigo **Daniel Seara** ha titulado: *Mitos y leyendas de Linq a SQL y otras cuestiones de acceso a datos*, y que también puede conseguir en la Web de **Solid Quality Press**: <http://www.solidq.com/ib/Press.aspx>.

Versiones

Esta característica solo la podemos usar con .NET Framework 3.5 y debemos tener una referencia a System.Data.Linq.dll, esa referencia la tendremos que agregar de forma manual a nuestro proyecto, pero si agregamos un elemento del tipo Clases de LINQ to SQL se agregará la referencia de forma automática.

LINQ to Entities

Con la llegada del Service Pack 1 de Visual Studio 2008 se han agregado algunas novedades a todo lo relacionado con el acceso a datos con LINQ, particularmente un nuevo proveedor de LINQ para ADO.NET conocido como *LINQ to Entities* (que forma parte de *ADO.NET Entity Framework*).

Simplificando mucho, ya que es todo un nuevo mundo lo que hay detrás de *Entity Framework* en general y de *LINQ to Entities* en particular, es parecido a *LINQ to SQL* y al igual que ocurre con esa tecnología el acceso a los datos lo haremos por medio de clases que tendrán su correspondencia con objetos de la base de datos. De hecho, podemos crear las clases que usaremos en nuestros proyectos también de dos formas parecidas a como vimos en la sección anterior: usando una herramienta desde la línea de comandos (**EdmGen.exe**, *EDM Generator*) o usando el diseñador incluido en Visual Studio 2008 Service Pack 1 (*Entity Data Model Designer*). Aunque la forma más fácil de generar las entidades es usando el diseñador de Visual Studio 2008 SP1, y eso es lo que vamos a usar para crear nuestro proyecto de ejemplo.

Crear un proyecto LINQ to Entities

Lo primero que debemos hacer es añadir un nuevo elemento al proyecto, en el cuadro de diálogo de nuevo elemento, en el grupo de **Datos**, seleccionamos **ADO.NET Entity Data Model** (ver figura 10.6), le damos un nombre (o dejamos el propuesto, ya que los nombres que usaremos desde el código no dependen tanto de ese nombre como en otros asistentes de datos) y en cuanto pulsamos en el botón **Agregar** se inicia el asistente.

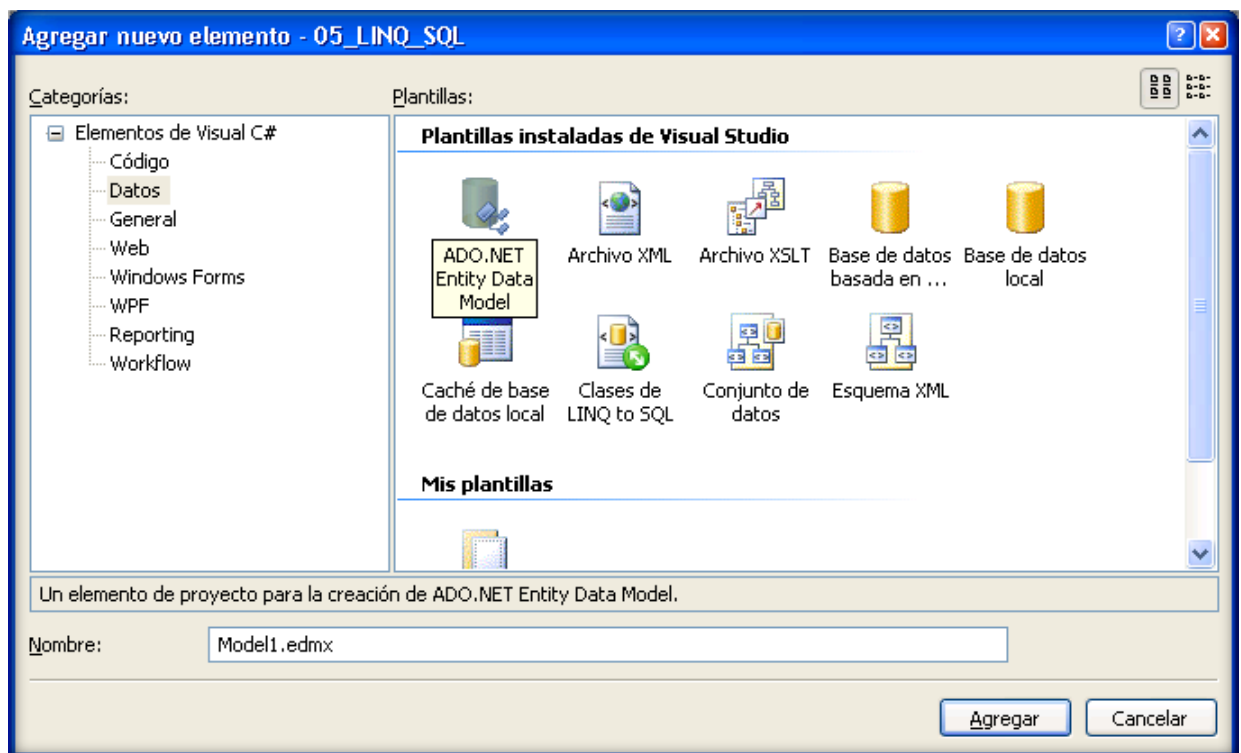


Figura 10.6. Añadir un nuevo elemento Entity Data Model

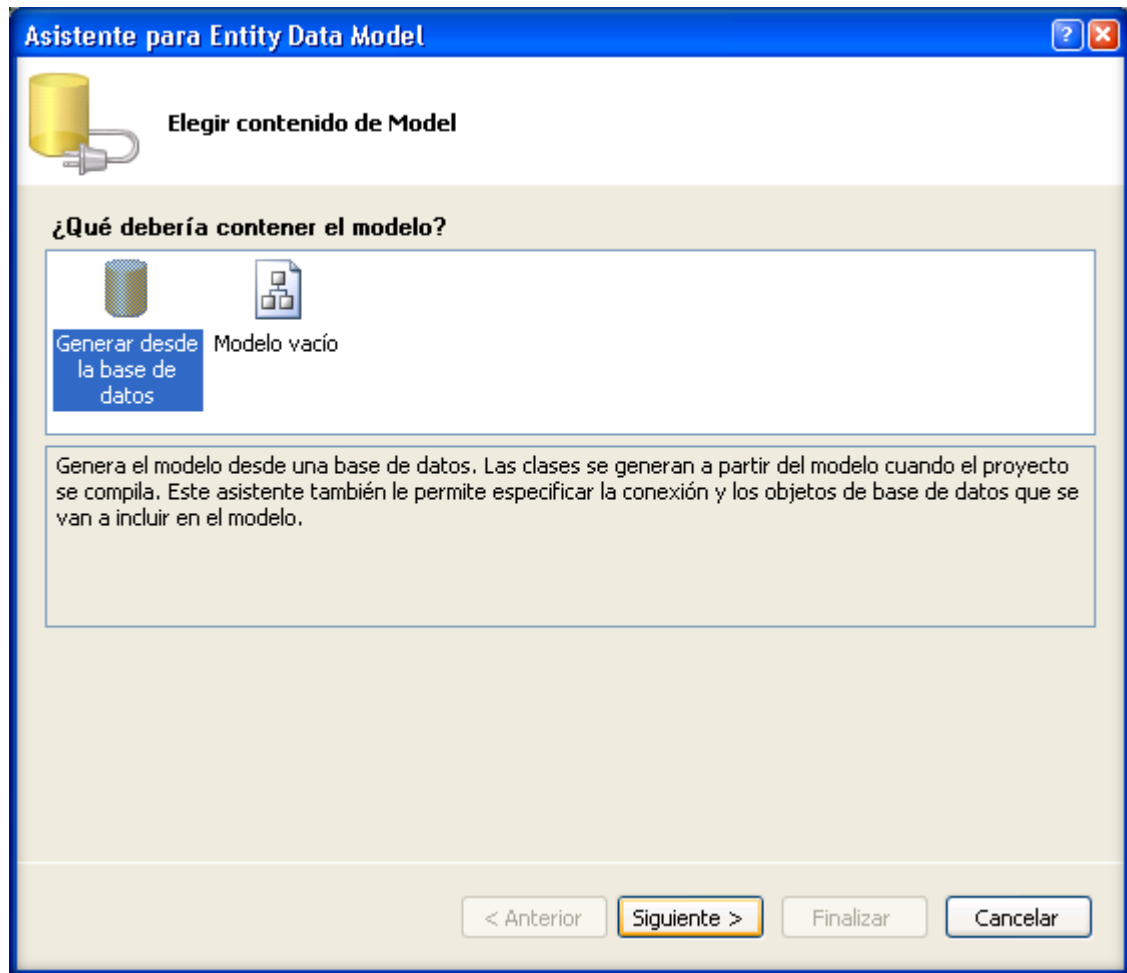


Figura 10.7. Asistente para Entity Data Model, paso 1

En el primer paso de este asistente (figura 10.7) nos pregunta qué debe tener el modelo de datos que vamos a usar, seleccionamos **Generar desde la base de datos**, pulsamos en **Siguiente** y entramos en el segundo paso del asistente, en el que nos pide que indiquemos una conexión, si ya tenemos algunas creadas, se mostrarán en la lista desplegable (ver la figura 10.8) y si no tenemos o queremos usar alguna diferente, podemos seleccionar una nueva conexión, los pasos para seleccionar la conexión es como ya estamos acostumbrados con el resto de asistentes de acceso a datos de Visual Studio, solo que el tipo de las bases de datos que podemos seleccionar, tal como vemos en la figura 10.9, solo son de SQL Server.

En este segundo paso del asistente (figura 10.8), vemos los datos de la cadena de conexión y el nombre de la conexión de entidad, pero a diferencia de lo que ocurre con otros asistentes, no se crea una entrada en la configuración de la aplicación para acceder a esa cadena de conexión.

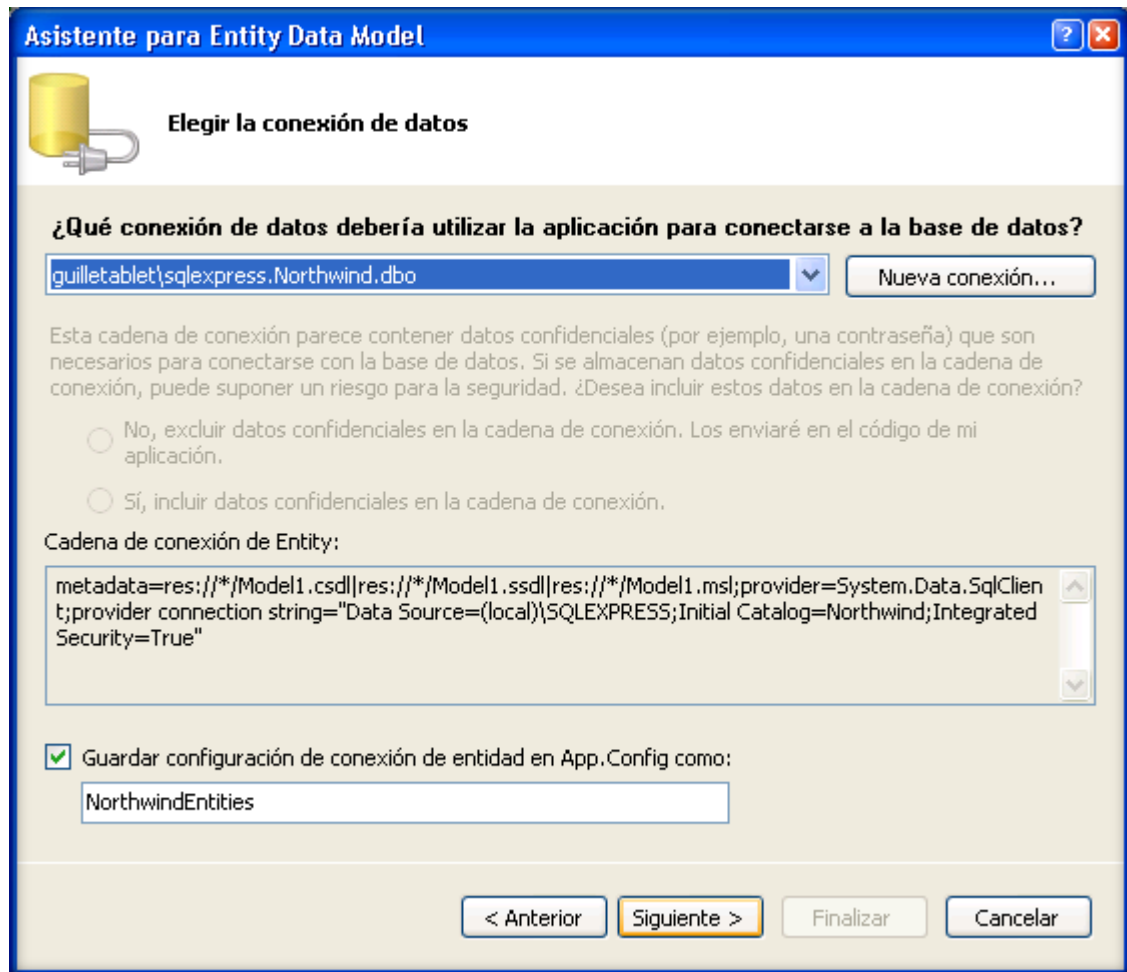


Figura 10.8. Asistente para Entity Data Model, paso 2

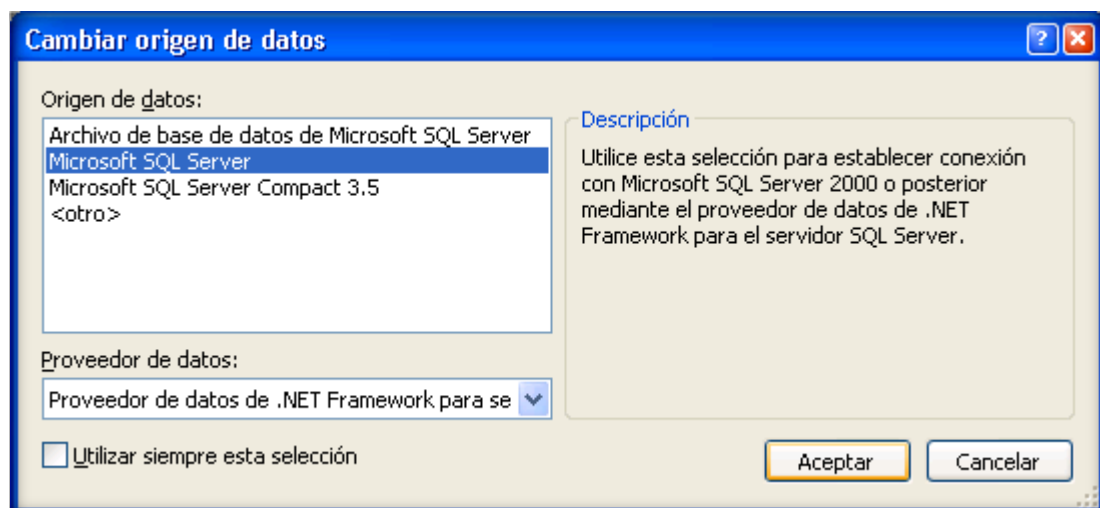


Figura 10.9. Cambiar el origen de datos

En el siguiente paso (figura 10.10), indicamos el nombre del espacio de nombres del modelo (**NorthwindModel** es el propuesto por el asistente), seleccionamos los objetos de la base de datos que queremos incluir en nuestro modelo de entidades, para este ejemplo, he seleccionado los mismos objetos que en los dos casos anteriores: la tabla **Employees**, la vista **Product Sales for 1997** y el procedimiento almacenado **Employee Sales by Country**. Una vez que pulsamos en **Finalizar**, tendremos a la vista el diseñador del modelo de entidades, que tal como podemos comprobar en la figura 10.11 es muy parecido al mostrado por el diseñador de objetos relacionales (ver la figura 10.5), pero con más cantidad de información sobre el objeto seleccionado.

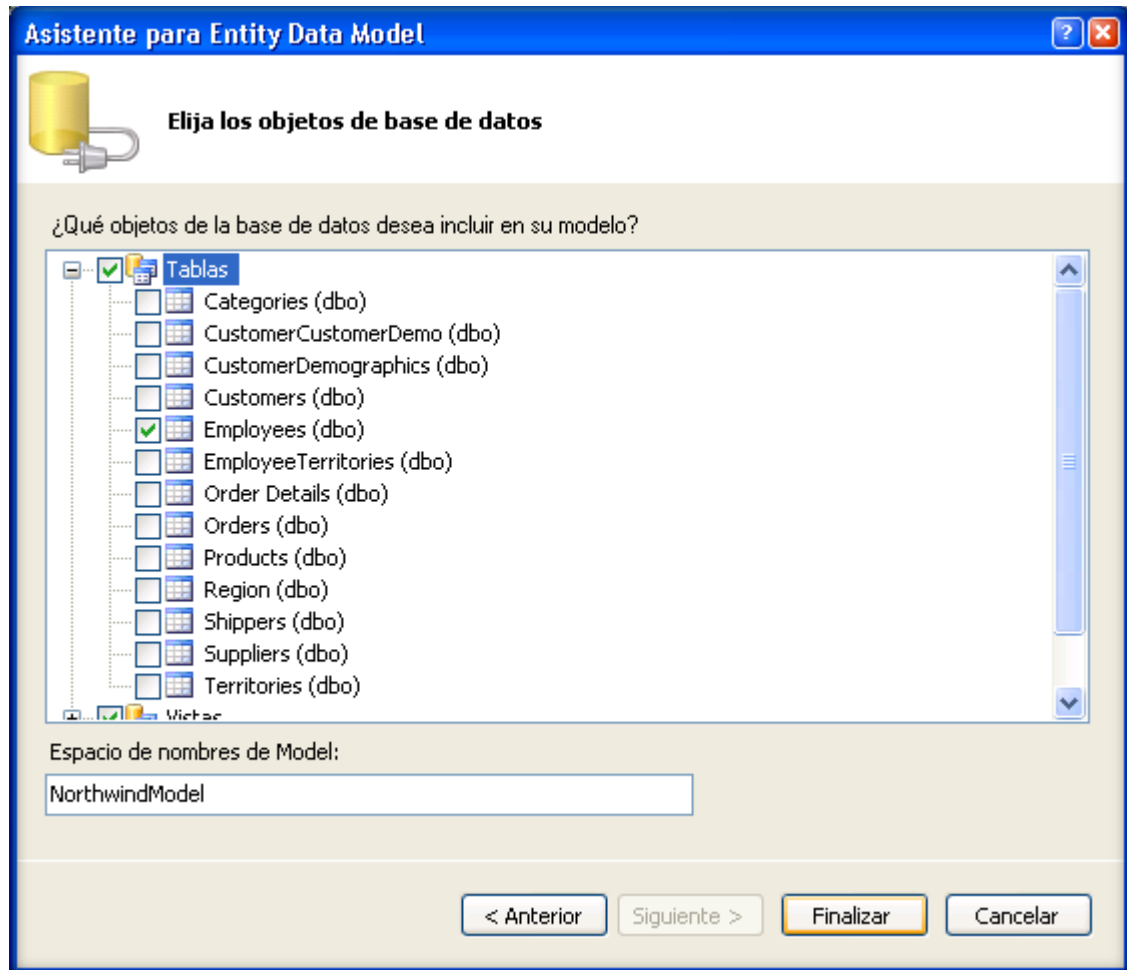


Figura 10.10. Asistente para Entity Data Model, paso 3

En el mismo panel en el que se encuentra el **Explorador de soluciones**, tenemos una ficha con el nombre **Explorador de modelos** (ver la figura 10.12, en esta captura he puesto la ventana para que no esté agrupada con el resto) en el que se muestran los diferentes objetos que tenemos en nuestro modelo de entidades (*Entity Data Model* o EDM).

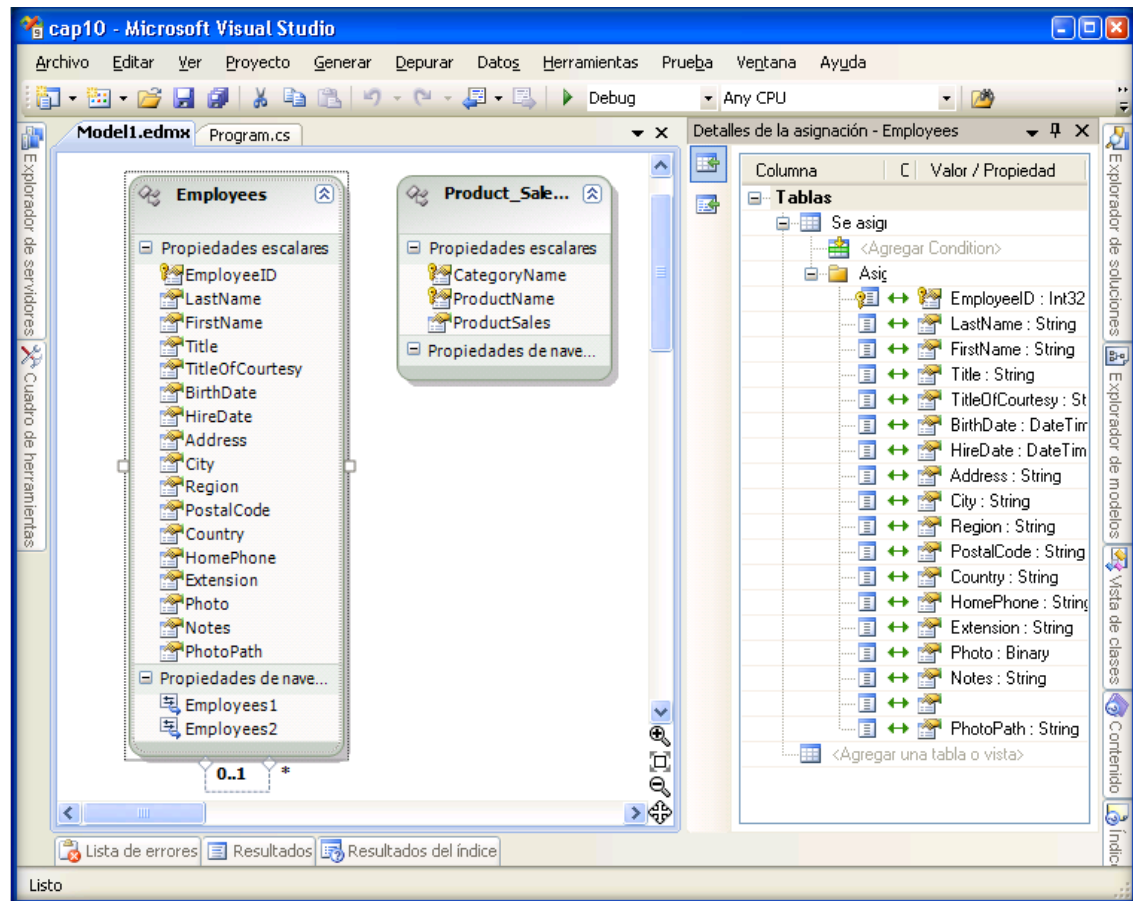


Figura 10.11. Diseño de Entity Data Model

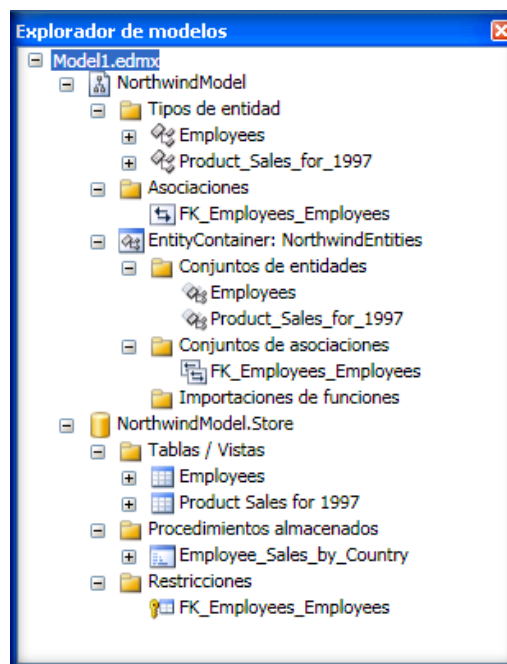


Figura 10.12. Explorador de modelos

En el primer nodo, se muestra el modelo conceptual y en el segundo el modelo correspondiente a la base de datos, es decir, qué tablas, vistas o procedimientos almacenados tenemos mapeados.

Sobre estos últimos, comentar, que a diferencia de lo que ocurre con el diseñador de objetos relacionales de *LINQ to SQL*, los procedimientos almacenados no están listos para usar, y de hecho son más complicados de usar de lo que en un principio pudiera parecer (a pesar de que se puedan crear importaciones de funciones a partir de un procedimiento almacenado, tal como vemos en la figura 10.13).

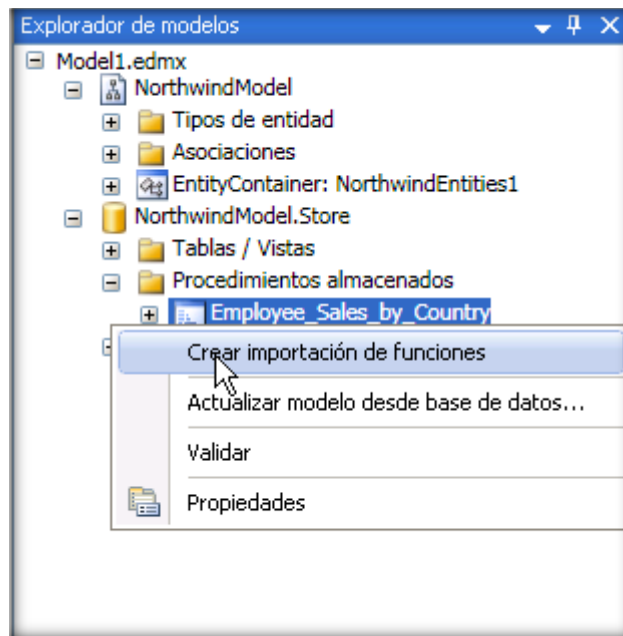


Figura 10.13. Crear importación de funciones

Asociar una función con un procedimiento almacenado

La solución más simple que he encontrado para usar los procedimientos almacenados que devuelven datos, es crear una tabla con los mismos campos que se devuelven en el procedimiento almacenado (los que se indican en la orden **SELECT** del procedimiento), crear una entidad ligada a esa tabla, y usarla como tipo devuelto por la función asociada al procedimiento almacenado.

En este ejemplo concreto, he mirado qué valores devuelve el procedimiento almacenado **Employee Sales by Country**, he creado una tabla en la base de datos **Northwind** con los campos indicados en la orden **SELECT** (ver el listado 10.22 con las instrucciones a ejecutar desde el Management Studio de SQL Server para crear la tabla **Employee Sales by CountryResult**) y he agregado esa tabla al diseñador de entidades. De esta forma, dispondremos de una entidad que podemos asociar como el tipo de datos devuelto por el procedimiento almacenado.

```
USE [Northwind]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Employee Sales by CountryResult] (
    [Country] [nvarchar](15) NULL,
    [LastName] [nvarchar](20) NOT NULL,
    [FirstName] [nvarchar](50) NOT NULL,
    [ShippedDate] [datetime] NULL,
    [OrderID] [int] NOT NULL,
    [SaleAmount] [money] NULL,
    CONSTRAINT [PK_Employee Sales by CountryResult] PRIMARY KEY CLUSTERED
    (
        [OrderID] ASC
    ) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
        IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
        ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Listado 10.22. Código de T-SQL para crear una tabla con los campos devueltos por el procedimiento almacenado Employee Sales by Country

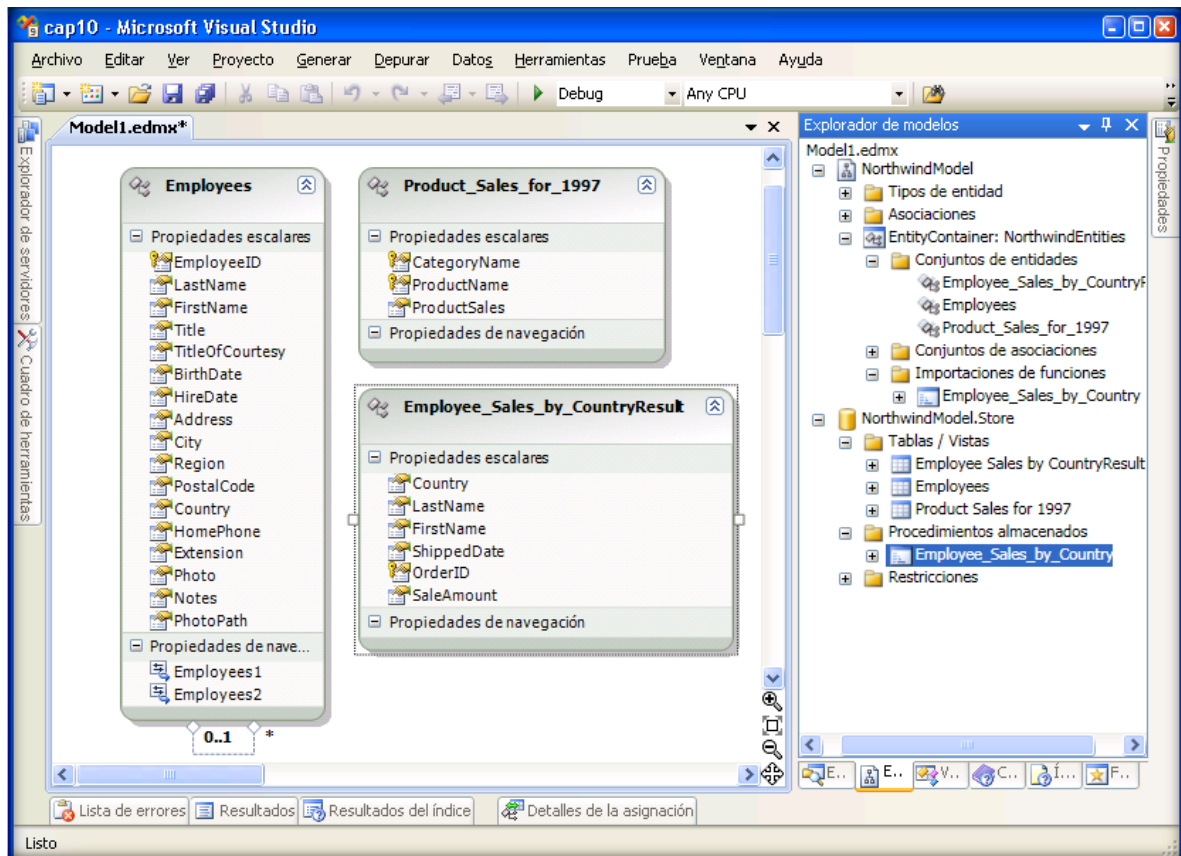


Figura 10.14. El modelo de entidades de los ejemplos

En la figura 10.14 vemos qué entidades tenemos en nuestro modelo y ahora veremos cómo asociar una función con ese procedimiento almacenado.

Cuando importamos una función (ver figura 10.13) se nos muestra un cuadro de diálogo como el de la figura 10.15. En ese cuadro de diálogo debemos indicar qué tipo de datos devuelve el procedimiento almacenado. Y en el caso de que devuelva algo (como es nuestro caso), tendremos que indicarle qué entidad se usará como tipo de datos de cada elemento de la colección devuelta.

Para nuestro ejemplo, vamos a usar la entidad **Employee_Sales_by_CountryResult** (ver la figura 10.16), creada a partir de la tabla de la base de datos que contiene los mismos campos que espera devolver ese procedimiento almacenado (la creada con el código del listado 10.22).

Una vez hecha esta asociación del procedimiento almacenado con una función, lo podremos utilizar desde nuestro código como veremos en la siguiente sección.

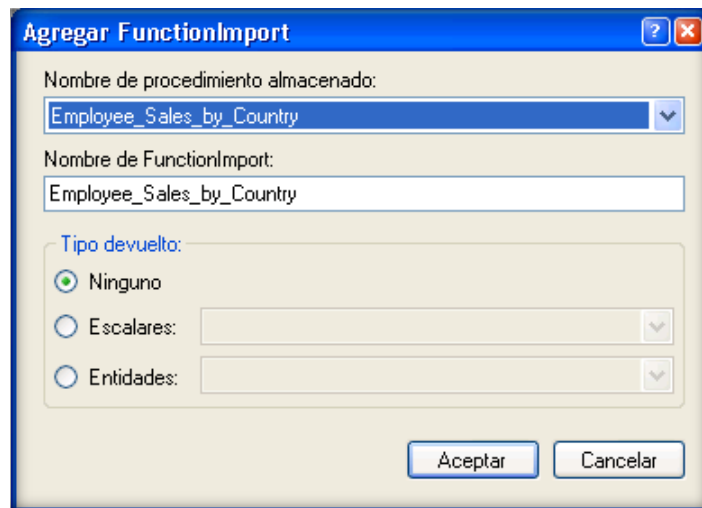


Figura 10.15. Agregar FunctionImport

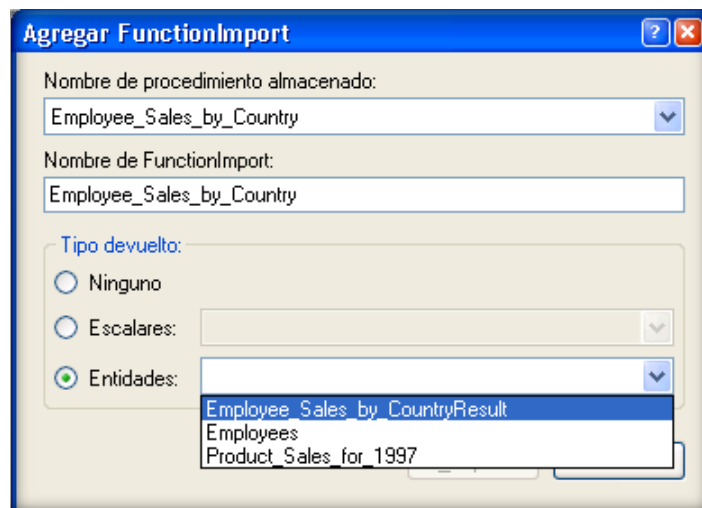


Figura 10.16. Agregar FunctionImport con la entidad a devolver como elemento

Acceder a los objetos creados por el diseñador de Entity Data Model

Una vez que tenemos creados los objetos (entidades) que nos permitirán acceder a los correspondientes objetos de la base de datos, veamos cómo usarlos.

Lo primero que tenemos que hacer es crear un objeto del modelo de entidades, en nuestro ejemplo es *NorthwindEntities* y después solo tendremos que acceder a las propiedades o métodos que se correspondan con los objetos de la base de datos, por ejemplo, para acceder a los datos de la tabla empleados, usaremos el código del listado 10.23 en el que filtramos los datos para que solo se incluyan aquellos empleados que el país empiece con la letra u mayúscula.

```
var db = new NorthwindEntities();  
var losEmpleados = db.Employees;  
var res = from emp in losEmpleados  
          where emp.Country.StartsWith("U")  
          orderby emp.LastName  
          select emp;
```

Listado 10.23. Consulta en los datos de la tabla Employees

Para acceder a la vista de las ventas del año 1997, podemos usar el código mostrado en el listado 10.24, en el que filtramos los datos para que solo muestre aquellos datos en los que el campo **CategoryName** sea igual a **Confections**.

```
var lasVentas97 = db.Product_Sales_for_1997;  
var res1 = from v in lasVentas97  
           where v.CategoryName == "Confections"  
           orderby v.ProductSales descending  
           select v;
```

Listado 10.24. Consulta en la vista Product Sales for 1997

Para acceder al procedimiento almacenado, lo haremos accediendo a la función creada por el diseñador tal como vemos en el código del listado 10.25.

```
var fecha1 = new DateTime(1997, 1, 1);  
var fecha2 = new DateTime(1997, 1, 31);  
var lasVentas = db.Employee_Sales_by_Country(fecha1, fecha2);
```

Listado 10.25. Consulta en el procedimiento almacenado (función)

Algo que debemos tener en cuenta, es que si recorremos los elementos de la consulta guardada en la variable *lasVentas* y después pretendemos volver a usarla, por ejemplo para mostrar solo las ventas de un día determinado, tal como vemos en el listado 10.26, al intentar procesar el segundo bucle, recibiremos un error indicándonos que el resultado de una consulta no se puede enumerar

más de una vez, la explicación "técnica" es porque este tipo de colecciones (**ObjectResult** en *LINQ to Entities* o **ISingleResult** en el caso de *LINQ to SQL*), libera el enumerador una vez enumerado el contenido, o como indica la documentación de Visual Studio: *contiene una secuencia de retorno única*.

```
foreach(var v in lasVentas)
    Console.WriteLine("{0}, {1:dd/MM/yyyy} {2,10:##0.00}",
        v.OrderID, v.ShippedDate, v.SaleAmount);

Console.WriteLine("Solo las ventas del día 16:\n");

var res2 = from v in lasVentas
            where v.ShippedDate.Value.Day == 16
            select v;

foreach(var v in res2)
    Console.WriteLine("{0}, {1} {2:dd/MM/yyyy} {3:##0.00}",
        v.LastName, v.FirstName,
        v.ShippedDate, v.SaleAmount);
```

Listado 10.26. Las consultas generadas a partir de un procedimiento almacenado solo la podemos recorrer una vez, por tanto, este código fallará

La solución para poder utilizar esos resultados más de una vez (sin tener que volver a llamar al método asociado al procedimiento almacenado) es crear una colección (o un *array*) con los resultados, de esa forma, tal como vemos en el listado 10.27, podremos usar esa nueva colección tantas veces como consideremos oportuno.

```
var fecha1 = new DateTime(1997, 1, 1);
var fecha2 = new DateTime(1997, 1, 31);

var lasVentas = db.Employee_Sales_by_Country(fecha1, fecha2);

// Si queremos enumerar este resultado más de una vez
// debemos crear una colección "normal"
var lasVentas2 = lasVentas.ToList();

foreach(var v in lasVentas2)
    Console.WriteLine("{0}, {1:dd/MM/yyyy} {2,10:##0.00}",
        v.OrderID, v.ShippedDate, v.SaleAmount);

Console.WriteLine("Solo las ventas del día 16:\n");

var res2 = from v in lasVentas2
            where v.ShippedDate.Value.Day == 16
            select v;

foreach(var v in res2)
    Console.WriteLine("{0}, {1} {2:dd/MM/yyyy} {3:##0.00}",
        v.LastName, v.FirstName,
        v.ShippedDate, v.SaleAmount);
```

Listado 10.27. Creando una colección a partir de la consulta, la podremos enumerar más de una vez

Y con esto damos por finalizado este repaso a las tecnologías de acceso a datos con LINQ y por extensión a este libro sobre las novedades de C# 3.0.

Versiones

Esta característica solo la podemos usar con .NET Framework 3.5 Service Pack 1 y debemos tener una referencia a System.Data.Entity.dll y System.Runtime.Serialization.dll, esas referencias la tendremos que agregar de forma manual a nuestro proyecto, pero si agregamos un elemento del tipo ADO.NET Entity Data Model se agregarán de forma automática.

Finalmente recordar al lector que todos los proyectos de ejemplo usados en este libro (tanto los mostrados como los no mostrados), están disponibles en el sitio Web del libro:

<http://www.solidq.com/ib/DownloadEbookSamples.aspx?id=2>.

Espero que haya disfrutado de la lectura tanto o más como yo en escribirlo.

Nos vemos.
Guillermo

Índice alfabético

Símbolos

& (AND), 113, 130
&& (AND), 113, 130
| (OR), 113, 130
|| (OR), 113, 130
#if, véase Métodos parciales
#endif, véase Métodos parciales
=> (operador lambda), véase Lambda

A, B

Action, 64, véase Lambda
Addin, 159
Aggregate, 133, 152
Aggregate (cláusula VB), 133
All, 108, 112, 138
Any, 108, 112, 138
Array (clase), 68, 100, 152
ascending, 102, 119, 136
AsEnumerable, 148, 182
AsQueryable, 148
Atributos XML, véase LINQ to XML
Average, 108, 110, 152
bool, 86, 112, 142

C

Cast, 148
CLR, 19, 22, 79, 186
Column (atributo), 188
Common Language Runtime, véase CLR
CompareTo, 67
Comparison, 68
ComVisible, 159
Concat, 151
Conjunto de datos, véase DataSet
Consultas, véase LINQ
Contains, 138
Contravarianza, 67
Count, 108, 110, 152

D

DataContext, 188
DataSet, 97, 105, 177, 178, 181, 191
DataTable, 181
DateTime, 182, 185
DateTime?, 185, 193
DefaultIfEmpty, 140
Delegado, 34, 42, 59, 60, 64, 88
delegate, 59, 89
Descendants, 163
descending, 102, 119, 136
Diseñador relacional de objetos, 97, 190
Distinct, 117, 136
Dictionary, 150
Documentos XML, véase LINQ to XML

E

Element, 161
ElementAt, 143, 146, 165
ElementAtOrDefault, 143, 146
Elementos XML, véase LINQ to XML
Elements, 161, 163, 168
Employees, véase Northwind
Empty, 140
Enumerable, 141, 149
Entity Data Model, véase LINQ to Entities
Entity Framework, véase LINQ to Entities
equals, 108
Equals, 53, 58, 142
Except, 136
Explorador de modelos, véase LINQ to Entities
Explorador de servidores, 191, 198
Expresiones lambda, véase Lambda
Extension (VB), 78

F

Field, 182
Fill, 180, 185
First, 143, 146, 165
FirstOrDefault, 143, 146
from, 98, 101, 106

Func, 64, 143
Funciones anónimas, véase Lambda
Funciones de agregado, 108, 131, 152
Funciones en línea, véase Lambda

G, H

GetHashCode, 53, 142
GetNamespaceOfPrefix, 170
GetTable, 189
group by, 121, 124, 139
GroupBy, 121, 124, 139
GroupJoin, 121, 123, 139
Herramientas (menu), 29

I, J

IComparable, 67
IComparer, 67
IDE, 17, 22, 27, 43, 77, 165
IEnumerable, 98, 108, 117, 131, 135, 148, 165, 177
IEqualityComparer, 142
Importar y exportar configuraciones, 29
in, 98, 106, véase from
Inferencia de tipos, 33, 38, 56
Inicialización de arrays, 34, 38
Inicialización de colecciones, 33, 36, 38
Inicialización de objetos, 33, 36, 38, 50, 93
INNER JOIN, 121
ISingleResult, 203
Integrated Development Environment, véase IDE
Intersect, 136
InvalidOperationException, 146
IQueryable, 135, 138, 149
join (cláusula), 106, 121, 123, 139
Join, 139

L

Lambda, 49, 59, 61, 65, 72
Last, 143, 147
LastOrDefault, 143, 147
LEFT, 95
LEFT JOIN, 121
let, 128
LINQ, 19, 31, 49, 58, 72, 91, 97, 98, 105
LINQ to ADO.NET, 97, 177
LINQ to DataSet, 97, 178

LINQ to Entities, 97, 194
LINQ to Objects, 97, 98
LINQ to SQL, 97, 186
LINQ to XML, 97, 157, 175
List, 38, 67, 93, 100, 149
Load, 163
LongCount, 108, 110, 152

M

Max, 108, 110, 152
Métodos de extensión, véase Métodos extensores
Métodos extensores, 75, 80, 83, 87, 90, 108, 131
Métodos parciales, 40, 43
Min, 108, 110, 153
Module (VB), 78
MSDN, 157

N

new[], 34
Northwind, 178, 186, 199
NotInheritable (VB), 90
null, 77, 140

O

O/R, véase Diseñador relacional de objetos
object, 49, 77, 179
ObjectResult, 203
OfType, 137, 148
Option Infer (VB), 71
Option Strict (VB), 71
orderby, 102, 119, 124, 135
OrderBy, 136
OrderByDescending, 136

P, Q

PadFillLeft, 95
params, 58, 72, 85
Parse, 162, 170
PasteXmlAsLinq (addin), 159
Parent, 162
partial, 41
Propiedades autimplementadas, 45
Queryable, 149

R

Range, 140
Repeat, 140
Resize, 100, 152
Reverse, 120, 135
Rows, 183
Runtime, 18, 68, 72, véase CLR

S

sealed, 90
select (cláusula), 95, 98, 100, 105, 112, 117, 122, 180
Select (método), 138
SelectMany, 138
SetAttributeValue, 172
SetElementValue, 172
Single, 143, 147
SingleOrDefault, 143, 147
Skip, 125, 139
SkipWhile, 126, 139
Sort, 67
SQL, véase T-SQL y LINQ to SQL
SqlDataAdapter, 179, 184
SqlMetal.exe, 190
StartsWith, 95
Sum, 101, 108, 110, 123, 132, 153
System.Core, 76
System.Data, 181
System.Data.Entity, 204
System.Data.Linq, 91, 187
System.Data.SqlClient, 181
System.Linq, 103, 130, 176
System.Xml.Linq, 157, 175

T

Table (atributo), 188
Take, 125, 139
TakeWhile, 126, 139

ThenBy, 136
ThenByDescending, 136
this, 76, véase Métodos extensores
Tipos anónimos, 49, 53, 56, 67, 101, 117
ToArray, 100, 148, 165
ToDictionary, 148
ToList, 57, 93, 100, 148, 165
ToLookup, 139, 148
ToString, 50, 118
Trim, 84, 86
T-SQL, 95, 106, 121, 180

U, V

UAC, 27
Union, 136
User Account Control, 27
Validate, 174
var, 33, véase Inferencia de tipos

W

where (cláusula), 95, 98, 106, 113, 137
Where (método), 137
Windows Presentation Foundation, 19
Windows Vista, 18, 27
WPF, 19
WriteLine, 95

X

XAttribute, 161
XDocument, 161
XElement, 161
XML, véase LINQ to XML
XmlSchemaSet, 174
XNamespace, 167
Xsd, véase LINQ to XML
Xsl, véase LINQ to XML

Aprenda C# 3.0 desde 0.0 - Parte 3, lo nuevo



Guillermo "Guille" Som

Es conocido en Internet por su portal dedicado exclusivamente a la programación, principalmente con Visual Basic y todo lo relacionado con punto NET, (<http://www.elGuille.info/>). Desde noviembre de 1997 es reconocido por Microsoft como MVP (Most Valuable Professional) de Visual Basic. Es orador internacional de Ineta con la que imparte charlas en muchos países de Latinoamérica y es mentor de Solid Quality Mentors, la empresa líder en consultoría y formación.



Es redactor de la revista dotNetManía, y ha publicado dos libros con Anaya Multimedia: Manual Imprescindible de Visual Basic 2005 y con Solid Quality™ Press: Novedades de Visual Basic 9.0.

Aprenda C# 3.0 desde 0.0 – Parte 3, lo nuevo contiene toda la información sobre la nueva versión del compilador de C# que se incluye en Visual Studio 2008. Novedades que se desglosan en tres partes, en la primera se cuenta todo lo referente al entorno de desarrollo, qué novedades se incorporan y cómo aprovecharlas desde el punto de vista del programador de Visual C#. En la segunda se explican con todo lujo de detalles, todas las novedades del lenguaje, desde las nuevas instrucciones, hasta las novedades "necesarias" para trabajar con la nueva tecnología de las consultas LINQ; tema que se cubre a fondo en la tercera parte del libro, centrándose principalmente en las instrucciones de consulta incluidas en el compilador para dar soporte a las diferentes tecnologías relacionadas con LINQ: LINQ to Objects, LINQ to XML, LINQ to DataSet, LINQ to SQL y LINQ to Entities.

Solid Quality Mentors es el proveedor global confiable de servicios de formación y soluciones avanzadas para las aplicaciones de misión crítica, inteligencia de negocios y alta disponibilidad de su empresa.

Solid Quality Mentors combina una amplia experiencia técnica y de implementación en el mundo real, con un compromiso firme en la transferencia de conocimiento, dada la combinación única de dotes lectivas y experiencia profesional que nuestros mentores ofrecen. De este modo no solamente ayudamos a nuestros clientes a solventar sus necesidades tecnológicas, sino que somos capaces de incrementar la capacidad técnica de sus profesionales, dándoles una ventaja competitiva en el mercado. Por eso llamamos Mentores a nuestros expertos: por su compromiso en posibilitar el éxito de su empresa y de sus equipos profesionales a largo plazo.

Nuestros expertos son profesionales reconocidos en el mercado, con más de 100 premios MVP (Most Valuable Professional) obtenidos hasta la fecha. Se trata de autores y ponentes en las conferencias más importantes del sector, con varios centenares de ponencias presentadas en conferencias internacionales durante los últimos años. Sirva como ejemplo, que nuestros expertos han diseñado los últimos cursos oficiales de Microsoft SQL Server 2005, y que han tenido el honor de impartir cursos de formación a empleados de Microsoft en todo el mundo, sobre sistemas de bases de datos y tecnologías de desarrollo de aplicaciones. Además, han participado de tres Training Kits oficiales de Microsoft sobre SQL Server 2005, así como en más de 15 libros de Microsoft Press en todas las áreas de la plataforma de acceso a datos de Microsoft.

